Dynamic User Interface
Supporting Type Analysis for
RDFGears

Master’s Thesis

Marojahan Mula Timbul Sigiro
<<Page left blank intentionally>>
Dynamic User Interface Supporting Type Analysis for RDFGears

THESIS

Submitted in the partial fulfillment of
The requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Mrojan Mula Timbul Sigiro
Born in Lumban Sigiro, Samosir, Indonesia
Dynamic User Interface Supporting Type Analysis for RDFGears

Author: Marojahan Mula Timbul Sigiro
Student ID: 4125770
Email: marojahan@gmail.com, marojahan@del.ac.id

Abstract

This thesis describes the design and implementation of an interactive, dynamic and functional web-based Graphical User Interface for RDFGears. RDFGears is a data integration framework for the semantic web. It allows users to express and execute complex data algorithms without the need to be concerned with the implementation details. The framework has its own language namely RDF Gears Language (RGL) that is used to formally define the data integration process as a workflow and an engine to evaluate the workflow and perform the data integration.

This thesis proposes an application that can be used as an interface between RDF Gears and users. We introduce a new design of the user interface and new features to improve the usability of RDF Gears. More support for editing nested workflows, query editor, interactive nested node copier and property editor are some of the features that are introduced to improve the usability and the interactivity.

RDFGears Function Definition Language is introduced to improve the extensibility of RDF gears engine. Core functions or user defined functions of RDF Gears can be extended without worried of the UI support. We also introduce a type system that implemented in the user interface. It performs the type checking operation during the workflow construction and provides live feedback. An evaluation to evaluate the correctness of our implementation is presented. The evaluation shows the improvement that we have made and the usability of the new features.

Graduation Committee:
Chair: Prof. dr. ir. G.J. Houben, Faculty EEMCS, TU Delft
Supervisor: Dr. ir. A.J.H. Hidders, Faculty EEMCS, TU Delft
Committee Member: Ir. H.J.A.M. Geers, Faculty EEMCS, TU Delft
Preface

This master thesis is the final work of my study at Delft University of Technology. It started at December 2012 when I decided to do my master thesis within the Web Information System (WIS) group. I have worked for some years as a teacher in a small university in Indonesia before I came back to University as a student again. I always like teaching but I always have passion for engineering especially programming and web technologies. That is the main reason why I chose this topic which is more engineering related topic rather than theoretical research.

Before I started the actual research and implementation, the idea that I had in mind was to build the application from scratch by using basic libraries for building web application. I went through the process of try and error to design the basic architecture of the application and to find the suitable libraries that I can use in my project. It took a few months for me to get a solid idea of how I was going to design and build the application. It was a slow process at the beginning but after a long and lots of work, I am here ready to present my final work.

I really want to take this opportunity to acknowledge all of those who have given me support and guidance. I would like to thank TU Delft for admitting me and to my sponsor who has provided funds for my study and always gives me support whenever I encountered any problems.

I am grateful to my supervisor Jan Hidders. I thank him for the patience and passion with which he supervised me. I have learned a lot of things from him, not only him as my thesis supervisor but also him as a teacher. I did many mistakes and sometimes out of ideas but through many discussions he helped me to solve the problems. My gratitude also goes for Prof. Houben, the chair of the research group, for allowing me to do this thesis in the WIS group. I also would like to thank Hans Geers for all the support that I got during two years of my Master study and also for joining my thesis committee. I also want to take this chance to thank Bihao and Vlad, for all the supports, “crazy ideas”, “not so crazy ideas” and motivations which are really helpful.

Finally, I would like to thank my parents and my brothers. Without all of your supports and motivations, it is impossible for me to complete my works here in Delft.

Marojahan Mula Timbul Sigiro
Delft, the Netherlands
October 29, 2012
# Contents

Preface ........................................................................................................... i

Contents ........................................................................................................ iii

List of Figures .................................................................................................. v

List of Tables .................................................................................................... vi

1 Introduction .................................................................................................. 1

1.1 Motivation ................................................................................................ 1

1.2 Problem definition and goal ..................................................................... 2

1.2.1 Research Goal .................................................................................. 2

1.3 Organization of this thesis ....................................................................... 3

2 Background ................................................................................................... 5

2.1 Semantic Web ........................................................................................ 5

2.2 RDF Gears ............................................................................................ 6

2.3 Scientific workflow system ...................................................................... 7

2.4 Type System .......................................................................................... 8

2.5 Related Work ........................................................................................ 10

2.5.1 Taverna ........................................................................................... 10

2.5.2 Unipro UGENE .............................................................................. 11

2.5.3 Galaxy ............................................................................................ 11

2.5.4 DERI Pipes ..................................................................................... 12

2.6 Conclusion ............................................................................................. 12

3 RDF Gears User Interface ........................................................................... 15

3.1 RDF Gears User Interface Requirements .............................................. 15

3.2 Design of the User Interface .................................................................. 17

3.2.1 Graphical User Interface Components ........................................... 18

3.2.2 RDFGears Function Definition Language ...................................... 23

3.3 Implementation ...................................................................................... 28

3.3.1 Technical Architecture ................................................................... 29

3.3.2 Core Classes of RDF Gears User Interface ..................................... 31

3.4 Conclusion ............................................................................................. 33

4 Type System ................................................................................................ 35

4.1 RDF Gears Types Overview .................................................................. 35

4.2 Type checking algorithm and rules ......................................................... 36
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.1 Type renaming</td>
<td>36</td>
</tr>
<tr>
<td>4.2.2 Type Substitution Table</td>
<td>37</td>
</tr>
<tr>
<td>4.2.3 Type Unification</td>
<td>38</td>
</tr>
<tr>
<td>4.2.4 Handling Recursive type</td>
<td>40</td>
</tr>
<tr>
<td>4.3 RDF Gears Types Declaration</td>
<td>40</td>
</tr>
<tr>
<td>4.4 Live Type Checking Mechanism</td>
<td>42</td>
</tr>
<tr>
<td>4.5 Conclusion</td>
<td>43</td>
</tr>
<tr>
<td>5 Evaluation</td>
<td>45</td>
</tr>
<tr>
<td>5.1 Comparison with existing RDF Gears User Interface</td>
<td>45</td>
</tr>
<tr>
<td>5.1.1 Constructing RDF Gears Workflow</td>
<td>45</td>
</tr>
<tr>
<td>5.1.2 Debugging Workflow</td>
<td>47</td>
</tr>
<tr>
<td>5.1.3 Executing Workflow</td>
<td>47</td>
</tr>
<tr>
<td>5.2 Usability of the Features</td>
<td>48</td>
</tr>
<tr>
<td>5.2.1 Flexible RDFGears Function Definition</td>
<td>48</td>
</tr>
<tr>
<td>5.2.2 Type Checking</td>
<td>48</td>
</tr>
<tr>
<td>5.3 Conclusion</td>
<td>49</td>
</tr>
<tr>
<td>6 Conclusion and Future Work</td>
<td>51</td>
</tr>
<tr>
<td>6.1 Conclusions</td>
<td>51</td>
</tr>
<tr>
<td>6.2 Future Work</td>
<td>52</td>
</tr>
<tr>
<td>References</td>
<td>55</td>
</tr>
<tr>
<td>Appendix A: XML Schema for RDFGears Function Definition Language</td>
<td>57</td>
</tr>
<tr>
<td>Appendix B: Example of RDFGears Workflow source that produced by the application</td>
<td>59</td>
</tr>
<tr>
<td>Appendix C: Example of RDFGears Function Definition</td>
<td>61</td>
</tr>
</tbody>
</table>
List of Figures

Figure 3.1 High level architecture .................................................................18
Figure 3.2 Full view of the Graphical User Interface ......................................18
Figure 3.3 Nested workflow .........................................................................19
Figure 3.4 User Interface Navigation ............................................................20
Figure 3.5 Log panel and visual notification ..................................................22
Figure 3.6 String parameter interface layout ...............................................25
Figure 3.7 Text parameter interface layout .................................................25
Figure 3.8 Query parameter interface layout .............................................25
Figure 3.9 Fields parameter interface layout ..............................................25
Figure 3.10 InputFields parameter interface layout ....................................26
Figure 3.11 List parameter interface layout .................................................26
Figure 3.12 Relation of RDF Gears UI with GWT and third party libraries .......29
Figure 3.13 Communication model between frontend and backend ..............30
Figure 3.14 Storage directory tree ...............................................................30
Figure 3.15 Class diagram of the Workflow graph editor core classes ..........31
Figure 3.16 Class diagram of Node property editor core classes ..................32
Figure 3.17 Class diagram of type system core classes ................................33
Figure 4.1 Live Type Checking Flow Chart ..................................................42
Figure 5.1 RDF Gears UI Comparison ..........................................................46
Figure 5.2 Type error ....................................................................................49
List of Tables

Table 3.1 Mapping between RDFGears Function Definition Language and RGL Expression ........................................................................................................................................................................28
Table 4.1 Type declaration in XML.................................................................................................................................................................................41
Table 5.1 Summary of improvements and new features ........................................................................................................................................47
1 Introduction

1.1 Motivation

In recent years the Semantic Web has been growing rapidly. Vast amounts of data that used to be available only in traditional form are being converted to RDF data (DBPedia, LinkedMDB, etc.)[5,22]. But, even though a vast amount of Web data is available, there are some problems that need to be taken into consideration. The web data is distributed, it comes from multiple autonomous and heterogeneous data source [4]. The data are located in different servers and accessible via different kind of mechanism (SPARQL, JSON, XML, etc.). There is also a possibility that the data is incomplete and need to be reformatted, RDF files may contain missing links and use different ontologies. So although the data are available, user need a way or tools to enable them to access and use the available data without encounter any problems above. There are some tools that offer the ability to integrate the data, retrieve data from multiple sources, clean the data or modify and validate the data. RDFGears is one of many tools that offer these functionalities.

RDFGears is a data integration framework for the semantic web. It allows the user to express and execute complex data algorithms without the need to be concerned with the implementation details. The framework has a data integration language namely RDF Gears Language (RGL) that can be used to formally define the semantic web data transformations. The framework also has an engine (The RDF Gears Engine) to evaluate RGL expressions and thus perform the actual data integration. RDFGears can be considered as an all in one tool that can be used for semantic data integration. However as other scientific tools, there are some features that need to be added or perfected. The existing Graphical User Interface (GUI) is lack of flexibility and it makes RDFGears relatively inconvenient to use. Moreover, it has an outdated library that causes error if the application is opened with a certain kind of web browsers.

This thesis attempts to improve RDFGears by provides a functional and dynamic web-based Graphical User Interface (GUI) and backend service that will extend the usability and the scalability of RDFGears engine. One of the main benefits that it offers is to enable researches and other users to extend the RDFGears engine. In the existing user interface, the process of creating new functions is complicated because the whole application need to be modified that includes the UI. We propose a different approach by creating a communication protocol between the engine and the UI, thus the application will be more flexible and extensible. We also propose a new design of the user interface to avoid cluttered layout and also support for editing nested workflows. The GUI also offers live type checking functionality that will enable us to do the type checking during the workflow construction. As part of the type system, we propose an update to the existing type system which is new and more flexible notions of type and new typing rules.
1.2 Problem definition and goal

This master thesis proposes a functional GUI which will improve the RDFGears. The basic idea of creating the GUI for RDFGears is to provide a graphical interface to create and manage RDFGears Language expressions. The RDFGears expression itself describes a workflow. The GUI will enable users to create a workflow in an interactive way with nodes, node’s property editor and connectors. The GUI also has a built-in type checking system to validate the correctness of the workflow. The users will be able to define input and output types of a certain node and the GUI will do the type-checking and construct the type of the workflow while the user is constructing the workflow graph. The idea is also to improve the scalability of the RDFGears by enable the users to add or remove certain RDFGears function more easily. The complete description of workflow nodes that represent RDFGears functions will be defined in XML files and the GUI will able to parse the files and transform it into graphical workflow nodes.

1.2.1 Research Goal

The research presented in this thesis was coordinated at TU Delft, as part of ongoing research in the Web Information System (WIS) group. The following main research goal was formulated:

Research goal: Design and develop an interactive, dynamic and functional web-based Graphical User Interface for RDFGears, an RDF transformation and integration workbench.

Even though the RDFGears is focusing on semantic web data transformation and integration but this research is not directly related to the semantic web data itself. As mentioned before, the main challenge is to improve the usability and the scalability of the RDFGears.

In order to understand the problem and achieve the research goal, there are some research questions that need to be answered. How does the RDF Gears engine work? What kinds of libraries and framework are needed that able to create high quality web application? And so on. The following list shows the background knowledge that we need to gain in order to understand our problem and the main questions that we need to answer in order to achieve our goal.

- We need basic background knowledge what is RDFGears and how does it work? What are the components of RDFGears framework? What is scientific workflow and what is needed to create a graphical workflow graph editor? What is type system in general and how does it work?

- Based on the background knowledge we need to provide answers to the following questions: is it possible to create a rich GUI without altering current RDFGears engine implementation? What features and functionalities are needed to create a
functional RDFGears User Interface? Is it possible to implement the type checking in the user interface?

All the questions that presented above also represent the steps that we took to achieve our goal.

1.3 **Organization of this thesis**

The structure of this document is as follows. First, chapter 2 presents the basic background knowledge of the semantic web, the concept of scientific workflows and type system. This chapter also introduces RDF gears framework. The architecture of the RDF Gears is described in this chapter. To have a better understanding in workflow editor, we also discuss the related work that includes several workflow editor applications.

Chapter 3 presents the discussion of our application. The chapter is divided into three parts. First, we define a set of requirements that need to be existed in our application. We form our requirements by first analyzing the existing in the existing RDF Gears UI to identify the problems that we need to solve. Then we extend the requirements based on our study. In the second part, we propose our design and implementation. In this part we present the design of the user interface, our approach to provide solutions. In the third part, we discussed the implementation. The architecture of the application, the core classes, and the libraries are explained in this part.

Chapter 4 presents the type system. First we introduce the RDF Gears types, the types that we formulate based on the existing RGL types. Then we discuss the algorithm and the rules of the type system. We explain the substitution and the unification rules and how we handled the case of recursive type. We also present the type definition language, a set of XML tags to define the types. At the end of this chapter, we discuss the implementation of the live type checking system and how the type checking is integrated with the workflow editor.

Chapter 5 presents the evaluation. The evaluation is focus on evaluating the correctness of our implementation. We compare our application with the existing RDF Gears UI and present the result. In this chapter, we also evaluate the usability of the features that we have implemented.

Chapter 6 concludes the thesis. It provides a summary with conclusions and discusses future work.
This chapter discusses the background concept that needed and also the existing works that related to this thesis topic. Section 2.1 presents a short introduction to the Semantic Web Technologies, Section 2.2 introduce the RDFGears framework, Section 2.3 present an introduction to scientific workflows and Section 2.4 presents the general overview of Type System. On the Section 2.5 we present the result of our study on various existing related work and Section 2.6 concludes the chapter.

2.1 Semantic Web

The World Wide Web invented by Tim Berners-Lee in 1989. It is a web of documents where each document is identified by an URI. The documents are stored and served by a web server and they can be retrieved by a web browser which communicates with the server using the HTTP protocol. HTML documents are machine-readable documents which have structures that describe the data that should be shown to the user and how the data will be displayed. The main idea of creating HTML documents is to create a document with well formatted layout in a way such that humans can understand the content easily. HTML also enables a document to be linked to another document. This is very useful for human users. However, to a machine, the layout and the link is useless in order to understand the semantic meaning of the document. Computer systems cannot deduct any information from these structures that are related to the semantic meaning of the document.

Semantic web technology, on the contrary, allows the representation of a web of data. It is driven by a vision which to allows machines to understand and respond to complex human requests based on the meaning of the requests and the meaning of the web data. This vision is attempted to be realized by augmenting existing data that we now have on the web. What is expected from this augmentation is that it will link web documents with ontological annotations and provide semantic meaning of the relation between documents that is understandable by machines. RDF is used to represent the data, to add semantic annotations to the documents and SPARQL is used to query the data. RDF, short for Resource Description Framework is a framework for expressing relations between resources and data. RDF allows describing web-based resources by creating a directed and labeled graph model connecting resources (URIs) with other resources as well as literal data. SPARQL, a recursive acronym stands for SPARQL Protocol and RDF Query Language. It consists of both a protocol and a query language. The SPARQL query us a syntactically-SQL-like language for querying RDF graphs via pattern matching [9]. RDFGears is focusing on using these two technologies to create a more powerful data transformation and integration tool.
2.2 RDF Gears

RDF Gears is a workflow system for integration and transformation of RDF data. It aims to provide a data integration framework that allows expressing and executing complex data integration algorithms without the need to be overly concerned with implementation details [15]. It provides more expressivity and flexibility in integrating Semantic Web data. RDF Gears is a framework and can be divided into three main components: RDF Gears Language (RGL), RDF Gears Engine and RDF Gears graphical user interface [15].

**RDF Gears Language (RGL)** is a data integration language, a language that can be used to formally define the RDF data transformation and integration. The transformation and the integration defined as a sequence of functions execution where output of a particular function become input to others, thus it will form a workflow. RGL is a workflow language that combines processors and form a workflow. Processors represent a function that will execute a certain process on its input. A processor has zero or more inputs and one output. The inputs are defined as input ports that can be connected with a particular output port of another process and it will represent the flow of data from one processor to another. The connections between processors will form an acyclic directed graph; the workflow. A workflow itself is considered as a function that can be used as a processor in a certain workflow thus will form a nested workflow which will be one of the challenges on this thesis, to find a way to modify nested workflow easier.

RGL’s data model has values and type system. The values of RGL combine the value system of the Names Nested Relational Calculus (NNRC) with the RDF data model [15]. The RGL values can be defined as follows:

- Every *RDF value* is a *value* where the *RDF value* is the union of pairwise disjoints set of *U* of *URIs* and *L* of *literals* [6].

- An RGL graph is a *value* where RGL graph is a finite subset of \((U \times U \cup L)\).

- *If r is a row over values*, then \([r]\) is a *value* that called a record.

- A *finite multi set of values is a value* that called a bag, with a restriction that the bag only contains values with the same type.

- A *special element \(\omega\) is a value called null value*.

RGL uses static type system that allows the evaluation of the RGL expression to be performed before the actual execution. The evaluation will ensure the correctness of the RGL expression. RGL introduces the types *U* for URI’s, *B* for blank nodes, *L* for literals, *V* as an abstract super type for RDF values of types *U*, *B* and *L*, and *G* for RDF Graphs. From this set of types, the RGL types are inductively defined as follows:

- *V* and *G* are types

- *If \(\rho\) is a row over types, the \(\text{Record}(\rho)\) is a type*
• If \( T \) is a type, the \( \text{Bag}(T) \) is a type

The semantic \( [T] \) of the RGL types \( T \) is defined as follows [15]:

• The semantics of the type \( V \) is \( [V] = U \cup L \cup \omega = \text{RDFValues} \) i.e. the set of all RDF values (except the blank nodes) and the null value

• The semantics of the type \( B \) is \( [B] = \{\text{True, False, } \omega\} \) i.e. the Boolean values and the null values

• The semantics of type \( G \) is \( [G] = \text{Graphs} \cup \omega \) i.e. the set of all graphs and null value

• For types of the form \( \text{Record}(\rho) \) the semantics \( [\text{Record}(\rho)] \) is the union of \( \{\omega\} \) and the set of all record \( [r] \) with \( \text{domain}(r) = \text{domain}(\rho) \)

• For types of the form \( \text{Bag}(T) \) the semantics \( [\text{Bag}(T)] \) is the union of \( \{\omega\} \) and the set of all finite bags over \( [T] \)

All RDF Gears processors’ inputs and output has types which defined within the processors implementation i.e. embedded in the functions implementation code. One of the challenges of this thesis is to find a more convenient way to define processors inputs and output types and makes this feature as part of the RDF Gears graphical user interface.

**RDF Gears Engine** is and interpreter for RGL, the application that performs the evaluation of the RGL expressions. The engine is the application that does the actual data integration by executing the functions defined in the RGL expression.

**RDF Gears Graphical user interface** is a web application that allows convenient construction of complex RGL expression. The GUI is merely a tool to makes the construction of RGL expression easier. It use drag and drop operation to edit the workflow and visualize the connection between processors as a visual directed graph. The GUI will produce an XML file which is the workflow source code that represents the RGL expression. The GUI itself is a separate implementation from the RDF Gears engine thus it is possible to execute a workflow without the UI. The current GUI is a prototype that was build based on the code of DERI Pipes [17]. It is implemented in Java and built on top of the ZK Framework, an AJAX framework that allows the creation of web application in Java without the need to write custom JavaScript code.

The current UI is a functional application with some limitations. These thesis aims to build a new UI from scratch which expected to be able to replace the current UI and also by adding features which not existed in the current application.

### 2.3 Scientific workflow system

A workflow is an abstract description of steps required for executing real-word process, and how the flow of information between them. Each step is defined by a set of activities that need to be conducted. Visually, a workflow usually defined as a directed graph, each step is
denoted by a graph node which has input and output. Connecting particular node to other nodes define the flow of information that will happen in the real-world process.

Scientific workflow system is a specialized form of workflow management system designed specifically to compose and execute a series of computation or data manipulation steps in a scientific application [11]. As the technology advances in both scientific instrumentation and simulation, the amount of scientific datasets is growing exponentially. This is one of the reason that scientific workflow is needed and become popular in scientific computation as more scientist are relying on workflow system to conduct they daily analysis on a vast amount of datasets. Furthermore, there are other motives that make the scientific workflow system become one of the main tools for scientific analysis. These include:

- Provides an easy-to-use environment for the scientists as they also able to create their own workflows by using the system
- Provides interactive tools for the scientists enabling them to execute their workflows and view their results in real-time
- Simplify the process of sharing and reusing workflows between the scientists
- Enable scientists to track the provenance of the workflow execution results and the workflow creation steps

The workflow engine is one of the most important things in a workflow system because at the end it is responsible to execute all the steps and responsible to maintain the flow of the information. However this is not part of this thesis. This thesis will focus on creating a frontend of an existing workflow engine and aim to create an easy-to-use environment, interactive and extendible. The frontend or the User Interface of a scientific workflow system is considered as one of the main factor to determine the adaptability of a system [12].

### 2.4 Type System

A computer system or a computer program is a system that processes values or data, in which the data itself have many kinds of formats or types. A certain data type may be compatible with other data types but it may also incompatible. For example, suppose we have a value or the a result of computed expression with a certain type and this value will be used as input in other step or expression which in turn expects input values of another type. Then it becomes important to know if the types are compatible because assigning a value to a variable with incompatible data types will lead to an error or a crash. A type system is a system that associates a type with a value and also with expressions. By examining the type of each value that will be processed, error can be minimalized or prevented which is the fundamental purpose of the type system [13]. A type system controls
the ways typed programs may behave, and makes behavior outside these rules illegal in purpose to avoid the error or crash due to type error.

Types can also serve as a form of documentation, since types illustrate the intent of the programmer. For instance, if a programmer declares a function as returning a string, this means that the returned value should not be used in an arithmetic operation. The same mechanism is applied in workflow system where a node in workflow graph represents a process with input and output, defining the type of the input and output beforehand will provide information about the data that flows from one process to another. The programmer will be able to understand the types of data a process required or produce, thus he will be able to prevent errors from happening.

The type system is not merely an annotation that defines the type so the programmer can understand and avoid errors. Type systems have other characteristics that distinguish them from other kinds of program annotations. There are three basic properties expected from of any type system [13]:

- The type system should be decidable verifiable there should be an algorithm (called a type checking algorithm) that can ensure that a program is well behaved. The purpose of a type system is not simply to state programmer intentions, but to actively capture execution errors before they happen.

- The type system should be transparent a programmer should be able to predict easily whether a program will do the type checking. If it fails to type check, the reason for the failure should be self-evident. The type checking algorithm should provide information regarding to the failure.

- The type system should be enforceable type declarations should be statically checked as much as possible and otherwise dynamically checked. The consistency between type declarations and their associated programs should be routinely verified.

The process of verifying and enforcing the types constrains –called type checking – can be performed during the compilation process or during runtime. Type systems that perform the type checking during the compilation need the type’s information to be predefined explicitly in the program code where the runtime type checking does not need it to. Thus, based on how the types are determined and checked, type systems can be divided into two categories: static typing and dynamic typing [14]. In a statically typed language, every variables or functions is bounded to a type. This information enables the type checker to perform the checking at compile time thus it will allow the errors to be caught early in the development cycle.

Opposed to the static typing mechanism, dynamic typing does not have the type information predefined. The type checking is performed during runtime by computing the values a variable receives or values returned by a function. In dynamic typing, the types are associated with the values, but variables have a generic type (object) thus it can refer to a value of any type. The run-time check in dynamic typing can potentially be more
sophisticated compared to static type system as they use dynamic information generated during runtime as well as the any information from the source code. On the other hand, runtime checks only assert that conditions hold in a particular execution of the program, and the checks are repeated for every execution of the program. This can be considered as a weakness of dynamic typing over static typing or maybe an advantage if looked from a certain point of view. However the comparison between the two type system, regarding of the advantage and disadvantage is not a part of this thesis.

2.5 Related Work

In the recent years, the research and the development in scientific workflow systems are growing [10]. Researchers and corporation are introducing their new scientific workflow system. The results of these researches often produce tools that can be used to assist researcher in their research. Each of the system has their own standard and protocol, even though some of the systems provide a support in importing or editing existing workflow file that has a different format. One thing that most of the tools have in common is the workflow editor. Even though the workflow file stored in a certain format, the visualization of the workflow graph is relatively the same. A node represents a function of process and the connection represents the dataflow. The following subchapters will try to provide a brief introduction of some of the scientific workflow system tools. The explanation will focus on the Graphical User Interface; the workflow editor and the workflow file management system.

2.5.1 Taverna

Taverna is a scientific workflow system that focuses on integration of molecular biology tools and databases [18]. This tool can be used to deal with large amounts of data that can be fetched from bio-informatics Web services and support different kind of protocols (SOAP/WSDL, REST Web services, BioMoby, Biomart, SoapLab Web Services).

Taverna workbench has a desktop based client application. The application will use remote web service thus it doesn’t need the data to be store locally. Taverna client application enables the user to add and remove remote web services manually based on the user needs. The workflow editor uses remote services as node. A node can be connected to another node – which is a remote service – by connecting the node’s ports through a drag and drop fashion. User can add workflow input port and workflow output port and connect them to other nodes to form a complete workflow.

The appearance of the workflow editor is clean and neat. The workflow graph editor only contains boxes that represent the nodes and the arrow line to represent the connection. The workflow editor applies the automatic graph layout. This feature could be a useful functionality since users do not need to arrange the graph layout manually, but this could also be a minus because it will restrict user from customizing the graph layout. The nodes property can be modified in a separate panel which give an impression of categorized and
focused. The client application can perform validation operation that can be triggered manually to detect if the current workflow contain any error. It has a similar purpose as the type checking to detect and prevent error from happening during runtime.

Taverna workflow editor has the ability to visualize nested workflow in a single workflow editor. Even though it need separate editor to edit nested workflow, this feature is a great addition which help the user to understand the entire workflow. The latest Taverna client has the feature to share and find workflows through myExperiment service. myExperiment is a Virtual Research Environment, a community social network for researchers that enable them to share their workflows.

2.5.2 Unipro UGENE

Unipro EUGENE is software with the main goal to assist molecular biologist to manage, analyze and visualize their research data without need of expertise in bioinformatics [19]. It integrates support for multiple biological data formats and like Taverna, it allows the retrieval of data from remote data sources. The software also integrates widely used bioinformatics tools which makes it a complete all-in-one tool for researchers. Among many of the tools it offers, workflow designer is one of its key components. It can be used as a visual tool for building complex analysis pipeline. Workflows constructed with Ugene workflow designer can be offloaded to a remote computational resource that running UGENE Remote Service or can be executed locally.

Ugene Workflow Designer user interface has three main components; Palette that contains grouped workflow elements, Scene the main drawing canvas to construct the workflow graph and Property Editor that provides information about currently selected workflow element and allows configuring it. This separation makes the user interface organized and relatively easy to use. The adding of workflow nodes and connecting nodes can be done in drag and drop fashion. It offers flexibility to user as user can arrange the layout of the graph manually. The visual representation of the workflow called schema and it is constructed from predefined elements which each element represents an algorithm to process the data or function to read and write data. Elements can be added or modified, however it is impossible to add existing workflow as an element, thus nested workflow is not supported.

2.5.3 Galaxy

Galaxy is a software system that provides sophisticated and computationally intensive data analysis support for researchers in molecular biology field [20]. The support is provided through a web application that provided as a publicly available Web service or a downloadable package that can be deployed in individual laboratories. Galaxy is also a data integration platform for biological data. It supports data uploads from the user's computer, by URL, and from many online resources (such as the UCSC Genome Browser, BioMart and InterMine). As other scientific workflow system, galaxy has a workflow management system in which users can create and edit workflow with workflow editor, store, and share
the workflow. The workflow editor is built using HTML and JavaScript for the frontend and python programing language for the backend system.

The Galaxy workflow editor is simple yet powerful. Workflow graphs can be constructed in drag and drop fashion. The workflow editor interface has four main components: the tools panel that provides all the tools than can be added as a workflow graph node, the canvas where the workflow graph constructed, the property panel where the node properties displayed and modified and the mini display where the complete small version of the workflow graph displayed. The last component is very useful where it can be used as navigation tools for a large workflow graph. The current galaxy workflow system does not support nested workflow which is a drawback for galaxy as a scientific workflow system. The workflow system enable user to download the workflow graph in JSON file format which can be imported into another galaxy server and user can export the workflow to myExperiment environment which makes it convenient for researchers to save and share their workflow.

2.5.4 DERI Pipes

DERI Pipes is an application to create data integration mashups [17]. It provides operators to work on various data formats like XML, RSS, RDF and JSON. The data integration process is constructed by combining the processors to form a workflow. The workflow graph represents the sequence of processors execution and the dataflow between them.

The current RDF Gears user interface was built based on the DERI Pipes user interface code. The workflow graph can be created in drag and drop fashion and the result can be exported in XML file format. The workflow editor has two main components, the list panel where the list of workflow and operators can be found, and the canvas where workflow graph is edited. The graph nodes property editor is embedded within the node in the form of HTML form. This feature enables the users to modify the processors properties directly inside the nodes without accessing another panel. This will save the step needed in editing the property but it give an impression of untidy and not very clean. Another drawback is that the code is using outdated library which will cause errors on some modern browsers. Nested workflow is supported by adding a certain workflow as node but the visualization of nested workflow is not supported. The user interface itself is easy to use but the lack of features and the outdated library are big drawbacks for DERI Pipes as a workflow management system tool.

2.6 Conclusion

The vision behind Semantic Web is to enable machines to understand the data on the web and to be able to respond to human request based on its meaning. As the development of the Semantic Web grows and more data are being converted into RDF data, this vision has more and more chance to be realized and to be used in daily life. However the needs of reliable and useful tools in the process cannot be avoided. Researchers need an assistant to
deal with large amount of data with different kind of formats and sometimes with errors that need to be fixed.

RDF Gears is one of many tools that can be used for this purpose. It offers the ability to deal with RDF Data, to do the data integration, to filter and manipulate the data. As the other tools that are introduced in this chapter, the engine itself is not enough. The user interface is a feature that will be a great addition to the system, to make the system easier to be used without needs of high level of technical skills. There are similar tools that offer many features in order to help the user especially in dealing with the workflow. A simple and clean workflow graph editor is a feature that offered by most of the tools. Some tools support the nested workflow and others tools don’t but it is considered as a very helpful functionality. All these features are offered to simplify the use of the system thus will increase the value and the usefulness of the system as an assistant for the researchers.
3 RDF Gears User Interface

This chapter presents the RDF Gears User Interface, a new user interface to complete the RDF Gears workbench. First, section 3.1 presents the requirements. Problems and questions are presented and they will be the base to define all the requirements. This section should make the rest of the discussion clearer as it will be the main reference for the rest of the discussion. Section 3.2 discusses our design. The design of the user interface is presented and the approach that has been done to provide solutions to all requirements that we have defined. This section is supposed to give a clear explanation of what components are really needed and implemented in the user interface. Section 3.3 presents the technical explanation of our implementation. This section presents the technologies that are used and the technical architecture of the application. The last section concludes the discussion.

3.1 RDF Gears User Interface Requirements

The main requirement is to provide a graphical user interface that enables us to interact with RDF Gears engine. It enables us to use all the functionalities of RDFGears without the need of manually write the RGL Expression and execute the engine through its command line interface which is relatively more complicated. As briefly introduced in chapter 1, the current RDF Gears user interface has offer functionalities that enable us to interact with the engine but it still has some limitations and contain errors. The limitations and the problems that we have encountered with the current RDF Gears user interface are listed as follows:

- **Cluttered**, the current RDF Gears workflow diagram editor embeds the node’s property editor inside the node. This approach can be considered useful because it provides all the information that we need regarding to the diagram, in a single page. At the same time it takes more space thus it leads to a cluttered interface because all the properties’ input forms are embedded in a small area of the nodes.

- **Uninformative**, the current interface does not provide any information about the node’s description. The description can provide information about how a certain processor works or what are the inputs that needed. The absent of this functionality will lead to a high learning curve for new users in order to understand the application.

- **Unscalable**, all the operator’s nodes are defined programmatically. Each of the processors has a java class that associated with it. This class will handle all the actions that applied to it and it is also responsible to layout the parameter’s input form. The list of the processor is defined manually by altering the java class. This approach is inflexible: it is impossible to add a certain processors without manually
altering the code and recompile the application thus makes it unscalable. This limitation will become an obstacle to the future development of RDF Gears.

- **Outdated library.** the current RDF Gears user interface was built based on DERI Pipes code. It was built using old ZK framework and Draw2d library [15]. We have tested the application with the latest version of various modern web browsers (Internet Explorer, Google Chrome, Safari, Firefox, and Opera) and as the result it was only working with Firefox web browser. The application showed noticeable errors or did not work completely on other web browsers. This is an unacceptable condition as researchers may work with different web browsers other that Firefox.

The new RDF Gears user interface is expected to be able to overcome the current application's limitations and the errors. Besides that, there are also other requirements which expected to be able to enrich and to make the user interface more useful. The complete requirements for the RDF Gears user interface are listed as follows:

- **Workflow graph editor.** the ability to visualize RGL Expression as workflow diagram is the main purpose of the user interface. This editor should enable the users to edit RGL Expression in an easier way. RGL Expressions will be visualized as directed workflow graphs where nodes of the graph represent the processors and the connections represent the dataflow between processors.

- **Support editing nested workflow.** this is a great feature for RDF Gears user interface as users will able to explore workflows more conveniently. Editing and visualizing nested workflows can be done in many approaches. The main requirement is to provide functionality that enables users to deal with nested workflows. Current RDF Gears User Interface does not support the editing of nested workflow in a single instance of the application. The nested workflow must be opened in a new separate browser window.

- **Organized and clean interface layout,** a clean, simple and elegant user interface will give an additional value because it will make the users more interested in using the application. A cluttered user interface should be avoided but without limiting users to get any information that they need.

- **Informative** The application should provide all the information that needed by the user. Each processor should have description in order to help user to understand how it works or what the processor does. The application must provide enough information on what is currently happening, give feedbacks when a certain operation failed or succeeded.

- **Scalable** One of the biggest problems with the current user interface is that all the processors and the nodes are defined programmatically. The code must be altered in order to add a new operator or to modify existing ones. The new approach is expected to enable RDF Gears researchers and developer to extend the engine
functionality without the need to modify the user interface. A protocol must be defined that can be used by the engine to communicate with the interface. As an example, when the engine has a new processor, it simply notifies the interface to provide associated visual node that represents the processor. This approach will improve the scalability of the application. The RDFGears engine can be extended (i.e. add more functions) and the user interface will be able to adapt with this extension.

- **Type system.** RDF Gears engine has applied the type system. It is able to do the type checking, to validate whether a certain workflow is well typed and executable or not. The problem is that the type checking operation is executed during the runtime and it is part of the RGL Expression execution process. The new approach is to bring the type checking to the front and integrate it with the user interface. The type checking will be performed during the construction of the workflow to check and give a live feedback whether the workflow is well typed or not. Even though the type system is part of the user interface, the algorithm and the design of the type system is discussed on separate chapter in this thesis.

- **Supporting tools.** the application should provide small useful functionalities to make it more convenient for the users to work with the user interface. These functionalities include but not limited to SPARQL Query editor, syntax highlighting, node’s type debugger and the functionality to copy, delete and categorize workflows.

- **Browser compatibility** this feature is really important because the users should not be restricted to use a certain kind of web browser. The expected compatibility is to enable different kinds of modern web browsers to run the application. The web browsers are assumed to have supported the latest standard of HTML5, CSS 3 and JavaScript.

These requirements are the general descriptions of the RDFGears User Interface functionalities. A more detailed and more technical discussion regarding of how the User Interface was designed is presented in the following section.

### 3.2 Design of the User Interface

In general, the application is divided into two main parts: Graphical User Interface which is the actual interface that can be used to interact with the system and RDFGears Function Definition Language which will act as a communication language between the RDFGears Engine and the User Interface. Figure 3.1 shows the high level architecture of the application and its relation with the RDFGears engine. The architecture shows that the User Interface and the RDFGears Engine are using the same data source, the processors. The files contained in the data source are divided into two categories, the workflow which is files that contain the RGL XML Syntax and processor which is written using Function Definition Language to define the sets of operators and user defined functions that
supported by the engine. RDFGears Function Definition Language will be explained further in this section.

![High level architecture](image)

**Figure 3.1 High level architecture**

### 3.2.1 Graphical User Interface Components

RDFGears User Interface consists of several components and functionalities. Each component is responsible for specific task to satisfy certain requirement. Figure 3.2 shows the full view of the Graphical User Interface.

![Full view of the Graphical User Interface](image)

**Figure 3.2 Full view of the Graphical User Interface**

The components and the functionalities of the user interface are grouped into four main categories. The components and the functionality are listed and explained as follows:
• **Workflow diagram editor**

This component is the canvas where the workflow diagram is constructed. The workflow diagram layout can be modified in drag and drop fashion. The size of the canvas will be adjusted depends on the size of the web browser window size. To avoid a cluttered interface, the visualization of the diagram on the canvas is different from the current RDF Gears User Interface. Nodes are drawn with the node's name as the title instead of the node's id. Nodes only contain list of its input ports and description text for certain function parameters. Input and output ports of the nodes are specified with different icons to emphasize its meaning. The purpose of these designs is to make the workflow diagram easier to understand and also to save space and makes the diagram simpler and cleaner.

![Workflow diagram editor](image)

**Figure 3.3 Nested workflow**

Nested workflow is visualized as a normal node but with an additional property to enable it to be opened and edited. Figure 3.3 shows a workflow (Fetch sameAs) is being nested in a workflow (A Mashup). Nested workflow nodes have two additional menus on its properties: *Open Workflow* and *Copy Workflow*. The Open Workflow menu will open the nested workflow graph in a new tab. Any related opened workflow (i.e. has the workflow as a node) will be notified when a workflow is saved thus the nested workflow can be reloaded to get the latest version. This mechanism will ensure all opened workflows are synchronized and contain the latest version of the nested workflow nodes. The process of editing nested workflow will be easier and less complicated compared to the current RDFGears User Interface’s approach.

• **Navigation**

The user interface brings the concept of simple and clean interface. For example, the workflow graph node displays minimal information, the node name and the input ports name. The consequence of this approach is less information will be displayed on the screen at one time and users may loss some information they needed. To avoid this problem is to provide good navigation, to enable users to access the processors or the workflow description.
There are three main operations in using the application that need to be handled: browsing the processors list which includes the workflow, creating and debugging the workflow and editing the workflow graph. The concept of categorization is used to provide a better access to browse the processors. As shown in Figure 3.4, processors are divided into three categories: *Workflows* which contains the list of the workflows, *Operators* which contains the list of RDF Gears core functions, and *Functions* which contains the list of the user defined functions. This categorization will make the process of adding nested workflow and user defined function easier. Finding certain workflow also will be easier with the search function that will search workflows based on its id, name and description.

Creating and debugging workflows are handled with the application main menu. The *debug* menu gives access to various tools that will provides all the information that needed in order to debug the workflow. These tools include exporting workflow to XML, view last saved workflow source, show nodes id, and debug nodes input and output port type. Workflow editing process is one of the most important things that need to be handled. The editor can be used to open and edit multiple workflows. It uses tab to navigate through multiple opened workflows. As explained before that this mechanism is also used to deal with the nested workflows. The workflow canvas utilizes the mouse click mechanism to navigate through workflows and node properties without providing any text link. Node property can be displayed by selecting certain node. Clicking the empty space on the canvas will display the workflow properties. This mechanism makes the whole process easier and efficient because users do not need to navigate out of the canvas to access the information about the workflow graph and the nodes.

- **Property editor**

The previous discussion about the navigation has shown that workflow graph’s node display limited information (processor’s name and ports name). The workflow graph node will not display any other processor’s (function’s) parameters. As shown in Figure 3.2, all the properties will be displayed in a separate panel located on the right side of the editor. This
panel is also the main interface to modify the properties of the nodes and workflows. The main reason behind this approach is to enable a processor to have as many properties as it needs without effecting the visualization of the workflow graph. As an example, suppose a certain processor has ten different parameters that need to be defined by the users in order to be able to work. In the previous approach (current RDFGears Interface) the processor's node will be drawn on the canvas with ten input forms for the parameters and it will lead to a cluttered interface. The new approach is to put the properties in a separate location, the property editor, which will avoid the interface for having a cluttered visualization. In addition, more properties are able to be displayed such as processors description or a more descriptive parameter name. As the result the user interface will be more informative and organized.

- **User Interface Feedback and Utilities**

Feedback is one of the most important principles in building application user interface. A small action done by the users through the user interface may trigger a complex operation that run in the background. As an example, saving a workflow which can be done by a click operation on save menu will trigger parameter checking process to ensure all the required workflow properties has been filled with valid data. The operation will check the workflow id validation to avoid duplicate workflow id and finally the operation will store the workflow file into file system. The save operation could be finished without any error but could also failed because of unfulfilled requirement. These information, success or failed, have to be informed to users so they can decide the next action that they need to do.

One of the main challenges in building modern web application that use AJAX technique is the asynchronous communication between the frontend application on the client and the backend application that run on the server. When a request is sent to the server in the middle of an execution of a function, the execution will not be blocked. The reply from server has to be handled by using different function. To simulate the synchronization, we use a modal dialog panel to provide information of running operation and at the same time it will disable user from executing other action that may affect the whole process.

In general, RDF Gears User Interface has two mechanisms in providing feedbacks. The mechanisms are:

- Modal dialog panel, this mechanism is used for operations such as saving workflow, open workflow, copy or delete certain workflow and other operations that need user interaction.

- Feedbacks through graphical information and dedicated message panel. As shown in Figure 3.5, The feedbacks includes different line color to visualize mismatch port types, warning icon to visualize nested workflow has been modified and dedicated message panel for type checking status.
To maximize the usability of RDFGears Interface, there are several utilities that added to the application. These utilities are useful not only in constructing workflow but also to make the process of sharing and debugging the workflow become easier. These utilities include:

- Import and export workflow source, this utility is used to export opened workflow into XML format and to visualize workflow graph from its XML source. It utilizes syntax highlighted XML editor to make the process more convenient.

- Workflow type viewer, this utility is for debugging purpose, to enable user to view the types of ports type that exist in the workflow graph.

- Query editor, a useful enhanced editor to write or edit SPARQL query

- Cyclic graph detector, to ensure there is no cyclic error exist in the workflow graph

- Workflow copier, an enhanced utility to copy certain workflow. The copy process is a shallow copy but it supports a copy-replace operation on nested workflow node. This will make the process of creating a deep copy of a workflow easier.

- Type checker, this utility is the application of the type system that will do a live type checking on the workflow that is being edited.
3.2.2 RDFGears Function Definition Language

One of the big features that introduced in this thesis is the feature that enables us to create and modify the processors that used by the engine without need to reprogram and recompile the user interface. It is accomplished by introducing a language that act as communication protocol between the RDF Gears Engine and the User Interface. RDF Gears Engine implements all the functions specified in the RGL Expression as processors. These functions can be considered as the capabilities of the engine, a set of tasks that can be done by the engine. On the other part, the user interface is responsible to provide access to those functions. So the main idea behind the Function Definition Language is to enable the engine to communicate with the interface, to let the user interface know what functions are supported by the engine and what parameters are needed in order to execute them.

RDFGears Function Definition Language is a descriptive language for specifying RDF Gears functions. It is also responsible to lays out the function user interface (e.g. function name, parameter form fields, description text). The language is modeled based on the characteristic of a workflow node which has input and output. The syntax of the language is modeled based on the RGL Expression to make it easier to relate the function definition with the RDF Gears workflow source which defined in RGL Expression. The language is expressed in XML. The detailed structure of the language is described as follows (the XML schema for the language can be found in Appendix A).

Root element, the outer-most tag of the XML which is processor. It is preceded only by the xml version tag. The element has two attributes, label and category. These attribute's value will be used by the interface, the label value as the processor name text and the category for the processors list grouping.

```
<?xml version="1.0" encoding="utf-8"?>
<processor label="Local SPARQL" category="Querying">
... 
</processor>
```

Listing 3.1 Root Element

Processor Description is defined using description tag name. Description text will be visualized by node property viewer. The purpose of this element is to provide information related to the processors.

```
<description>
Write SPARQL Query, bind the variables in the query to the input fields
</description>
```

Listing 3.2 Processor description

Input element is the main element that contains most of the information about the function. It describes the functions input.

```
<inputs>
    <function type="typeName">
        <param name="parameterName" type="parameterType" label="displayLabel">
            <description>
```

Listing 3.3 Input element
Conceptually, processors input sources are categorized into two, data flow from other processors and function’s parameters. The input data flow handler is represented by the processors input port. The Input port is declared using `data` tag name. It has three attributes; `iterate`, `name` and `label`. `Iterate` contains the value of the port iterate flag which is `true` or `false`. Attribute `name` contain a unique name of the input port which also act as an identifier. The scope of the identifier is local, only within a processor. This attribute will be used both by the interface and the engine. Attribute `label` contains value which will be used as the attribute name text on the property viewer. The reason to have both `name` and `label` is to enable us to define a unique complex id and at the same time we can define a readable name. `Inputs` element may contains zero or more `data` element and each element will represent one input port. As show in Listing 3.3 line 10, the input port type is declared as child element. The discussion about the port type is presented in Chapter 4 Type System.

Function definition is declared using `function` tag. It has an attribute, `type`, which is the name of one of the processors known to the RDF Gears Engine. As shown in the example Listing 3.7, the function’s `type` attribute must have the same value as processor's `filename`. The value will be used as the processors type when the workflow is exported to RGL Expression and it will be used by the RDF Engine as the function or workflow identifier.

The function parameter declared as child node with `param` tag name. A function may have zero or more parameter. There are four attributes that could be added to `param` tag: `name`, `label`, `value` and `type`. Attribute `name` is a mandatory parameter which will be used as the parameter id. Attribute `label` contains the value of the parameter display name and will be used by the property viewer. Attribute `value` is an optional attribute that will contain the default value of the parameter. `Type` is a different attribute compared to the other three. The value of `type` will decide the type of the parameter and how it will be handled and how the field forms will be visualized by the interface. There are seven of parameter types in RDFGears Function Definition Language. The list below shows the possible values of `type` and the parameter's form layout on the interface.

- **String** is a parameter with short string as the values.
• **Text** is a parameter with long text as the values.

Figure 3.7 *Text* parameter interface layout

• **Query** defines a parameter with SPARQL queries as the values. Basically it is the same type with *Text* but with the interface will integrate the form fields with SPARQL Query Editor.

Figure 3.8 *Query* parameter interface layout

• **Fields** defines as parameter with a set of *String* as the values. The field form can be added or removed dynamically during runtime.

Figure 3.9 *Fields* parameter interface layout
• *InputFields* defines a parameter that which the values represent a variable that bound to input ports. The field form is rendered as synchronized dynamic fields between the property viewer which will show the field editor and the graph nodes which will show the input port.

![InputFields parameter interface layout](image)

**Figure 3.10** *InputFields* parameter interface layout

• *List* defines a parameter which has a list of string as its possible values. The parameter is visualized as a list box with predefined option values. The options are defined with option tag and wrapped inside source tag. This option is embedded as the parameter's child element. Listing 3.4 shows an example of function parameter with the type of *List*.

```
<param name="operator" type="List" label="Operator">
  <source>
    <option value="OP_LESS" label="a &lt; b"/>
    <option value="OP_LESS_EQUAL" label="a &lt;= b"/>
    <option value="OP_GREATER" label="a &gt; b"/>
    <option value="OP_GREATER_EQUAL" label="a &gt;= b"/>
    <option value="OP_EQUAL" label="a == b"/>
    <option value="OP_NOT_EQUAL" label="a != b"/>
  </source>
</param>
```

*Listing 3.4 Example of List parameter*

The parameter's interface layout is shown in Figure 3.11 below.

![List parameter interface layout](image)

**Figure 3.11** *List* parameter interface layout
- Parameter with constant value, defined by declare a parameter with has only two attributes, name and value. This type of parameter does not have input form or custom layout. The example use of this parameter is to define a user defined function. A user defined function is a custom defined RDF Gears function which implemented in java and can be identified by its full class path. This path is declared as constant parameter and later it will be used by the engine to invoke the function during the workflow execution. Listing 3.5 line 2 shows the real example of constant parameter to declare a custom user defined function with values nl.tudelft.rdfgears.rgl.function.imreal.CoordinatesToCountry.

```xml
<function type="custom-java">
  <param name="implementation" value="nl.tudelft.rdfgears.rgl.function.imreal.CoordinatesToCountry"/>
</function>
```

Listing 3.5 Constant parameter

Besides the element’s attribute, parameter can have an optional description tag as child element. This element’s value is the description string of the parameter. This description text provides helpful information about the parameter. The example of the parameter description is shown in Listing 3.7 line 10.

**Output** tag declares the processor’s output. The output element will be visualized as the node output port. Output element has a child element which is the type of the data that will be produced by the processor. The discussion about the port type will be presented in Chapter 4 Type System.

```xml
<output>
  <type> Type pattern </type>
</output>
```

Listing 3.6 Output

An example of processors definition using the Function Definition Language is shown in Listing 3.7. The example shows that the function name is the same with the file name which is sparql. The function has two parameters, bindVariables with the type of InputFields and query with type of Query.
Filename: sparql.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<processor label="Local SPARQL" category="Querying">
  <description>
    Write SPARQL Query, bind the variables in the query to the input fields
  </description>
  <inputs>
    <function type="sparql">
      <param name="bindVariables" type="InputFields" label="Variables"/>
      <param name="query" label="Query" type="Query">
        Write SPARQL Query
      </param>
    </function>
  </inputs>
  <output>
    <var name="output"/>
  </output>
</processor>
```

**Listing 3.7 Example of processor definition**

At the end of the workflow graph construction, the end result of the process is an RGL Expression. The transformation of the Function Definition into RGL Expression is a direct transformation. The purpose of this approach is to make the transformation of RGL Expression into visual workflow graph easier. This will makes the process of importing existing RDF Gears workflows become easier. Table 3.1 shows the mapping between the function definition and RGL Expression.

<table>
<thead>
<tr>
<th>Function Definition Language</th>
<th>RGL Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;processor&gt;</code></td>
<td><code>&lt;processor&gt;</code></td>
</tr>
<tr>
<td><code>&lt;function type=&quot;fType&quot;&gt;</code></td>
<td><code>&lt;function type=&quot;fType&quot;&gt;</code></td>
</tr>
<tr>
<td><code>&lt;param name=&quot;parameter_name&quot;</code></td>
<td><code>&lt;config param=&quot;parameter_name&quot;&gt;</code></td>
</tr>
<tr>
<td><code>..&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;data iterate=&quot;false&quot; name=&quot;port_name&quot;&gt;</code></td>
<td><code>&lt;inputPort iterate=&quot;false&quot; name=&quot;port_name&quot;&gt;</code></td>
</tr>
</tbody>
</table>

**Table 3.1 Mapping between RDFGears Function Definition Language and RGL Expression**

### 3.3 Implementation

In previous section, the design of the RDF Gears User Interface has been discussed. In this section we will discuss the implementation of the User Interface. The application is a java web application that can be divided into two main layers, the frontend and the backend. The frontend is a single page web application that uses HTML and JavaScript. The backend is a java application that is used to interact with the data store in the server. The following sub-section will present a detailed discussion of the technical architecture and the core classes of the application.
3.3.1 Technical Architecture

The user interface is a java web application that can be run in various java application servers (e.g. Tomcat, Jetty web server). It is implemented using Google Web Toolkit (GWT) and several supporting libraries. GWT is an open source set of tools from Google that allows us to create a complex JavaScript front-end web application. There are two advantages of using GWT to develop RDF Gears user interface compared to other framework or libraries. First, it allows us to develop the application in Java programming language. This is a big advantage since Java is a more structured programming language compared to JavaScript thus makes the development process easier. Second, GWT compiler automatically compiles and optimizes the application into several web browser specific web applications. It will guarantee the compatibility of the JavaScript code used in the application with the web browsers that used by the user.

![RDFGears User Interface](image)

**Figure 3.12 Relation of RDF Gears UI with GWT and third party libraries**

As shown in Figure 3.12, the application also used third party libraries that needed and used for specific purpose.

- **GWTQuery**, also known as GQuery is a library written in GWT that allows us to write code similar to infamous JavaScript library jQuery. GWTQuery allows GWT to be used in progressive enhancement scenarios where the GWT code itself considered to heavyweight.

- **Gquery-dnd-bundle** is a GWTQuery plugin that consist of GWT classes specifically written to handle drag and drop operation. This library is used to handle all the drag and drop operation in the interface (e.g. graph node drag operation and canvas drag operation).

- **CodeMirror** is a lightweight JavaScript library that can be used to build a code editor in web browser.

As explained before, the user application has two layers, frontend and backend. The frontend is a single page web application that runs independently in client web browser. All the functions that needed by the interface to in processing the workflows are embedded in the client application (e.g. file parser, type checker). However, all the data, the function definitions and workflow source file are located in the server. Client application has to communicate to the backend to retrieve or to store certain workflow or processor file.
The communication model between the frontend application and the backend is shown in Figure 3.13. The application use GWT Remote Procedure Call (GWT RPC) framework in order to transfer the data between frontend and backend. GWT RPC is not the same as web services like SOAP or REST, it is a lightweight method for transferring data between server application (backend) and client application (frontend). It provides service on the backend that can be used to exchange Java object over HTTP.

The workflow and processor source files are stored in the file system as XML files. All the files are contained in a root directory, data. Each workflows and processors has their own file and they are stored in three different directories under data directory. As shown in Figure 3.14, workflow files stored in workflows directory, RGL core functions’ definition files are stored in processors directory and all the user defined function files are store in functions directory. In order to enable the application to know the location of the storage, the full path of the directory has to be defined in the configuration file (WEB_INF\rdf-gears-ui-config.xml).
3.3.2 Core Classes of RDF Gears User Interface

In previous section we have discussed the technical architecture of the application. Now we will discuss the core classes of the RDF Gears User Interface. The source code of the application is written in Java and we will discuss the core classes based on the application source code. To make the discussion easier to understand, we grouped the most important classes based on their functionality and their purpose.

**Workflow graph editor.** The graph editor involves five important classes. The main class that handles all the process is RGCanvas. It orchestrates all other specific-purpose classes. RGCanvas is also responsible for providing the actual canvas to draw the graph where the node and the connector will be drawn. The second class, RGWorkflow, is the class that responsible to hold the workflow information and provides all the functions to manipulate workflow data.

The other classes are the classes that responsible for the workflow graph visualization. The graph visualization is handled by three main classes, Node, NodePort, and Path. Node is an abstract class that provides all the basic methods and event handler of the graph node. This class is extended by three other classes, RWWorkflowInputNode that is responsible for the workflow input node, RWWorkflowOutputNode that is responsible for workflow output node and the generic node class, RGNODE, that is responsible for the other processors node. All the node classes have method `draw()` which is used to render the visual representation of the node on the canvas.

![Figure 3.15 Class diagram of the Workflow graph editor core classes](image)

NodePort is the class that visualize and provides handlers for node input port and output port. It is used by the node classes to hold the information of the port’s type and the connections. The port type can be retrieved using `getType()` that will return an object that hold the port type. The visualization of the connection between ports is handled by the...
Path class. It is used to draw the Bezier line that connects the ports. Path class uses PathRenderer class to actually draw the line in the browser by using HTML5 Canvas [23] or in SVG [27] as the alternative. The class diagram of the workflow graph editor classes is shown in Figure 3.15. The class diagram only shows the most important classes and its most important methods.

**Node property editor**, as shown in the class diagram in Figure 3.16, the property editor uses two main classes, RGPropertyPanel and RGFunctionParam. RGPropertyPanel is responsible for providing the container of the nodes properties which includes the function parameters. The workflow node use RGFunctionParam class to hold the parameter value and information, one instance for each parameter. The node’s property interface (fields form) is laid out by calling display() method that belongs to the RGFunctionParam and pass the container object as the parameter.

![Figure 3.16 Class diagram of Node property editor core classes](image)

**Workflow list and navigation.** Workflow list panel is handled by RGNavigationPanel. It is responsible to provide the layout and the container to display the workflow list. The workflow and processors list itself is retrieved from server in form of XML file. The file is parsed by RGWorkflowsPanel class and display the list in a more user friendly list. The other navigation functionality is the editor tab which is handled by the RGTabBar class and the application menu which is handled by the RGMenu class.

**Type system.** RDF Gears type system is handled by three main classes, RTGType, RTGTypeUtils and RTGTypeChecker. RTGType is the class that parses and holds the port type information. Basically is it a class with properties and method that represent the XML
format of the port type. RGTypeUtils is the class that provide generic methods that used in the type system (e.g. cleanWhiteSpaces(), getSimpleVarType(), getSimpleBoolType(), getSimpleIntType()). RGTypeChecker is the main class that does the type checking operation. This class will apply all the type checking rules (see Chapter 4) to validate whether the workflows is well typed or not and to produce the final type of the workflow. The class diagram of the type system is shown in Figure 3.17.

![Class diagram of type system core classes](image)

**Communication and storage.** GWT RPC the communication mechanism between frontend and the backend is implemented by two interfaces and one implementation class. They are located in two different packages, client and server. RGService and RGServiceAsync is the GWT RPC service interface that act as proxy so the frontend can call the service on the server. The Service is implemented in RGServiceImpl class. This class uses DataDriver class to interact with the file system to retrieve or to store workflow or processor files. This approach is used to create a modular architecture. It will make the transition from file system storage to another kind of storage (e.g. database, cloud storage) become easier by only modifying the DataDriver class.

### 3.4 Conclusion

In this chapter we have shown that the current RDF Gears User interface has few problems. These problems become our starting point, how to design and create a new RDF Gears User Interface that works without any errors and to make it scalable. The problems that we identified are become our requirements. Among the requirements that have been introduced, clean and organized UI, scalability, nested workflow support and type system are the most important. Property editor is introduced to make the user interface cleaner.
and organized. RDFGears Function Definition Language is introduced that offer a new approach in defining workflow graph node. It enables us to extend RDF Gears core functions or users defined functions without reprogram and recompile the user interface. The process of editing nested workflow is supported by enabling the editor to open and edit multiple workflows.

In the last part of this chapter, we show how the RDF Gears User Interface is implemented. The application is built with Google Web Toolkit. The application has two layers, the client application (frontend) and the server application (backend). GWT RPC framework is used to provide an efficient and lightweight data transfer mechanism between the frontend and the backend. In the last sub-chapter, set of core classes of the application has been introduced. The core classes are divided into five categories based on their characteristic and functionality; classes for workflow graph editor, node property editor, workflow list and navigation, type system and classes for communication and storage.
4 Type System

In the previous chapter, we have discussed that one of the main features of the RDF Gears User interface is the integration of the type system with the user interface. In the beginning of this thesis, we also have discussed the RDF Gears in general that includes the data model and the type system of RDF Gears Language. In this chapter we will discuss how the RDF Gears type system was designed and implemented in the user interface. First, section 4.1 present the overview of RDF Gears types. This shows how we formulate the types based on the existing RDF Gears types. In section 4.2 we discuss how the type’s inference system works, it includes the rules and the algorithm that used to do the type checking. The algorithm and the rules are formulated based on the result of a personal discussion with my thesis supervisor. In section 4.3 we discuss how we declare the type. In this section new present a set of XML tags that we use to declare the RDFGears types. Section 4.4 discusses the mechanism of the live type checking and the last section concludes the discussion.

4.1 RDF Gears Types Overview

The type system that implemented in the user interface is not entirely a new type system. It is based on the RGL types system thus the main purpose of this discussion is not about defining a new type system for RDF Gears but how the type system is designed and implemented in the user interface. As explained in section 2.5, RGL use the static type system that adopted the Named Nested Relational Calculus (NNRC) and the RDF Data model. Based on the RGL types, the set of Types \( T \) in RDF Gears defined as follows:

- Basic types \( B \) that contains two types, \( \text{Bool} \) and \( \text{Int} \)
- Tuple type which contains a set of distinct \( a_1, ..., a_n \in F \) field names and \( T_1, ..., T_n \in T \) field types in form of \( \langle a_1:T_1, ..., a_n:T_n \rangle \)
- Extendible tuple type which is a tuple type with extendible field size denoted as \( \langle a_1:T_1, ..., a_n:T_n \rangle^+ \)
- Bag type which is a finite collection of types denoted as \( \{T\} \)

We also define a set of type variables \( V \). Variable is a special type that can contain or refer to any other types \( \tau \) where \( \tau \in T \). For the rest of the discussion we will use the short hands \( \langle a_i: \tau_i \rangle_{i=1}^{n} \) to denote tuple type \( \langle a_1: \tau_1, ..., a_n: \tau_n \rangle \) and \( \langle a_i: \tau_i \rangle_{i=1}^{+} \) to denote extendible tuple type \( \langle a_1: \tau_1, ..., a_n: \tau_n \rangle^+ \). The disjoint tuple union of two fixed tuple \( \tau_1 \) and \( \tau_2 \) is denoted as \( \tau_1 \oplus \tau_2 \). We define such that:

\[
\langle a_1: \delta_1, ..., a_n: \delta_n \rangle \oplus \langle b_1: y_1, ..., b_m: y_m \rangle = \langle a_1: \delta_1, ..., a_n: \delta_n, b_1: y_1, ..., b_m: y_m \rangle \quad \text{where} \quad \{a_1, ..., a_n\} \cap
\]

\[
\{b_1, ..., b_m\} = \emptyset
\]
\{b_1, ..., b_m\} = \emptyset. The same notation is used to denote the disjoint tuple union of two extended tuple.

### 4.2 Type checking algorithm and rules

This section will specify the mechanism to determine whether a workflow is well-typed or not. In this discussion we will see the workflow in a more applicative way, as a collection of processors with input ports and output port and connection that represent the data flow.

For every workflow, we only interested in the set of processor’s ports \(P\) and the set of connection \(C\) that connect processor’s ports and represent flow of data between them. Every port \(p \in P\) has type \(\tau \in T\) that represent the structure of the data the processor will accept as input if \(p\) is an input port, or the structure of the output data if \(p\) is an output port. Each port connection \(c \in C\) connects an output port to an input port. An output port can be connected to multiple input ports but not the other way around, an input port can only be connected to a single output port. A connection between ports also means that the port’s type have to be matched or should be combined into single type. For example, an output port \(p_1\) with tuple type \(\tau_1\) connected to an input port \(p_2\) with type variable \(\tau_2\) and based on the behavior of type variable, the type of \(p_2\) will be substituted by the type of \(p_1\) which is a tuple type \(\tau_1\) denoted by \(\tau_2 \rightarrow \tau_1\). This substitution will be propagated through other connections since the variable \(\tau_2\) is may also used in other ports. In the case of non-type variables (Basic types, tuple, and bag), type unification will be done which will end with extended substitution table or failed in case of un-matched types.

So, basically the process of determining the whether a certain workflow is well-typed or not is the process of building substitution table and type unification. A workflow is well-typed if a valid substitution table can be built.

#### 4.2.1 Type renaming

Type can be declared as a variable or contain variables. The scope of a variable during the declaration is within a processor means that variables with the same name in a processor refer to the same variable but not in different processors. During the type checking the scope of a type variable is global within one workflow. The substitution of a variable \(v_1\) in a processor also applied to all other type variable \(v_n \in V\) that exist in the workflow where \(name of v_1 = name of v_n\).

In chapter 3 we have discussed that each processor has its own file written in Function Definition Language and the declaration of the port’s type are embedded in the file. This is done without considering that other processors may have port’s type that contain variable with the same name. Type renaming is the process of assigning a unique name for all type variables to ensure there are no two or more variables with the same name in different processors before the process of building substitution table and type unification is performed, i.e. suppose there is \(v_1 \in V\) in processor \(X\), type renaming is a process to ensure there is no \(v_2 \in V\) in processor \(Y\) where \(X \neq Y\) and the name of \(v_2\) is equal to the name of \(v_1\).
4.2.2 Type Substitution Table

Substitution table $S$ is the table that contains all the type variable names substitution. Substitution mapping function $M$ is used to get the mapping of a certain type variable name based on the substitution table content. For example, if $S$ contains entry that maps variable $v$ to type $\tau$ the application of $M$ on $v$ will return $\tau$ denoted as $M(v) \Rightarrow \tau$. If $S$ does not contains any mapping for $v$, the application of $M$ will be undefined on $v$ (i.e. return $null$) denoted as $M(v) \Rightarrow \bot$. Suppose we have an application of $M$ on $v$ that return type $\tau$, the deep application of the substitution mapping function $M^*$ on a variable $v$ that will return type $\sigma$ denoted as $M^*(v) \Rightarrow \sigma$, is the application of $M$ to all variables $v_{1..n}$ that exist in $\tau$ so there cannot be any occurrence of $v_i, i = 1..n$ where $M(v_i)$ is defined (i.e. $S$ contains an entry that maps $v_i$ to a certain type).

Furthermore, the deep application of mapping function $M^*$ on variable $v$ based on the content of $S$ is defined by the following rules.

Variables:

\[
\begin{align*}
M(v) & \Rightarrow \bot & M(v) & \Rightarrow \tau & M^*(v) & \Rightarrow \sigma & \sigma \in V \\
M^*(v) & \Rightarrow v & & & & & \\
\end{align*}
\]

Basic Types:

\[
M(v) \Rightarrow \tau & \quad \tau \in B \\
M^*(v) & \Rightarrow B
\]

Fixed and extendible tuple types:

\[
M(v) \Rightarrow \langle a_i; \tau_i \rangle_{i=1..n} & \quad \{M^*(\tau_i) \Rightarrow \sigma_i\}_{i=1..n} \\
M^*(v) & \Rightarrow \langle a_i; \sigma_i \rangle_{i=1..n}
\]

\[
M(v) \Rightarrow \langle a_i; \tau_i \rangle^+_{i=1..n} & \quad \{M^*(\tau_i) \Rightarrow \sigma_i\}_{i=1..n} \\
M^*(v) & \Rightarrow \langle a_i; \sigma_i \rangle^+_{i=1..n}
\]

Bag or collection of types:

\[
M(v) \Rightarrow \{\tau\} & \quad M^*(\tau) \Rightarrow \sigma \\
M^*(v) & \Rightarrow \{\sigma\}
\]

At this point, maybe it is confusing because the rules only apply $M^*$ on type variable. As mentioned before, this discussion will explain the type system from the application point of view, how the actual algorithm works. In this perspective, we have input and output ports that have type $\tau \in T$. To make the application of the type system easier and simpler, we
decompose all the port types. So, for all port types \( \tau_1 \ldots \tau_n \) where \( \tau_i \in V, i = 1 \ldots n \), an entry of \( v_i \rightarrow \tau_i \) will be added to substitution table and the type of the port \( p_i \) will be substituted by type variable \( v_i \), therefore \( M' \) can be applied since all the ports types are not type variable.

The application of \( M' \) is to get the final type of each port. The final type of each port can be inferred after the final substitution table is built. Substitution table \( S \) is extended by iterating all the connected input ports in the workflow. So, for each input ports \( p_i, i = 1 \ldots n \) where there is an output port \( p' \) that connected to \( p_i \) we get the type of \( p_i \) as \( \tau_1 \) and the type of \( p' \) as \( \tau_2 \). Because of the type decomposition, both \( \tau_1 \) and \( \tau_2 \) are type variables. The next step is applying \( M \) on them and unifies the returned types. Two types are match if the types can be unified and mismatch otherwise. The iteration will unify the types which will end with the final substitution table or failed due to type mismatch.

### 4.2.3 Type Unification

Type unification is a procedure that attempts to unify two types that ends with extended substitution table with entries that describe the implied substitutions if the unification succeeded. In the previous section we have discussed that each step of unification will involve two port types \( \tau_1 \) and \( \tau_2 \). In this procedure we define \( M' \) as the repetitive application of \( M \) on \( \tau_1 \) and \( \tau_2 \), which means that \( M \) will repeatedly applied on the returned value until it return non type variable or the substitution of the variable is undefined which will return the last type variable on the substitution chain.

In this algorithm we define \( E \) as a set of equations in form of \( \sigma_1 \equiv \sigma_2 \) where \( \sigma_1 \neq \sigma_2 \). The elements of \( E \) defines the equalities of types that should hold (match) because of the composition of the workflow. We also have \( \tau_1 \) and \( \tau_2 \) which are the types that we want to unify, the substitution table \( S \), and the repetitive application is of \( M \).

We let \( (S,E)_{[\tau_1 \rightarrow \tau_2]} \) denote the state that we obtain if we (1) remove the substitution for \( \tau_1 \) from \( S \), (2) add a substitution entry of \( \tau_1 \rightarrow \tau_2 \) to \( S \). Basically it means that \( \tau_1 \) and \( \tau_2 \) substituted by the same type which is the result of the unification of \( \tau_1 \) and \( \tau_2 \). This algorithm will allow the substitution chain between type variables and that is why we need \( M' \). The reason why we do not simply replace \( \tau_1 \) with \( \tau_2 \) is because in the application level, the processor port still hold \( \tau_1 \) as its type, all the changes only occurred within \( S \) and \( E \).

As explained in the previous section, the final substitution table – which is the main goal of the type checking – is built by iterating all the connected input ports and unifies the types (type of input and output port). Now we will discuss how the final substitution table is built. We will denote a single unification step as \( (S,E) \leftrightarrow (S',E') \), defined by the following rules.

**Note:** We have to keep in mind that \( \tau_1 \) and \( \tau_2 \) are the ports type which is type variable after the decomposition.
Variables:

\[ M(\tau_1) \Rightarrow \bot \]
\[ (S, E \cup \{\tau_1 \equiv \tau_2\}) \Rightarrow (S, E)_{\tau_1 \rightarrow \tau_2} \]

Basic types:

\[ M(\tau_1) \Rightarrow \sigma \quad M(\tau_2) \Rightarrow \sigma \quad \sigma \in B \]
\[ (S, E \cup \{\tau_1 \equiv \tau_2\}) \Rightarrow (S, E)_{\tau_1 \rightarrow \tau_2} \]

Tuple

\[ M(\tau_1) \Rightarrow \langle a_i; \sigma_i \rangle_{i=1 \ldots n} \quad M(\tau_2) \Rightarrow \langle a_i; \delta_i \rangle_{i=1 \ldots n} \]
\[ E' = E \cup \{\sigma_i \equiv \delta_i | 1 \leq i \leq n\} \]
\[ (S, E \cup \{\tau_1 \equiv \tau_2\}) \Rightarrow (S, E')_{\tau_1 \rightarrow \tau_2} \]

\[ M(\tau_1) \Rightarrow \langle a_i; \sigma_i \rangle^+_i \quad M(\tau_2) \Rightarrow \langle b_j; \delta_j \rangle_{j=1 \ldots m} \]
\[ E' = E \cup \{\sigma_i \equiv \delta_i | 1 \leq i \leq n\} \]
\[ (S, E \cup \{\tau_1 \equiv \tau_2\}) \Rightarrow (S, E')_{\tau_1 \rightarrow \tau_2} \]

Bag or collection of types:

\[ M(\tau_1) \Rightarrow \langle a_i; \gamma_i \rangle_{i=1 \ldots n} \oplus \langle b_j; \sigma_j \rangle^+_j \]
\[ M(\tau_2) \Rightarrow \langle b_j; \delta_j \rangle_{j=1 \ldots m} \oplus \langle c_k; \varepsilon_k \rangle^+_k \]
\[ \{a_1, \ldots, a_n\} \cap \{c_1, \ldots, c_k\} = \emptyset \]
\[ S' = S[t_2 \rightarrow \langle a_i; \gamma_i \rangle_{i=1 \ldots n} \oplus \langle b_j; \sigma_j \rangle^+_j \oplus \langle c_k; \varepsilon_k \rangle^+_k ] \]
\[ E' = E \cup \{\sigma_i \equiv \delta_i | 1 \leq i \leq m\} \]
\[ (S, E \cup \{\tau_1 \equiv \tau_2\}) \Rightarrow (S', E')_{\tau_1 \rightarrow \tau_2} \]

After applying this unification step to all possible unifications in the workflow, we have temporary extended \( S \) and \( E \) that contains all the unifications that still need to be made. The content of \( E \) is reduced by applying one of the unification rules. The unification may fail if there is no rule that can be applied (i.e. mismatched types). The algorithm continues until no more reductions are possible. The unification succeeds if \( E \) is empty thus a valid final substitution table is produce otherwise we conclude that the unification cannot be done (i.e.
workflow is ill-typed). If the unification is succeeds then we can retrieve the final type of each ports in the workflow by applying $M^*(\tau)$ where $\tau$ is the type of the port.

### 4.2.4 Handling Recursive type

Recursive type is a special case in the RDF Gears type system that has to be avoided. The substitution algorithm is designed with assumption that the substitution table does not contain any recursive mapping therefore it has to be handled separately. The consequence of having recursive mapping is the infinite substitution when the deep substitution function applied on certain variable. The case of recursive type substitution is possible when it involves tuple and bag type because the structure of tuple and bag type allows type to be nested.

To make it clearer, let we give an example. Suppose we have processor $A$ with an output port $pA_1$ and type variable $\gamma$. Then we have processor $B$ with two input ports $pB_1$ with type tuple $t = \langle a: \beta \rangle$ and $pB_2$ with type variable $\beta$. Suppose the workflow is constructed with these two processors in a way that $pA_1$ is connected to both $pB_1$ and $pB_2$, the substitution table will have a recursive mapping. Based on the substitution and the unification algorithm that we presented, the content of substitution table will be $[\gamma \rightarrow \langle a: \beta \rangle, \beta \rightarrow \langle a: \beta \rangle]$ with an assumption that the sequence of the type unification is $(\gamma, \langle a: \beta \rangle)$ followed by $(\gamma, \beta)$. We can clearly see that the mapping is recursive.

To prevent this from happening, we validate all the potential substitution before we extend the substitution table with the entry. In the given example, in the process of unifying $(\gamma, \beta)$, we can found the potential mapping which is $\beta \rightarrow M(\gamma) = \beta \rightarrow \langle a: \beta \rangle$. Before we extend the substitution table with this entry, we check whether the mapping of $\beta$ to the deep substitution of $\gamma$ is recursive. The checking is straightforward by iterating all the type variable in the result of $M^*(\gamma)$ whether it has $\beta$ which is recursive or not. The reason that we need to implement the deep substitution is because it will substitute all the nested type and return the temporary final type.

### 4.3 RDF Gears Types Declaration

In the previous sections we have discussed the RDF Gears types and the algorithms that used in the type system. Now we will discuss how to declare or to define the type of a certain port. We use XML that is embedded in the function definition to represent the type. For each of the types, we define an XML tag which later will be parsed by the type system. Table 4.1 shows the XML representation of each type.
Extendible tuple \( \langle a: \beta, b: \langle b1: \text{bool}, b2: \text{int} \rangle \rangle^* \)

Bag \( \{\{\beta}\} \)

<table>
<thead>
<tr>
<th>Table 4.1 Type declaration in XML</th>
</tr>
</thead>
</table>

As show in the example of function definition in Listing 3.7, the type is wrapped in a type tag \( \langle \text{type} \ldots \rangle \). Then type element is added as child element to certain port to declare the port type (Note: the port definition is explained in section 3.2.2).

The type of RDF Gears workflow file or RGL expression is defined with the same approach except for the workflow output port type. The input port of a workflow is defined explicitly with tag \( \langle \text{workflowInputPort name="input_port_name"} \rangle \). The definition of the workflow input port type is straightforward by embedding the type as the child element. The case is different with workflow output port. The workflow output port is not defined explicitly. To define the workflow output port type, we wrap the type output-type tag \( \langle \text{output-type} \ldots \rangle \) and add the element as the child node of network element. Listing 4.1 shows the example of workflow input port type in line 15-17 and the definition of workflow output type in line 21-23.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rdfgears>
  <metadata>
    <id>aMashup</id>
    <name>A Mashup</name>
    <description>a mashup that consumes linked data, and follows and merges all owl:sameAs links</description>
    <category>Mashup</category>
    <password/>
  </metadata>
  <workflow>
    <workflowInputList x="300" y="100">
      <workflowInputPort name="input0">
        <type>
          <var name="a68"/>
        </type>
      </workflowInputPort>
    </workflowInputList>
  </workflow>
</rdfgears>
```
### 4.4 Live Type Checking Mechanism

In section 3.3.2 we can find the discussion regarding to the implementation of the type system, the architecture and the core classes. In this section we will present how the live type checking mechanism is designed and implemented. The main purpose of implementing type checking in the user interface is to allow RDF Gears to evaluate the workflow type before it is executed by the RDF Gears engine. This approach also allows us to do the type checking as the workflow is being constructed. The application will give a live feedback if there is a type error and the users can respond to it immediately.

![Live Type Checking Flow Chart](image)

**Figure 4.1 Live Type Checking Flow Chart**

As we can see in Figure 4.1, the live type checking process is started by a set of events as a trigger to the type system to do the type checking. The events are:

- Open an existing workflow
- Connecting processor’s ports
- Delete ports’ connection
- Delete certain processor from workflow graph
• Changing the value of port iterate state

These are the events done by user while editing or opening a workflow. The type checking will be performed on the active workflow whenever one of these events occurred.

4.5 Conclusion

In this chapter we have discussed the type system that is implemented in the RDF Gears User Interface. If we see the type system as one big process, it can be divided into four phases. The first phase is type declaration. Port types are declared in XML and embedded in the Function Definition. We have shown the mapping between RDF Gears types and the XML tags that represent the types. The second phase is the pre-type checking process which includes type renaming and type decomposition. This purpose of this process basically is to normalize the declaration of the types into a standard form that will be easier to be processed by the type system. The third phase is type checking itself which is the type unifications and the process of building the substitution table. The last phase is the outcome of the previous phase which has two possibilities, if the unification succeeded the final type can be computed with the deep substitution function or the application will show the error if the unification failed.
Chapter 3 describes the design and the implementation of the RDF Gears Graphical User Interface. In Chapter 4 we have discussed the RDF Gears type system that is integrated with the user interface. In this chapter we present our evaluation. The main purpose of the evaluation is to evaluate the correctness of our implementation and it does not include any survey or feedback from users. First we will do a comparison between our application and the existing RDF Gears UI to see the improvements that we have made and the correctness of our implementation. Then we will evaluate the usability of the main features which includes the RDFGears Function Definition Language and the type checking system. At the end of this chapter, we present the conclusion of our discussion.

5.1 Comparison with existing RDF Gears User Interface

In chapter 4 we have discussed the problems with the existing RDF Gears User Interface and how we define our requirements by using those problems as our starting point. We have discussed the reason behind all the features that we have implemented. In this evaluation we will show how our application’s designs and implementation are different from the existing one. We divided the comparison in three parts which basically are the phases of working with workflow. The phases are the construction, debugging, and the execution of the workflow.

5.1.1 Constructing RDF Gears Workflow

As we have explained before, one of the biggest improvements of our application is the ability to open and edit multiple workflows at one editor at the same time. The old approach with the current RDF Gears UI is to open another instance of the application in separate browser and edit the workflow. By this approach we can still edit multiple workflows but obviously it is inconvenient. Moreover our application will detect if the edited workflow is nested in other opened workflow and it will give notifications if the save operation occurred. Figure 5.1 shows the appearance of the editor in editing a workflow. The top most figure is the existing RDF Gars UI and our application is at the bottom.
The Figure shows both the application opening *aMashup* workflow. A workflow that has two other workflow nested as processors, *graph_closure* and *fetchSameAs*. As we can see in the figure, our application can open all three workflows in editor tabs. The tabs make it easier to switch between one workflow to another. Besides the tabs, there are also compare other functionalities and features that used in constructing the workflow graph. The result is summarized in table 5.1.
<table>
<thead>
<tr>
<th>Features/Functionalities</th>
<th>Existing Application</th>
<th>RDF Gears User Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add workflow nodes</td>
<td>Drag and drop fashion</td>
<td>Drag and drop fashion</td>
</tr>
<tr>
<td>Editing nested workflow</td>
<td>Not supported</td>
<td>Supported with tabs</td>
</tr>
<tr>
<td>Edit multiple workflow</td>
<td>Not Supported</td>
<td>Supported by open multiple workflow in tabs</td>
</tr>
<tr>
<td>Node's properties</td>
<td>Embedded in workflow nodes</td>
<td>Displayed and edited with nodes properties editor</td>
</tr>
<tr>
<td>Query Editor</td>
<td>Standard text area form</td>
<td>SPARQL Query Editor</td>
</tr>
<tr>
<td>Add nested workflow or user</td>
<td>With Function Node</td>
<td>Each workflow and user function can be added as a node that has it owns function definition file</td>
</tr>
<tr>
<td>functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cyclic workflow graph</td>
<td>Not supported</td>
<td>The editor prevent user from creating cyclic graph</td>
</tr>
<tr>
<td>detection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Workflow List</td>
<td>Uncategorized list</td>
<td>Categorized workflow list and user defined functions</td>
</tr>
<tr>
<td>Web browser support</td>
<td>Mozilla Firefox</td>
<td>The latest version of all major web browsers (Has been tested with Internet Explorer 9, Mozilla Firefox 15.1, Google Chrome 21, Opera 12.02)</td>
</tr>
</tbody>
</table>

Table 5.1 Summary of improvements and new features

5.1.2 Debugging Workflow

Debugging a workflow in our application is easier and more convenient. The debugging in this case means that we can see all the properties of the workflow to find the problem in case of the execution of the workflow is failed. The current RDF Gears UI only provides functionality to export the workflow graph into XML which is not informative enough. Our application provides tools that make the debugging easier. We can inspect the input and output port types of each through type viewer in debug menu or in property editor which will help to solve the type error or to understand the dataflow between processors.

The application also provides tool to view the original source of the workflow and by original means the last saved version. Suppose that we open and modify a workflow, we can compare the original source of the workflow and the current one to analyze the different in of its RGL expression. The node id which is hidden by default also can be shown with hide/who node id menu item from the debug menu.

5.1.3 Executing Workflow

Compared to the existing application, our application uses the same approach to execute or to run a certain workflow. First, the workflow has to be saved and store in server. The application will invoke the rdfgears-rest application that will run the RDF Gears engine and execute the workflow. We execute the workflow by using Run menu that can be found
on each entry of the workflow list. In our test case, we successfully execute the \textit{aMashup} workflow and produce the expected result.

\section*{5.2 Usability of the Features}

In this section we will present the evaluation of the newly introduced features. Basically most of the features that introduced are new since we build the application from scratch and with a new design, but in this section we will focus on two big features the Function Definition Language and the Type Checking. We will evaluate the usability, whether they are really useful and functioned as expected.

\subsection*{5.2.1 Flexible RDFGears Function Definition}

The goal of the Function Definition Language is to enable the users to define and modify the processor's graph node definition in a flexible way. Our test case, \textit{aMashup}, has proven the usability of this feature. All the processors definition and the user defined function that used in the workflow were defined purely written in RDFGears Function Definition Language which is expressed in XML. In addition, all the existing core functions of the RDFGears engine have been defined with Function Definition Language. It was done without any configuration or any changes to the application’s code. The application automatically transforms the function definition into visual workflow node and provides the layout of the function’s parameter input form on the property editor. The implementation has proven to be correct. With this improvement, theoretically, we are able to define as much processors or user defined function as we want. This is a big improvement for RDF Gears workbench because now we can extend the engine to support more operators or user defined functions without worried about the user interface. It proves that the RDFGears Function definition language have improved the scalability of RDFGears.

\subsection*{5.2.2 Type Checking}

Even though type checking is one of the main features, it is still an experiment, a prove-of-concept to show that the type system can work on the user interface. Based on our evaluation we prove that the type system is functioned as expected. As shown in Figure 5.2 we purposely create an error by connecting nodes with mismatch types (i.e. tuple to bag) and the type system can detect and show the error.
The type checker provides the log to show type error information. The workflow graph also visualizes the error by drawing the connection with different color to show the location of the error.

Even though the type system works as we expected, it still has limitation. The type system only support the statically define types which means it doesn't support he dynamic type. What we mean by dynamic type is the type that generated during runtime. For example tuple union, when we want to create a processor that combines the fields of two tuple. It is impossible to statically define the generic form of the type of this processor with our type system as the field of the tuple is known during the runtime. This is considered as future work and discussed in the last chapter of this thesis.

5.3 Conclusion

In this chapter we have compared the features and the functionality of the existing RDF Gears UI and our application. We have shown that our application has more advanced features that can be used to work with RDF Gears workflow. The new design of the interface design (i.e. introducing property editor, nested workflow editor) is shown to be functioned and also improve the usability of the editor in general.

We have also presents the evaluation of two new main features that we introduced in our application. The RDFGears Function Definition Language that can be used to create function definition as expected. It shows the improvement as it makes it easier to extend the RDF Gears Engine without worrying about reprogramming the UI. While evaluating the type system, we found that it is still need to be improved to fully support the RDF Gears type system. The discussion about the improvement that still needs to be made is presented in the next chapter.
6 Conclusion and Future Work

This chapter discusses the conclusions regarding to our research goal. First we discuss whether we have achieved our research goal or not. We will describe the achievements that we have made by answering the research questions based our result. Then we describe the contribution that we have made in this thesis. Finally, we conclude the chapter by giving an overview of identified future work.

6.1 Conclusions

This section reviews our research goal, and determines whether we have achieved our goal or not. The research goal of this thesis project was defined as:

“Design and develop an interactive, dynamic and functional web-based Graphical User Interface for RDFGears, an RDF transformation and integration workbench.”

In order to achieve this goal, first we needed to understand RDFGears. We found that RDFGears has three main components: RDF Gears Language (RGL), the engine, and the user interface. From this fact, we have a basic knowledge that the implementation of the user interface and the engine is separated thus we can design and implement our application without directly interact with the RDF Gears Engine.

In Chapter 3 we present the detail explanation of our design and implementation of the user interface. Our application is a single page web application built with Google Web Toolkit. It has two layers, the frontend which is the web application that will be used by user to build RDF Gears workflow and the backend that runs on server that will manage all the files that stored in the file system. The application is implemented separately from RDF Gears engine. We introduced RDFGears Function Definition Language, a communication protocol between the engine and the User Interface. This protocol enable us to extend the engine (i.e. add operators, add user defined functions) without worried about the user interface. This is also answering the question in our research question that indeed it is possible to implement the GUI without altering the RDF Gears engine. Furthermore it improves the RDFGears as it term of extensibility.

Besides the RDFGears Function Definition Language, the application offers several features that we believe will improve the RDF Gears. The complete features are explained in chapter 3. The most important features includes, support for editing nested workflow, node property editor and type system. The editor implements tabbed editor to edit multiple workflows. The synchronization between tabs is added to enhance the ability to edit nested workflows.

The main purpose of the node property editor is to avoid the interface from having cluttered layout. We implement a separate panel to display all the node properties thus we have a minimal display of the workflow graph nodes. The type system is implemented by
adopting the RGL type system. In chapter 5 we describe the algorithms and the rules that we use in the type system. We define a set of XML syntax to define port types which is embedded in the function definition. The implementation of the type system answered our research question that even though there are some improvements that need to be made but it is possible to implement the type system of the RDF Gears on the user interface.

In chapter 5 we presented the evaluation of our application to evaluate the correctness of our implementation and it shows that our implementation has proven to be correct and our application is functioned as expected. In the evaluation we successfully build a complex workflow with our application. We also have discussed the usability of our application. Base on the result we conclude that we have achieved our research goal.

6.2 Future Work

The result of this this thesis project is hoped to be a contribution to the development of RDF Gears workbench. However there are still some features that need to be implemented or need to be improved.

**Type System** There are two improvements that can be made in the typing system. First, as explained in chapter 5 the type system still have limitations. The type system only supports static declaration of the type. Meanwhile, there are some conditions that need dynamic type construction. In the evaluation, we gave an example of tuple union. A generic type for tuple union cannot be declared. The second improvement that can be made is a dynamic substitution table. Our current approach is to rebuild the substitution table and restart all the operations every time the type checking operation occurred. Even though the application has a good performance with current approach but obviously it is not the best approach in term of performance. The idea for future work is to build the substitution table dynamically. The type system only updates the type mapping entry that related to the current operation (i.e. adding/removing a connection).

**Nested graph Visualization** Our current approach in dealing with nested workflow is by editing the nested workflow in a new editor tab. However the visualization of nested graph itself still hasn't implemented. The feature that can be implemented in the future is to enable the editor to open the nested graph in the same tab by drawing the nested workflow as one big nested graph. A real example of this approach is Taverna workflow editor. Even though the nested graph is open is visualized in read only mode but it will help the user to understand the whole workflow. This will be a good improvement for RDF Gears User Interface.

**User Management** Even though the application can be used by multi users, but the current application hasn’t support this feature naturally. Currently, multi user mode is only available with manual coordination between users to avoid concurrent modification of certain workflow. The workflow restriction is also an issue where certain workflow should be able to be protected. The improvement that could be made is to implement a user
management system in the application. Each user has an account. This account will be linked to workflows they have created. It will make the process of sharing workflow easier. The user system can also linked with a scientific community social network such as myExperiment to enable user share the workflow globally.
References


Appendix A: XML Schema for RDFGears Function Definition Language

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">
  <xs:element name="processor">
    <xs:complexType>
      <xs:all>
        <xs:element name="description" type="xs:string" minOccurs="0"/>
        <xs:element name="inputs">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="function">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="param" minOccurs="0" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:all>
                          <xs:element name="description" minOccurs="0"/>
                        </xs:all>
                    </xs:complexType>
                  </xs:sequence>
                  <xs:attribute name="name" type="xs:string" use="required"/>
                  <xs:attribute name="label" type="xs:string" use="required"/>
                  <xs:attribute name="type">
                    <xs:simpleType>
                      <xs:restriction base="xs:string">
                        <xs:enumeration value="String"/>
                        <xs:enumeration value="Fields"/>
                        <xs:enumeration value="InputFields"/>
                        <xs:enumeration value="List"/>
                        <xs:enumeration value="Text"/>
                        <xs:enumeration value="Query"/>
                      </xs:restriction>
                    </xs:simpleType>
                  </xs:attribute>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="label" use="required"/>
            <xs:attribute name="iterate" use="required">
              <xs:simpleType>
                <xs:restriction base="xs:string">
                  <xs:enumeration value="true"/>
                  <xs:enumeration value="false"/>
                </xs:restriction>
              </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="data" minOccurs="0" maxOccurs="unbounded">
              <xs:complexType>
                <xs:all>
                  <xs:element name="type" type="rdfgears_type"/>
                </xs:all>
              </xs:complexType>
            </xs:attribute>
          </xs:complexType>
        </xs:element>
      </xs:all>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
<xs:schema>
  
  <xs:complexType>
    <xs:sequence>
      <xs:element name="output">
        <xs:complexType>
          <xs:all>
            <xs:element name="type" type="rdfgears_type"/>
          </xs:all>
        </xs:complexType>
      </xs:element>
      <xs:element name="label" type="xs:string" use="required"/>
      <xs:element name="category" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:complexType>

  <xs:complexType name="tuple_field">
    <xs:all>
      <xs:element name="field" minOccurs="0">
        <xs:complexType>
          <xs:complexContent>
            <xs:extension base="rdfgears_type">
              <xs:attribute name="name" use="required"/>
            </xs:extension>
          </xs:complexContent>
        </xs:complexType>
      </xs:element>
    </xs:all>

  </xs:complexType>

  <xs:complexType name="rdfgears_type">
    <xs:choice>
      <xs:element name="var">
        <xs:complexType>
          <xs:attribute name="name" use="required"/>
        </xs:complexType>
      </xs:element>
      <xs:element name="int"/>
      <xs:element name="bool"/>
      <xs:element name="tuple" type="tuple_field"/>
      <xs:element name="etuple" type="tuple_field"/>
      <xs:element name="bag" type="rdfgears_type"/>
    </xs:choice>
  </xs:complexType>

</xs:schema>
Appendix B: Example of RDFGears Workflow source that produced by the application

```xml
<!-- filename:aMashup.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<rdfgears>
  <metadata>
    <id>aMashup</id>
    <name>A Mashup</name>
    <description>a mashup that consumes linked data, and follows and merges all owl:sameAs links</description>
    <category>Mashup</category>
    <password/>
  </metadata>
  <workflow>
    <workflowInputList x="300" y="100">
      <workflowInputPort name="input0">
        <type>
          <var name="a68"/>
        </type>
      </workflowInputPort>
    </workflowInputList>
    <network output="node_20" x="919" y="304">
      <output-type>
        <var name="a71"/>
      </output-type>
      <processor id="node_18" x="429" y="217">
        <function type="custom-java">
          <config param="implementation">workflow:fetchSameAs</config>
        </function>
        <inputPort iterate="false" name="uri">
          <source workflowInputPort="input0"/>
        </inputPort>
      </processor>
      <processor id="node_19" x="606" y="80">
        <function type="custom-java">
          <config param="implementation">workflow:graph_closure</config>
        </function>
        <inputPort iterate="false" name="defaultURI">
          <source workflowInputPort="input0"/>
        </inputPort>
        <inputPort iterate="false" name="graph">
          <source processor="node_18"/>
        </inputPort>
      </processor>
      <processor id="node_20" x="722" y="211">
        <function type="sparql">
          <config param="bindVariables">graph</config>
          <config param="query">PREFIX j.11: &lt;http://dbpedia.org/ontology/&gt;
            PREFIX rdfs: &lt;http://www.w3.org/2000/01/rdf-schema#&gt;
            PREFIX j.6: &lt;http://data.linkedmdb.org/resource/movie/&gt;
            PREFIX j.1: &lt;http://dbpedia.org/ontology/&gt;
        </function>
      </processor>
    </network>
  </workflow>
</rdfgears>
```
PREFIX dc: &lt;http://purl.org/dc/terms/&gt;
PREFIX owl: &lt;http://www.w3.org/2002/07/owl#&gt;
CONSTRUCT {
} WHERE {
    GRAPH ?graph {
        FILTER(
            (?p = j.1:abstract) &amp;&amp; ( lang(?o)="en" || lang(?o)="ru" )
            ||
            (?p = rdfs:label) &amp;&amp; ( lang(?o)="en" || lang(?o)="ru" )
            ||
            ?p = j.11:director
            ||
            ?p = j.6:director
            ||
            ?p = owl:sameAs
            ||
            ?p = j.6:runtime
            ||
            ?p = dc:title
        ).
    }
}
</config>
</function>
&lt;inputPort iterate="false" name="graph">
    &lt;source processor="node_19"/&gt;
&lt;/inputPort&gt;
&lt;/processor&gt;
&lt;/network&gt;
&lt;/workflow&gt;
&lt;/rdfgears&gt;
Appendix C: Example of RDFGears
Function Definition

RDFGears Processor

```xml
<?xml version="1.0" encoding="utf-8"?>
<processor label="Local SPARQL" category="Querying">
  <description>Write SPARQL Query, bind the variables in the query to the input fields</description>
  <inputs>
    <function type="sparql">
      <param name="bindVariables" type="InputFields" label="Variables"/>
      <param name="query" label="Query" type="Query">
        Write SPARQL Query
      </param>
    </function>
  </inputs>
  <output>
    <type>
      <var name="output"/>
    </type>
  </output>
</processor>
```

RDFGears User Defined Function

```xml
<?xml version="1.0" encoding="utf-8"?>
<processor label="Fetch RDF" category="RDF Utility">
  <description>Fetch RDF from with specified URI</description>
  <inputs>
    <function type="custom-java">
      <param name="implementation" value="nl.tudelft.rdfgears.rgl.function.standard.FetchRDF"/>
    </function>
    <data iterate="false" name="uri" label="uri">
      <type><var name="uri"/></type>
    </data>
  </inputs>
  <output>
    <type>
      <var name="rdf"/>
    </type>
  </output>
</processor>
```