JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

Re-engineering of a Grid Aware Medical Database System Based on a Metamodel

MASTER'S THESIS

for obtaining the academic title

MASTER OF SCIENCE

in

INTERNATIONALER UNIVERSITÄTSLEHRGANG
INFORMATICS: ENGINEERING & MANAGEMENT

composed at ISI-Hagenberg

Handed in by:

Amira Zaki, 0856338

Finished on:

15th of July, 2009

Scientific Advisor:

A.Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Schreiner

Hagenberg, July 2009

Acknowledgements

I would like to express my gratitude to Prof. Dr. Bruno Buchberger for giving me this opportunity to pursue my Masters studies in an international and stimulating environment, which he initiated in Hagenberg.

I am deeply indebted to my academic advisor, Prof. Dr. Wolfgang Schreiner, for his continuous guidance, constructive advice and constant encouragement throughout the course of production of this work. He has been abundantly helpful and has assisted me in numerous ways.

My stay at RISC/MI was quite interesting and educational. In particular, I would like to thank Dipl. Ing. Thomas Kaltofen for his ideas and endless support.

Sincere appreciation goes to my former advisor, Prof. Dr. Slim Abdennadher, who recommended this Masters program, for being a motivational and influential character for the last six years.

I cannot forget to extend a special thanks to all the ISI students for making our daily life a truly unique experience; together we shared many enjoyable, exceptional and unforgettable times.

Our stay at Hagenberg would have been incomplete without the continuous help and support of Mrs. Betina Curtis, I am grateful to her careful coordination and organisational skills.

Last but not least, this dissertation would not have been complete without the love and dedication of my parents. Throughout the years, I have been dependant on their never-ending support and understanding. To them I dedicate this thesis.

Abstract

Over the last few decades, the amount of medical data available through the advancements in medical and clinical researches has greatly increased. Therefore, the need for huge storage capacities, effective data retrieval methods and high performance computations has become significantly important. The aim of this work was to develop a simplified metamodel for SEE++, a software system for the computer-aided simulation of eye motility disorders and their treatments. The poor performance of the current over-engineered metamodel was the driving force behind the necessary enhancements. Thus, the newly developed metamodel allows the integration into a Grid environment to achieve near real-time performance for the numerous simulations and complex calculations performed. This was done using an instantiation of YAMM, a tool for creating web-based database applications based on a metamodel, comprising both the data and model conjointly in a relational back-end. Our work involved restructuring YAMM to parse its database interactions into a generic form to query any resource. An interoperability layer was created to handle various database connections. The use of a relational back-end was implemented and an interface was designed for using any other system like a Grid-aware database. Moreover, the methodology of instantiating YAMM to represent the SEE++ data model was investigated, resulting in the discovery of a few problems in YAMM's expressiveness, which introduced ambiguity in the SEE++ data model representation.

Table of Contents

List of Figures

Chapter 1

Introduction

Huge advances in the fields of medicine and clinical research have greatly inflated the amount of medical information available. Such an explosion in accessible data has necessitated the need of introducing computer technologies to manage and analyse the data store. The outcome of applying data mining techniques upon medical data would certainly be beneficial, in particular for researchers interested in discerning the complexity of healthcare processes in real-life situations. The knowledge gained could have profound impacts on the understanding of the human anatomy, the cure of diseases and in various medical advances.

The vast amount of data and high computational demands of its processing urge the use of a more advanced computing mechanism. The concept of "Grid computing" arose in the mid 1990's as a means of collaborating various entities into one large computational resource. It is analogous to an electrical power grid, where multiple grids can be connected together to produce an output of even greater magnitude [18]. The Grid promises multiple advantages when serving medical applications, providing them with a large storage capacity and enhanced functionality.

The SEE-KID project is a research project initiated by the Research Unit for Medical Informatics of RISC Software GmbH (RISC/MI), to provide medical decision support for eye surgeons. Its main product, SEE++, is an interactive software system for the biomechanical simulation of the human eye and common eye surgery techniques. The software is used to simulate eye motility disorders such as strabismus (squinting). It allows surgeons to simulate pathological situations and visualise them in a three-dimensional way. Therefore a surgeon can simulate the surgery procedure determining the optimal treatment for a patient prior to surgery. In scope of the SEE-KID project, the SEE-GRID project was established to

extend SEE++ by utilising Grid computing to enhance its capabilities. The use of the Grid aims to achieve near real-time performance for the numerous simulations and complex calculations performed [6].

Currently patient data are stored in binary files. However to meet the technological demands of faster Grid-based simulations and the potential of the insight gained by data mining techniques, it would be better if a database is used for storage. A prototype of a metamodel-based medical database was developed in [29]. The system had serious performance limitations and its Grid integration was not implemented. These complications were investigated in [27], reaching the conclusion that the current database metamodel is too complex and requires simplification.

Another RISC/MI work-in-progress is the YAMM (Yet Another Meta Model) project; a framework for creating web-based database applications built on a metamodel. It stores both the application's data model and the actual data conjointly. Currently YAMM works with a relational back-end using SQL [33].

The primary goal of this work is to re-engineer the database back-end required by the SEE-KID project, in order to allow its integration into a Grid-aware environment. The back-end should be based on a metamodel concept, and include an interoperability interface making it capable of using various back-ends. Our work involved creating a new back-end using a modified version of YAMM, however its instantiation for use with the SEE-KID project was unsuccessful as it revealed some problems in YAMM.

In the scope of this work an interoperability layer was introduced into YAMM. The aim of this layer is to extract all of YAMM's back-end interactions and wrap them into a generic query structure. The interoperability layer contains an interface for translating the generic query into the desired querying language. It also comprises another interface that handles the back-end connections to various database solutions. Our work included the translation into SQL and the connection with a relational database. The potential for other back-end options is possible and the means to use them was highlighted, but not yet implemented in this thesis. An attempt was made to instantiate YAMM to represent the SEE-KID data model. However a problem was discovered in YAMM's expressiveness. It was found incapable of fully indicating the required relations of SEE-KID without introducing ambiguity. Thus it was concluded that YAMM has to be modified to enable integration with SEE-KID.

This thesis is organised as follows: Chapter 2 gives an overview of the concept of metamodels and an introduction to the field of Grid computing, in addition to an explanation

of the two projects involved in relation to this work. Chapter 3 presents the problem addressed; it starts by explaining the existing systems, highlighting their limitations and requirements, finalising with the procedure followed to enhance them. Chapter 4 is the core chapter of the thesis; it discusses the changes done to modify YAMM and introduces the interoperability layer. Chapter 5 elaborates on the attempt made to instantiate YAMM for use with the SEE-KID project and indicates the difficulties encountered. At the end Chapter 6 summarises the work performed and sheds the light upon the recommendations for future enhancements.

Chapter 2

State of the Art

This chapter presents an overview of the basic concepts covered in this thesis. It helps to provide the necessary background information for the concept of a metamodel and Grid computing. In addition the chapter gives an overview of the projects related to this work.

2.1 Models and Metamodels

Data must be stored in some structure for it to be useful. According to [34], a "model" is a particular design or version of a product. It follows that a "data model" is an abstract model that describes how data is represented and accessed. *Entity Relationship Diagrams* (ERD) are graphic tools used to provide a visualisation of the structure of data models for real world problems [10]. They are used in software development processes in the design and analysis of requirements.

As the name implies, ERD model data as *entities* and *relationships*, where entities have *attributes* [4]. An *entity* is an object about which data is being stored. It could be a person, place, object, event or concept. They are usually described by nouns. A *relationship* is an association between two or more entities, and they are usually described by verbs. An *attribute* is a property or characteristic of an entity or relationship. For example, a person (entity) having an id, name, date of birth and gender (attributes) can own (relationship) a car (another entity) having an id, type and colour (attributes). The ERD visualising this is shown in Figure 2.1, where entities are represented as rectangles, relations as diamonds and attributes as ellipses.

The term "meta" originates from the Greek language to mean "after", "with","beyond"

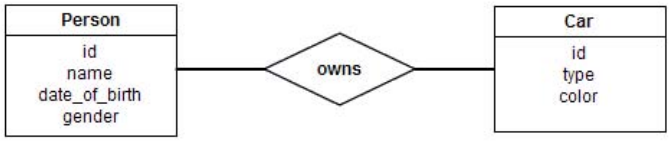

Figure 2.1: Simple ERD showing a person owning a car

or "adjacent" [34]. In the English language, it is used as a prefix to indicate an abstract concept used to give additional information. In computer science, it is commonly used to mean "about" and provides an underlying definition or description. A common term is "metadata" which refers to "data about data".

Consequently putting both definitions together, a "metamodel" is a *model about a model*. It is used to define the basic structure needed to build models. A *meta entity-relationship* model is an entity-relationship model describing an entity-relationship model. The entities in the metamodel are the objects of an entity-relationship model, which are: entities, relations and attributes. Figure 2.2 shows a visualisation of such a meta ERD.

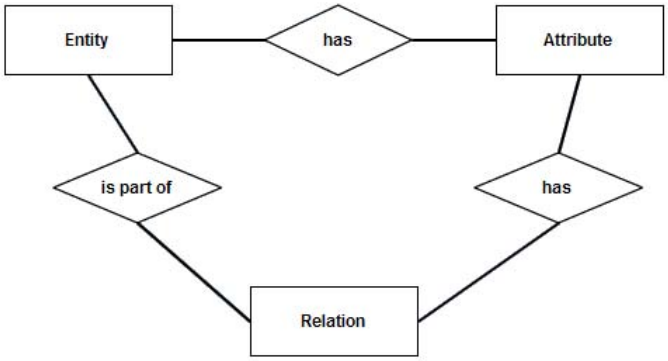

Figure 2.2: Diagram of a meta ERD, consisting of entities, attributes and relations [33]

Metamodels are used in databases to provide a description of the employed database data model in addition to database properties which are not dependant on a particular database state. It should contain information that describes and stores the database schema, together with data such as access rights and integrity and security rules [23].

At a later stage of software development, ERDs are used to build the concrete base of the implementation. Entities are transformed into tables, attributes into columns and relations into foreign key references. This is a direct transformation of the model into a database.

The limitation of using an ERD is that once is it implemented the model cannot be easily

changed. The reason for this is that a single change in the structure of the ERD would result in several changes in the software using that model. If for example an attribute is added to an entity, then the data model has to be changed and the software procedure accessing this entity's table also have to be adjusted.

On the other hand, using a metamodel rather than a direct data model increases the *adaptability* of the system; it allows the easy integration of new products. If it is required to add an attribute to an entity then no changes are performed to the metamodel, however only an entry is inserted in the attributes table. The metamodel implementation remains untouched; this increases the stability of the database containing the metamodel.

2.2 YAMM Project

The Medical Informatics unit of the RISC Software GmbH has developed the YAMM and the SEE-KID projects. The company and the projects are introduced in this section and in Section 2.6.

2.2.1 RISC Software GmbH

RISC Software GmbH[1] is an internationally recognised IT service company. 80% of the company is owned by the Johannes Kepler University (JKU) of Linz (Austria), while the Upper Austrian Research GmbH (UAR)[2] owns the remaining share. The company was founded in 1989 by the RISC institute (Research Institute for Symbolic Computation) of the JKU, which is one of the leading research institutions in symbolic computation, in order to carry out applied and industrial research and development.

RISC Software GmbH is competent in four core areas in which it offers successful solutions to its customers, namely Algorithms, Logistics, Simulation and Software Development Processes. The close cooperation with the institutes of the JKU provides it with a direct access to fundamental science and to the results of the most recent research. This enables RISC Software GmbH to apply innovative solutions to customer projects.

One important part of RISC Software GmbH is the Research Unit for Medical Informatics (RISC/MI)[3]. The department moved in 2008 from the UAR to RISC Software GmbH. Its

[1] www.risc-software.at
[2] www.uar.at
[3] www.risc-mi.at

work focuses on the development of medical software systems for computer-aided clinical support. Research projects for simulating eye muscle surgeries and the treatment of burn injuries that are already carried out at RISC/MI provide important know-how in the fields of image processing, medical simulation and visualization. RISC/MI strengthens the development of new methods for clinical diagnostics and treatment planning on the basis of interdisciplinary research that combines medical and technical knowledge.

2.2.2 YAMM

One of the recent developments of the RISC/MI is a tool for creating web-based database applications based on a metamodel. The tool is called YAMM (Yet Another Meta Model) [33]. It was created for the intention of being used in one of the projects of the department, in particular to design and maintain a database for Aneurysm patients. An aneurysm is a balloon-like bulge in an artery; it can grow large and burst, thereby causing dangerous bleeding inside the body [28]. The database stores information about the patients and the investigations performed for them. Currently YAMM is being extended to support any medical application and thus increase its scalability; the work of this thesis will help support this generalisation.

YAMM uses a web-based interface developed in PHP to connect to a MySQL database back-end. The PHP forms directly interact with the database to insert, delete and update the data model and the data itself.

YAMM has a fixed data model consisting of eleven entities [33], to store information regarding the used data model, the actual data and additional metadata. Three entities (*Entity, Relation, Attribute*) are used to describe the data model as explained earlier. The entity *Type* is used for the data types of attributes; to represent types like integer, float, string and so on. The data itself is stored via two entites, one to represent an instantiation of an entity (*Record*) and one to store the contents of attributes (*Value*).

The remaining entities are used for metadata purposes to store access rights and for applying integrity rules. A user management and auditing system is used through three entities (*User, Usergroup, Audits*) to keep track of the users of the system and a log of the changes they perform. The last two entities (*Filter* and *Visibility*) are used for the functionality of the system, like providing a means to constrain the choices for user input as required by the tool's front-end interface. The ERD for YAMM showing the eleven entities, their attributes and how they are related is shown in Figure 2.3.

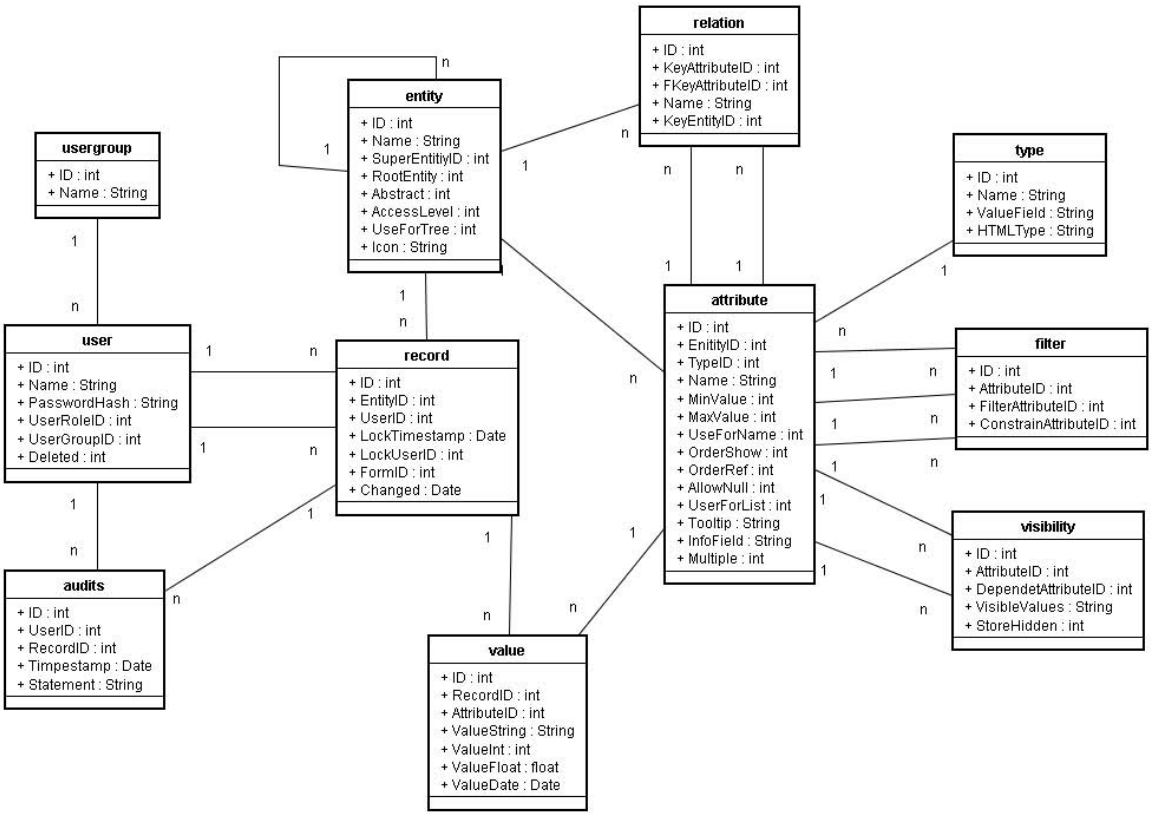

Figure 2.3: ERD of YAMM [33]

To illustrate how YAMM works and stores both the data and the data model, given below are the entities present in YAMM for the previously explained example of a person owning a car (shown in Figure 2.1). Figure 2.4 shows the entries required to represent the ERD of the example, i.e. the entities, attributes and relations entries. Figure 2.5 shows the entries required to insert data of a female patient named "Jane Smith" with an ID of 1234 and born on 9/1/85. The figures describe only a subset of the actual columns for each table.

YAMM provides a web-interface to add, update and delete entries in its eleven entities. This facilitates an easy mechanism to update the data stored in the database and also the model from which the data is structured. Snapshots of the interface are shown in Figures 2.6 and 2.7.

Any data model must have a certain prerequisite in order to make it feasible to model with YAMM. The restriction is that all the data stored should revolve around one single root element. This base element should comprise all the sub-data entries. For the Aneurysm

Entity

ID	Name
1	Person
2	Car

Relation

Name	KeyAttributeID	FKeyAttributeID
owns	1	5

Attribute

ID	Name
1	Person_ID
2	Name
3	Date_of_birth
4	Gender
5	Car_ID
6	Type
7	color

Figure 2.4: ERD entities of person owning car example

Value

ID	RecordID	AttributeID	Value String	Value Int	...
1	1	1	*null*	1234	
2	1	2	Jane Smith	*null*	
3	1	3	9/1/85	*null*	
4	1	4	female		

Record

ID	EntityID
1	1

Figure 2.5: Storing one record of data of the Person entity

database this is the patient entity; it contains all the patient diagnosis and treatment data. Any application can be used with YAMM provided that it abides by this restriction.

2.3 Grid Computing

Originally, the term "Grid" comes from an analogy with the electrical "power grid". The idea was that accessing computer power from a computer grid would be as simple as accessing electrical power from an electrical grid [18]. The output obtained should be of great strength, and multiple grids can be connected to one another to produce an output of even greater strength.

In the mid 1990s, the term "Grid" was used to describe a distributed infrastructure for advanced science and technology. It is based on the idea of collaboration and getting various entities to work together. It can be defined as coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organisations [12]. Resource sharing

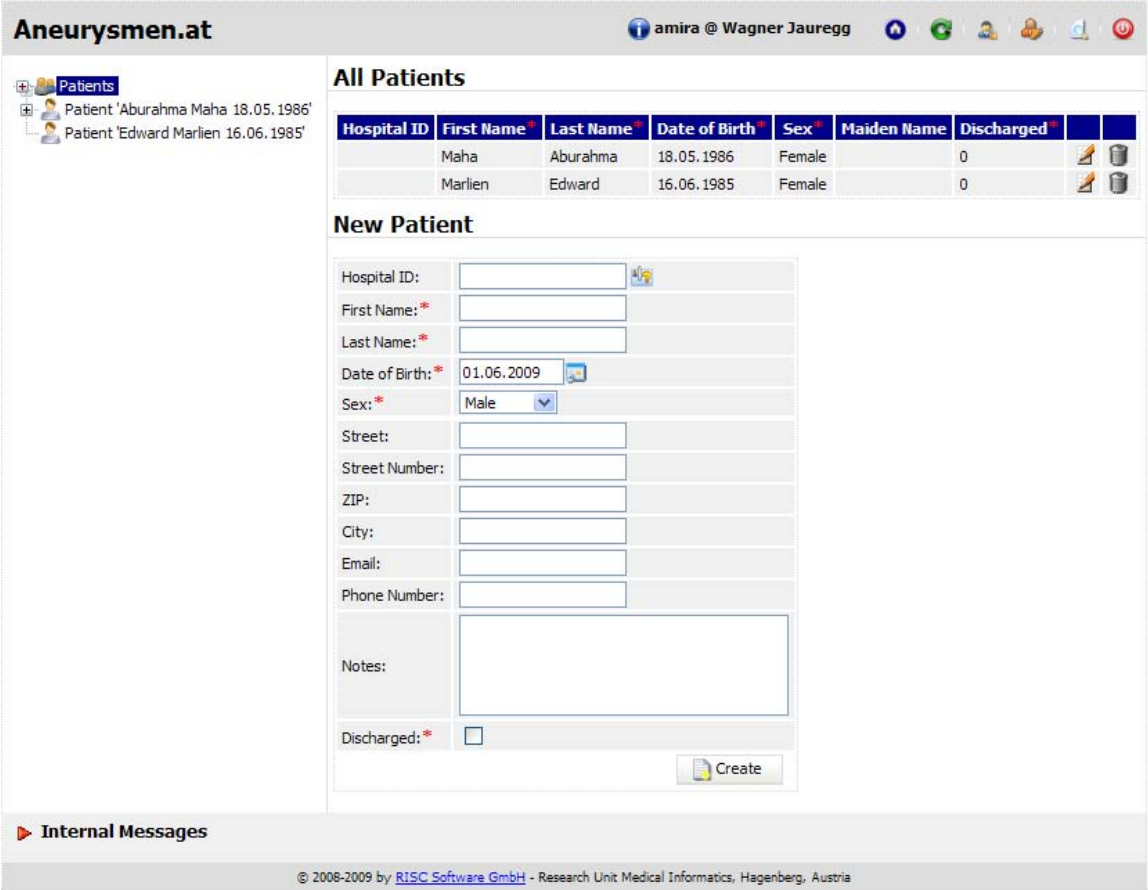

Figure 2.6: A snapshot of the YAMM interface for entering patient data

is not restricted to file sharing; it is rather concerned with direct access to computers, remote software, data, sensors and other resources. The sharing is controlled in an effective way, such that the shared content, sharing parties and sharing conditions are well defined. Individuals and institutions following these conditions build up virtual organisations [14].

The power of the Grid comes from operating a global network of computers (virtual organisations that are geographically distributed) as one large computational resource, as illustrated in Figure 2.8. This vast computational resource can be utilised in various applications to make use of expertise (like in visualisation of large scientific data sets), to make use of power and storage (like in computationally demanding data analysis), or to increase functionality and availability (like in coupling of scientific instruments with remote computers and archives) [13].

Like all engineering constructs, a Grid has a certain architecture or design which identifies

Aneurysmen.at - Administration amira @ Wagner Jauregg

Entities | Relations | Types | Audits | Filters | Visibilities | Users

Entities

ID	Name	Super	Root	Abstract	UseForTree	Access	Icon
16	Aneurysm	(none)	☐	☐	☑	1 Everyone	aneurysm
15	Aneurysm Type	(none)	☐	☐	☑	1 Everyone	(None)
55	Clinical Event	(none)	☐	☐	☑	1 Everyone	(None)
56	Clinical Followup	Followup (52)	☐	☐	☑	1 Everyone	flag
49	Coil	(none)	☐	☐	☐	1 Everyone	(None)
51	Coil Company	(none)	☐	☐	☑	1 Everyone	(None)
57	Coil Thickness	(none)	☐	☐	☑	1 Everyone	(None)
50	Coil Type	(none)	☐	☐	☑	1 Everyone	(None)
47	Complications	(none)	☐	☐	☑	1 Everyone	(None)
54	Consequence	(none)	☐	☐	☑	1 Everyone	(None)
23	Cycle	(none)	☐	☐	☑	1 Everyone	time
37	Cycle Presentation	(none)	☐	☐	☑	1 Everyone	(None)
36	Dome to Neck Ratio	(none)	☐	☐	☑	1 Everyone	(None)
41	Endovascular Treatment	Treatment (40)	☐	☐	☑	1 Everyone	hospital
52	Followup	(none)	☐	☐	☑	1 Everyone	flag
35	Form	(none)	☐	☐	☑	1 Everyone	(None)
32	Location	(none)	☐	☐	☑	1 Everyone	(None)
33	Modality	(none)	☐	☐	☑	1 Everyone	(None)
46	Morphological Result	(none)	☐	☐	☑	1 Everyone	(None)
4	Patient	(none)	☑	☐	☑	2 Group	patient
44	Premedication	(none)	☐	☐	☑	1 Everyone	(None)
34	Previous Treatment	(none)	☐	☐	☑	1 Everyone	(None)
45	Procedural Medication	(none)	☐	☐	☑	1 Everyone	(None)
31	Sex	(none)	☐	☐	☑	1 Everyone	(None)
53	Status	(none)	☐	☐	☑	1 Everyone	(None)
48	Surgical Method	(none)	☐	☐	☑	1 Everyone	(None)
43	Surgical Treatment	Treatment (40)	☐	☐	☑	1 Everyone	hospital
40	Treatment	(none)	☐	☐	☑	1 Everyone	hospital
39	WFNS	(none)	☐	☐	☑	1 Everyone	(None)
*		(none)	☐	☐	☐	1 Everyone	(None)

Internal Messages

© 2008-2009 by RISC Software GmbH - Research Unit Medical Informatics, Hagenberg, Austria

Figure 2.7: A snapshot of the YAMM interface for changing the entities of the data model

the fundamental components of the Grid, defines their purpose and functionality and specifies how they interact together. As described in [12], the Grid can be described by a four layered architecture. The lowest layer is the *fabric* layer which provides the Grid resources such as computational resources, storage systems and network resources. The second layer is the *resource and connectivity protocols* layer. This layer defines the protocols needed for

communication and authentication for Grid-specific network transactions between the entities described in the fabric layer. The third layer is the *collective services* layer; this defines the protocols and services regarding the interactions across collection of resources, enabling the high-level elements of the Grid to communicate with one another. The top-most layer is the *user applications*; this layer is composed of the Grid-aware applications which users interact with.

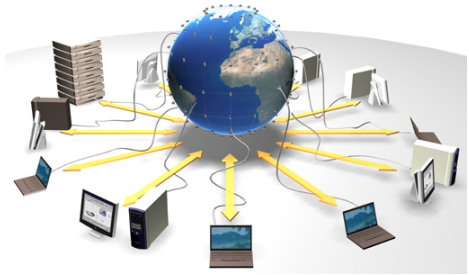

Figure 2.8: The Grid: Global network of computers and resources [15]

Many technologies have evolved over time to implement Grid architecture concepts. They aim at providing a mechanism for sharing and coordinating the use of different resources from distributed components, to provide an integrated service.

In particular, the Globus toolkit has developed since the late 1990's to become what The New York Times called "the de facto standard" for Grid computing [17]. It is an open source architecture, that includes software services and libraries for resource monitoring, discovery, and management, in addition to security and file management. The Globus toolkit plays a central part of science and engineering projects and many large scale applications rely on it, such as the Network for Earthquake Engineering and Simulation (NEES)[4] and the Earth System Grid (ESG)[5].

Another Grid architecture called gLite was developed by the European EGEE (Enabling Grids for E-sciencE) [6]. It was formed with the collaborative efforts of more than 80 people in 12 different academic and industrial research centres. It provides an advanced framework for building Grid applications utilising the power of distributed computing and storage resources across the Internet. The Grid infrastructure consists of more than 41000 CPUs with around five Petabytes of memory. It is well suited for the uses of scientists whose needs arise in performing large, highly complicated calculations.

[4] www.nees.org
[5] www.earthsystemgrid.org
[6] www.eu-egee.org

2.4 Grid Data Management

Grid systems are formed to solve problems too complex or too expensive to solve with local resources, especially for data-intensive applications. Problem solving is concerned with the consumption and production of information. Thus *information on the Grid* is important. The Grid itself is a complex and information-rich environment. Grid middleware uses information about the availability of services, their purpose, ways in which they can be combined and configured, and how they are discovered, invoked and evolve. Thus *information about the Grid* is also important. Hence the amount of data to be handled whether on or about the Grid is vast, and mechanisms should exist to efficiently manage this data [12].

It is necessary to have data Grid services that ensure secure, reliable and efficient data transfer and the ability to register, locate and manage multiple copies of datasets [1]. There are numerous solutions to provide these services, such as GridFTP, RFT and RLS.

GridFTP is a protocol developed by the Globus Alliance, it transfers bulky data in a secure, robust, fast and efficient means [20] and is commonly used to handle file-based data. Another solution is the Reliable File Transfer (RFT) service, which performs transfer of files and directories but in a "job-scheduler" mechanism that monitors the transfer status and supplies status notifications [21].

It is necessary to have copies of data available (known as replicas) in various locations on the Grid in order to reduce access latency, and improve reliability and load balancing. Replica management services are thus needed to create the copies, register them in a replica catalogue and find all existing copies of a file when queried. Globus toolkit has a component related to data replication known as the Replica Location Service (RLS). It offers a distributed registry that keeps track of replicas in a Grid environment [22].

2.5 Grid Database Resources

The amount of data produced in research and business environments has increased greatly, thus the need arose for managing the massive amounts of data across organisational and geographical boundaries. Integrating data from multiple databases and making use of the distributed data has become an important issue in many Grid-based applications. Many approaches exist for dealing with this issue, in particular are the OGSA-DAI project and the AMGA Metadata Catalog.

OGSA-DAI

The OGSA-DAI project was initiated in 2002 [35]. It offers means to develop effective data management, with special rules regarding data access and intergration. It allows resources like relational, XML or file-based databases to be accessed via the Grid. It is composed of two main products: OGSA-DAI and OGSA-DQP.

OGSA-DAI allows querying and updating numerous data sources through a web service based presentation layer. The data sources can be of different types. OGSA-DQP is a Distributed Query Processing service which extends OGSA-DAI allowing queries to be executed over remote collections of relational data services.

Numerous research organisations and users from Europe, USA, China, Japan and Russian are now using OGSA-DAI to make their data resources Grid-enabled. It is used in various fields such as geographical information systems, meteorology, transport, computer-aided design, engineering, astronomy and medical research.

AMGA Metadata Catalog

AMGA (ARDA Metadata Catalogue Project) is a service that allows Grid users to store various metadata used by applications. It studies the needs on metadata catalogues in Grid environments and provides an interface for metadata access on the Grid [3]. This means of database access service allows tasks running on the Grid to access databases by providing a Grid style authentication, in addition to an opaque layer which hides the different underlying database systems from the user. It also provides a replication layer which enables local access of databases and means to replicate changes between the different databases [26].

2.6 SEE-KID Project

2.6.1 SEE-KID

One of the main projects of RISC/MI is the SEE-KID[7] project, which is a Software Engineering Environment for Knowledge-based Interactive Eye Motility Diagnostics. The SEE-KID project was initiated by Prof. Dr. Siegfried Priglinger, former head of the ophthalmologic department at the convent hospital of the “Barmherzigen Brüder” in Linz. In its initial phase, the project was carried out by the Upper Austrian University of Applied Sciences

[7] www.see-kid.at

in Hagenberg in cooperation with the convent hospital of the "Barmherzigen Brüder" in Linz, the Department of Neurology of the University Hospital Zürich (Switzerland) and the Smith-Kettlewell Eye Research Institute (United States, San Francisco). In 2003 the UAR founded the Department of Medical Informatics in Hagenberg, which has continued the development of the SEE-KID project since then. Later on in 2008, the department moved to RISC Software GmbH and the UAR owns 20% of its shares.

One of the main results of the SEE-KID project is the Simulation Expert for Eyes + Diagnoses + Surgery Simulation (SEE++) [9]; an interactive software system for simulating eye motility disorders like strabismus, and the required surgeries to treat them.

Strabismus, commonly known as squinting or having crossed eyes, is a disease which hinders coordinated eye movement. It is a condition in which a person suffers from a misalignment of the eyes, having one or both of the eyes point in different directions. This results in failing to focus the eyes on a single object and experiencing double images. Various treatment methods exist, including use of special eye glasses and a treatment technique involving a method of covering the normal and/or squinting eyes. However a more effective technique is the use of strabismus surgery [8]. The surgery is complex because it involves six extraocular muscles, which are related in a way that changing one muscle affects the others. Surgeons have no real possibility to determine what technique to use nor which muscles to operate on, therefore multiple surgeries are needed before an effective treatment is reached. Thus having a computer-based simulation of the surgery beforehand can facilitate the chance for doctors to simulate the surgical procedure before it is performed. This improves the possibility of performing surgery that corrects the eyes sufficiently after the first surgical treatment [7].

The SEE++ simulation software can be used for strabismus surgeries, by allowing surgeons to simulate pathological situations and to visualise them graphically and interactively in a three dimensional way, as shown in Figure 2.9. It calculates the effects of eye muscle surgeries in comparison to a selected standard model thus enabling the surgeon to observe the trend of the surgery and determine the respective optimal treatment for a patient before performing the surgery. Without software support, these treatments are very complicated to plan and a patient would usually require more than one operation.

Currently SEE++ stores patient records into binary files. This suffices the current needs, however there are a couple of enhancements in SEE++ that are better addressed if a relational database is used to store patient records.

Moreover the simulation process in SEE++ is very time consuming because the surgeon

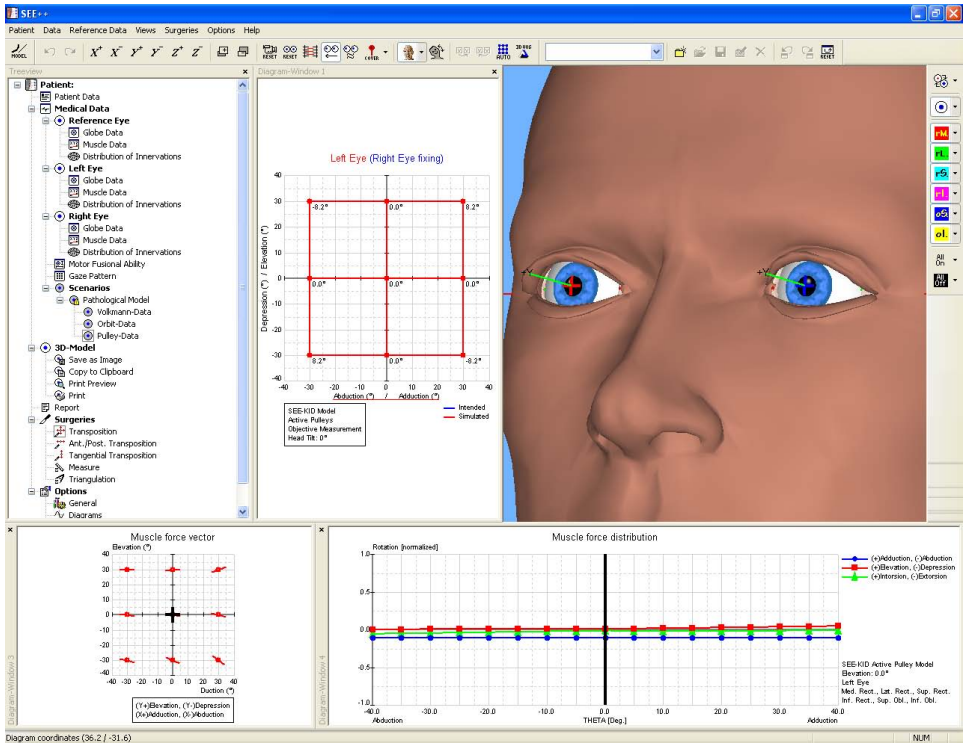

Figure 2.9: A snapshot of the SEE++ software

has to adapt many parameters manually. A parameter optimisation algorithm can help suggest values for these parameters; this would guide the creation of simulations. To speed up the optimisation computation, the algorithm could obtain the starting parameters from previously stored patient records. Consequently using binary files to store the patient records is not a suitable means and a database is required. The various ways to investigate the possibility of accelerating the software were studied in the master work of Johannes Watzl [36].

2.6.2 SEE-GRID

The higher requirements of the system led to the initiation of the SEE-GRID project. SEE-GRID is a research project aimed to develop a Grid-enabled version of SEE++. This version shall utilise the computational power of Grid computing techniques and provide near real-time performance for the simulations and calculations performed. The development is carried out in cooperation with the Austrian Grid Development Center (AGEZ), a branch

of the Austrian Grid[8] project, which is run by the RISC Software GmbH. Currently there exists a Grid-based implementation of the computation and calculation services required by SEE++ [5] and a prototype of a metamodel-based medical database. The database solution and a corresponding data model were developed in the Diploma thesis of a student from the Upper Austrian University of Applied Sciences in Hagenberg [29].

The database system had important performance limitations and its Grid integration was still missing. These issues were investigated in the Master thesis of a student (Imre Matkó) from ISI-Hagenberg [27]. Based on the benchmarking and profiling of the SEE-GRID database components, it was concluded that the current database metamodel is too complex. This causes most of the runtime to be spent with database queries and data transformations. Consequently, the need for a simplified metamodel to improve the performance of the database system has arisen, which is the primary aim of the work of the present thesis.

The master thesis, work of Matkó, also involved evaluating several Grid data-resource management tools. Two tools, namely OGSA-DAI [35] and AMGA [16], were found to be promising solutions for the project. The design of the SEE-GRID database was extended with a Globus "Web Service Resource Framework" based interface and with the services of OGSA-DAI. This design was prototyped and OGSA-DAI was evaluated with respect to the needs of SEE-GRID. The actual integration of either of these tools was not performed yet.

[8] www.austriangrid.at

Chapter 3

Problem Statement

This chapter aims to present the problem addressed by the thesis. First an overview of the existing systems is given; highlighting their problems and limitations. Then the requirements needed by the systems are elaborated. In the end, the solution approach is explained to identify the path in which the requirements are to be fulfilled.

3.1 Existing Systems

The focus of the thesis handles two projects implemented by RISC/MI, namely YAMM and SEE-KID (including both SEE++ and SEE-GRID). Both projects were introduced briefly in the previous chapter. This section provides technical details on the current status of their systems.

3.1.1 Current Version of YAMM

As discussed in the previous chapter, the current implementation of YAMM offers means for designing and storing both the metamodel and the actual data conjointly. At the front-end, the system has a web-based interface developed in PHP. Through this interface it is possible to manually change the structure of the metamodel and also to manipulate the data being stored. At the back-end, the system connects to a MySQL database in order to store the data and data model amongst the 11 entities described earlier. This is illustrated in the system architecture shown in Figure 3.1.

Although the system was implemented for use in the Aneurysm database project, at the

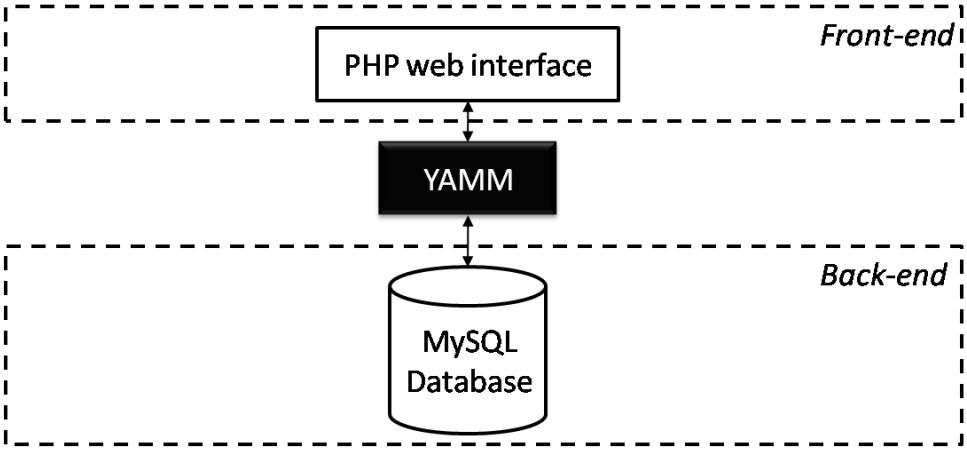

Figure 3.1: Current system architecture of YAMM

time of writing, many efforts are being made to extend YAMM and generalise its features. The aim is to make it flexible enough to refactor the existing data model and integrate it into different applications. One such effort is the extension of the front-end of YAMM to have a mechanism to export the data stored as an XML document. This new feature facilitates easy data transmission through XML documents. Another extension is the means to import the metamodel as an XML Schema Document (XSD). This will allow the automation of the processes of instantiation of the metamodel for use in any application.

At the back-end, YAMM is limited to the use of a connection to a relational database, in particular to a MySQL database. It is not possible to use any other back-end like an XML database or a Grid-aware database. YAMM's PHP scripts directly connect to the MySQL database and manipulate the data stored.

3.1.2 Current SEE++

As mentioned in the previous chapter, the current database system of SEE++ is a metamodel-based implementation which uses web service technologies for communication [29]. Its core is a "persistence component" which realises the Transport Metamodel Mapping in order to integrate the database in Grid environments. In particular, it uses the Spring framework for its Object/Relational mapping functionality using the open source tool called Hibernate [24]. However this persistence component offers poor performance while the aim is to have real-time operations.

The performance of the current model was evaluated in the work of Matkó [27] where he

identified two main bottlenecks: the database operations requested by the object-relational mapping and the object graph transformation to and from the transport model. The reason for this is that the transport model and the metamodel based representations are quite complex with a 3–4 level tree based organisation. Since the software does not internally operate with the transport model, the performance of the object-relational mapping is influenced by the internal representation. Therefore, a possible improvement could be the use of a simpler metamodel instead of the one currently in place. Another possible improvement could be to find a more optimised database access process which involves fewer data transformations.

Moreover, the component supports only one data source; thus the data can be stored on a single node only. The whole persistence component is thus over-engineered and uses a metamodel which is too general.

The main performance bottleneck of the software is the over-engineered and complex metamodel which causes a non-linear decrease in performance as the size of the processed data is increased. He recommended optimising the metamodel and the data transformation process as this should increase performance.

3.1.3 Current SEE-GRID

As mentioned earlier, the aim of the SEE-GRID project was to have a Grid enabled version of SEE++ that utilises Grid computing techniques since algorithms may be required to be executed on the Grid. The Grid-aware persistence component realises the integration of the database system into Grid environments. The integration into Grid environments such as gLite or Globus is not a complex task because the component has a web service based design. The work of Matkó [27] involved designing an access interface for data resources which will eventually use higher level Grid services provided by special middleware solutions. Matkó evaluated two such services, namely OGSA-DAI and AMGA and he provided, in his thesis, a comparison of them. His work also included the extension of the current design of SEE-GRID and he provided a Globus web service interface for the database using OGSA-DAI. He also recommended considering the future integration of the database into other Grid toolkits like gLite.

3.2 Requirements

The projects implemented by RISC/MI are still on-going projects. The systems presented have certain requirements which should be met for future enhancements, this section highlights their present needs.

3.2.1 YAMM

For the sake of increasing the flexibility of YAMM and in an attempt to meet the needs of various applications, it is required to create an interoperability layer for the generalisation of the type of database back-end used. The idea is to change the back-end of YAMM, so that it implements an interface which connects to an abstract interoperability connection layer written in a programming language which supports various database connections. An example of such a language is Java, since it offers various means for database connectivity in addition to its platform-independence [11]. The system architecture for the new requirements of YAMM is illustrated in Figure 3.2.

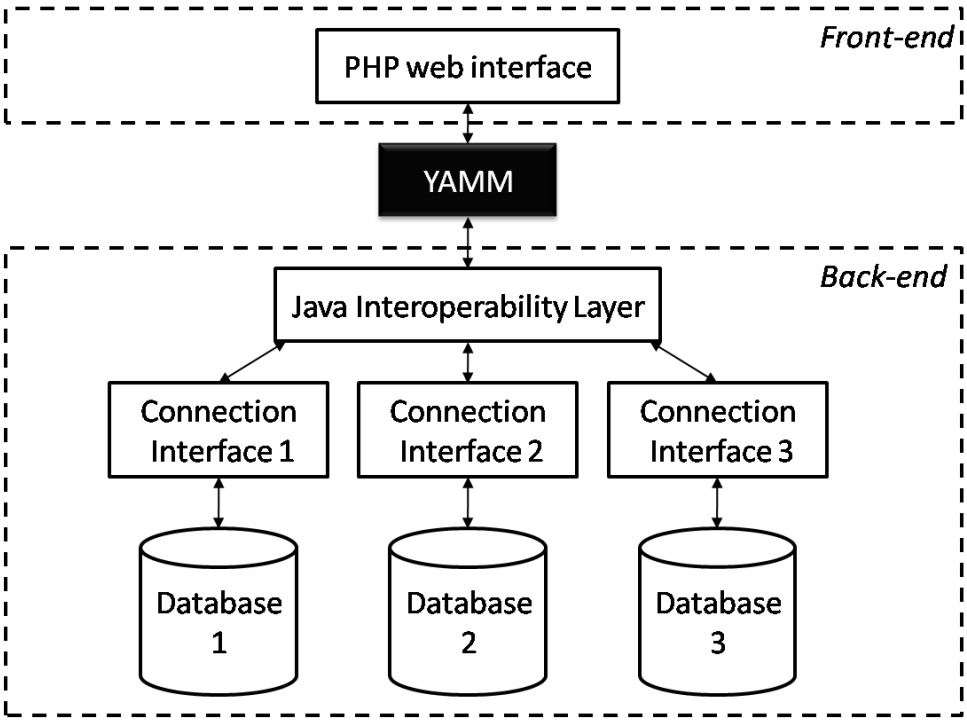

Figure 3.2: Recommended YAMM system architecture

In order for the abstract interoperability layer to function, a structure should be designed as a common means to store all queries generated by YAMM. Thus it is required that a survey is first done for all the different types of queries that YAMM generates. This is needed to help identify the optimum structure necessary to represent a query. The interoperability layer should then offer an interface for parsing the queries to whatever syntax is required by the intended database back-end and also the methods necessary for the database connections.

3.2.2 SEE++ and SEE-GRID

Due to the findings of Matkó regarding the complexity of the currently available persistence component, we are driven to re-engineer the component and suggest a more simplified metamodel. This need arises with various concerns regarding the optimisation of the performance of other components. The new simplified metamodel should be able to store the transport metamodel used by SEE++. Moreover it should be feasible to be Grid-aware allowing use of technologies such as OGSA-DAI and/or AMGA back-ends. It is possible to totally remove the Hibernate persistence component and replace it with a new simplified mapping and back-end layer. However, the new persistence component must be based on a database management system. Its functionality should remain at least equal to the existing file-based persistence component because both components will exist in parallel. Moreover, it should be possible to import and export patient records using files, and also to save, load or delete them using the database persistence mechanism.

3.3 Solution Approach

After analysing the requirements of the existing systems, an inspiration arose to modify both systems and merge them together. The idea is to create the interoperability layer for YAMM as illustrated in Figure 3.2 to enable the use of multiple database back-ends. The layer will be written in Java and a bridge between Java and PHP will be required for the interactions between the front-end and the interoperability layer. YAMM will be used to design and manage the metamodel required by SEE++. This can be done by importing the XSD (XML Schema Document) of the existing SEE++ transport model into YAMM, using the newly created importing mechanism. YAMM will store the model in its simple means and enable querying of the data through the interoperability layer regardless of the choice of database. This enables testing the performance of multiple database options.

Chapter 4

Metamodel Design and Implementation

This chapter describes the work done in order to produce the interoperability layer required for the generalisation of YAMM. It starts by the analysis of YAMM in order to identify the exact requirements of the layer. This is followed by a description of the Generic Query Structure suggested for representing all queries of YAMM and to be used for all database interactions. After that a discussion is presented of the interoperability layer starting with its design, then the technicalities involved, the actual implementation and samples of modifications performed to introduce this layer into YAMM. At the end of the chapter, we present the back-end interfaces that the interoperability layer requires; where the common interfaces for parsing and database connectivity are discussed, followed by a specific example of an implementation for an SQL parser and a relational database connector are given.

4.1 Existing YAMM Implementation

YAMM is developed in PHP. PHP is a general-purpose scripting language that is especially suited for web development [31]. It originally stood for Personal Home Page, but it is nowadays known for the recursive acronym "PHP Hypertext Preprocessor". It is unlike an ordinary HTML page in that a PHP script is not sent directly to a client by the server; instead, it is parsed by the PHP module, which is installed on the server. HTML elements in the script are left alone, but PHP code is interpreted and executed. PHP code in a script can query databases, create images, read and write files and talk to remote servers. The

output from PHP code is combined with the HTML in the script and the result is sent to the user's web-browser.

PHP can be deployed on most web servers, many operating systems and platforms, and can be used with many relational database management systems. PHP offers several extensions for handling databases; there are many vendor specific database extensions that provide functions for interactions with database severs. For example, there exists such extensions for MySQL, PostgreSQL and Oracle databases [31].

YAMM works by using pages written in PHP to connect to the MySQL database to store the representation of the data model and the actual data as discussed earlier. YAMM contains 27 PHP scripts which are necessary for the functionality of the system. It uses the functions provided by PHP MySQL extension for the database interactions. Mainly there is a script (`YDB.php`) containing methods that handle the possible interactions with the database; the functions perform functionalities such as executing a query, querying for a single result, returning the number of rows present in the last query, starting, committing or rolling back a transaction and so on. These functions all use the provided MySQL functions. The queries to be issued are all given in SQL syntax. The queries themselves are formulated in the other PHP scripts, then for executions the methods are called from the `YDB` script. Some queries are fixed, others are built on demand depending on the search criteria. The scripts contain around 130 queries which are constructed for the needs of YAMM.

4.2 Analysis of YAMM Queries

In order to introduce the interoperability layer, all database interactions must be analysed; in particular the queries constructed by YAMM should be inspected. The aim of this section is to provide a summary of the different queries and suggest a means of clustering them in order to investigate the minimal SQL syntax used.

YAMM constructs around 130 queries in SQL syntax. These SQL statements can be divided into two sets, one set as a means of *data manipulation* to perform query and update commands, and another set as a means of *data definition* to permit database tables to be created or deleted, to define indexes, to specify links between tables, to impose constraints between tables and to obtain information about the database [19].

Both sets of statements define the database interactions of YAMM and thus should be both represented accordingly. However since the majority of commands used in YAMM are used to manipulate data, the query model was chosen to be optimal for that purpose.

Consequently the subset of SQL statements used by YAMM was extracted and examined in order to design a query structure optimised for the subset of the YAMM SQL syntax.

The data manipulation queries used by YAMM are of *four* types:

- `SELECT`: to extract data from the database.
- `UPDATE`: to update data in the database.
- `DELETE`: to delete data from the database.
- `INSERT`: to insert new data into the database.

Though they are not many, the data definition statements are of *three* types:

1. **Transactional Statements** - control transactions in database access. A database transaction is a larger unit that comprises multiple SQL statements. A transaction ensures that the action of the multiple statements is atomic with respect to recovery. A classic example is a bank operation that transfers funds from one type of account to another, requiring updates to two tables. Transactions provide a way to group these multiple statements in one atomic unit. YAMM uses such transactions, and the type of statements used are as follows:
   - `BEGIN`: begins a new transaction.
   - `COMMIT`: commits the current transaction, making its changes permanent.
   - `ROLLBACK`: rolls back the current transaction, cancelling its changes.
   - `SET autocommit`: disables or enables the default autocommit mode for the current session; an enabled autocommit mode means that as soon as a statement that modifies a table is executed, the update is stored on disk to make it permanent.
2. **Database Administration Statements** - are used for the purpose of various database administrative issues, like showing or viewing some of the information, or managing user accounts and the access control system. YAMM uses only one such statement:
   - `SHOW`: has many forms that provide information about databases, tables, columns, or status information about the server.
3. **Data Definition Statements** - are used to create the database structures that will hold the data. In YAMM the only type of data definition statement used is:

- `CREATE OR REPLACE VIEW`: creates a new view or replaces an existing one; where a view is a virtual table based on the result-set of an SQL statement.

4.3 Specification of YAMM Queries

4.3.1 SELECT

The generic syntax of any `SELECT` statement as obtained from the MySQL documentation [30] is as follows:

```
SELECT
    [ALL | DISTINCT | DISTINCTROW ]
      [HIGH_PRIORITY]
      [STRAIGHT_JOIN]
      [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
      [SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
    select_expr [, select_expr ...]
    [FROM table_references
    [WHERE where_condition]
    [GROUP BY {col_name | expr | position}
      [ASC | DESC], ... [WITH ROLLUP]]
    [HAVING where_condition]
    [ORDER BY {col_name | expr | position}
      [ASC | DESC], ...]
    [LIMIT {[offset,] row_count | row_count OFFSET offset}]
    [PROCEDURE procedure_name(argument_list)]
    [INTO OUTFILE 'file_name' export_options
      | INTO DUMPFILE 'file_name'
      | INTO var_name [, var_name]]
    [FOR UPDATE | LOCK IN SHARE MODE]]
```

After inspecting the `SELECT` statements used by YAMM, the minimal statement structure with the used clauses is as follows:

```
SELECT
    [DISTINCT]
    select_expr [, select_expr ...]
```

```
    [FROM table_references
    [WHERE where_condition]
    [ORDER BY {col_name} [DESC], ...]
```

The `DISTINCT` keyword indicated is required to filter the result set from duplicates; use of this keyword is optional.

The list of `select_expr` terms comprises the select list that indicates which columns to retrieve. Terms specify either a column, an expression or can use `*` (to select all columns). There must be at least one `select_expr`. Moreover it should be possible to model expressions using aggregate functions. YAMM requires the use of two aggregate functions: `MAX` and `COUNT`, in the form of: `COUNT(column_name)` or `MAX(column_name) + constant`. Furthermore it could be possible that the `select_expr` is the result obtained from a `SELECT` statement, thereby nesting of `SELECT` statements inside the `select_expr` term should be feasible.

A `select_expr` can be given an alias using `AS alias_name`. The alias is used as the expression's column name and can be used in the `ORDER BY` clause.

The `FROM table_references` clause indicates the table or tables from which to retrieve rows. It is possible to use multiple tables by performing a join. For each table specified, one can optionally specify an alias:

```
tbl_name [[AS] alias]
```

The syntax of the `JOIN` expression used in the `table_references` as required by YAMM's SQL statements is less than the general one. In particular the necessary structure is as follows:

```
table_references:
    table_reference [, table_reference] ...

table_reference:
    table_factor
  | join_table

table_factor:
    tbl_name [[AS] alias]
  | ( table_references )
```

```
join_table:
    table_reference JOIN table_factor join_condition
  | table_reference {LEFT|RIGHT} [OUTER] JOIN table_reference join_condition

join_condition:
    ON conditional_expr
```

The `WHERE` clause of the `SELECT` statement, if given, indicates the condition or conditions that rows must satisfy to be selected. `where_condition` is an expression that evaluates to true for each row to be selected. The statement selects all rows if there is no `WHERE` clause.

The structure of the `where_condition` is as follows:

```
where_condition:
    conditional_expr
  | where_condition AND where_condition
  | where_condition OR where_condition
```

The structure of the `conditional_expr` which expresses a condition on a `column_name` is as follows:

```
conditional_expr:
    column_name operator column_name
  | column_name operator VALUE
  | column_name IN select
  | LIKE PATTERN
operator:
    = | != | <> | > | >= | < | <=
```

The `ORDER BY` clause, if given, specifies the desired ordering of the results. To sort in reverse order, the keyword `DESC` (descending) is added to the name of the column in the `ORDER BY` clause that you are sorting by. The default is ascending order; in YAMM this is specified by not writing any trailing keyword.

4.3.2 UPDATE

The generic syntax of any `UPDATE` statement is as follows:

```
UPDATE [LOW_PRIORITY] [IGNORE] table_reference
    SET col_name1={expr1|DEFAULT} [, col_name2={expr2|DEFAULT}] ...
```

```
    [WHERE where_condition]
    [ORDER BY ...]
    [LIMIT row_count]
```

After inspecting the `UPDATE` statements used by YAMM, the minimal statement structure is as follows:

```
UPDATE table_reference
    SET col_name1=expr1 [, col_name2=expr2] ...
    WHERE where_condition
```

The `UPDATE` statement updates columns of existing rows in the named table with new values. The `SET` clause indicates which columns to modify and the values they should be given. Each value can be given as an expression.

The `WHERE` clause specifies the conditions that identify which rows to update. The structure of the `where_condition` is as described earlier.

4.3.3 DELETE

The generic syntax of any `DELETE` statement is as follows:

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM tbl_name
    [WHERE where_condition]
    [ORDER BY ...]
    [LIMIT row_count]
```

After inspecting the `DELETE` statements used by YAMM, the minimal statement structure is as follows:

```
DELETE FROM tbl_name
    [WHERE where_condition]
```

The table from which the rows are to be deleted is given by the `tbl_name`. The selection of rows to be deleted is done by specifying the `where_condition`, whose structure is as explained earlier. This clause is optional, and when it is not specified then all rows are deleted from the target table.

4.3.4 INSERT

The generic syntax of any `INSERT` statement is as follows:

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
    [INTO] tbl_name [(col_name,...)]
    {VALUES | VALUE} ({expr | DEFAULT},...),(...),...
    [ ON DUPLICATE KEY UPDATE
      col_name=expr
        [, col_name=expr] ... ]
```

Or

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
    [INTO] tbl_name
    SET col_name={expr | DEFAULT}, ...
    [ ON DUPLICATE KEY UPDATE
      col_name=expr
        [, col_name=expr] ... ]
```

Or

```
INSERT [LOW_PRIORITY | HIGH_PRIORITY] [IGNORE]
    [INTO] tbl_name [(col_name,...)]
    SELECT ...
    [ ON DUPLICATE KEY UPDATE
      col_name=expr
        [, col_name=expr] ... ]
```

After inspecting the `INSERT` statements used by YAMM, the minimal statement structure is as follows:

```
INSERT
    INTO tbl_name (col_name,...)
    VALUES (expr,...)
```

The table to which the additional rows are to be inserted is given by the `tbl_name`. The columns to which the data is added are specified by a list of comma separated `col_name` expressions. The entered values are specified by a comma separated list of `expr` items.

4.3.5 Transactional Statements

Since the number of transactional statements used by YAMM is quite small, the query structure will be tailored to fit the few statements. Therefore the required transactional statements are of two types:

1. **Single keyword** - these compromise statements containing a single keyword, for example: `BEGIN`, `COMMIT` or `ROLLBACK`
2. **Set statement** - these compromise of an assignment of a value to a variable, for example: `SET AUTOCOMMIT = 0`

4.3.6 Database Administration Statements

YAMM uses only one such statement. In particular its structure is as follows:

```
SHOW
   FULL FIELDS
   FROM tbl_name
```

This statement returns the fields from the table indicated by `tbl_name`.

4.3.7 Data Definition Statements

YAMM uses only one such statement for creating a view. Its structure is as follows:

```
CREATE
    OR REPLACE
    ALGORITHM = UNDEFINED
    VIEW view_name
    AS select_statement
```

This statement creates a view (or replaces an existing one) with the given `view_name` for a given `select_statement`. The `select_statement` is a `SELECT` statement that provides the definition of the view, it can select from base tables or from other views.

4.4 Design of Generic Query Structure

From the specification of the possible queries generated by YAMM, a hierarchy of classes can be designed to provide an object oriented structure to hold any query. The design for the query structure is given in the class diagram in Figure 4.1. The dark shaded (red) class (`Query`) represents the generic query object which can hold all SQL statements. The classes in light shade (grey) are the sub-classes which represent each of the statements described earlier.

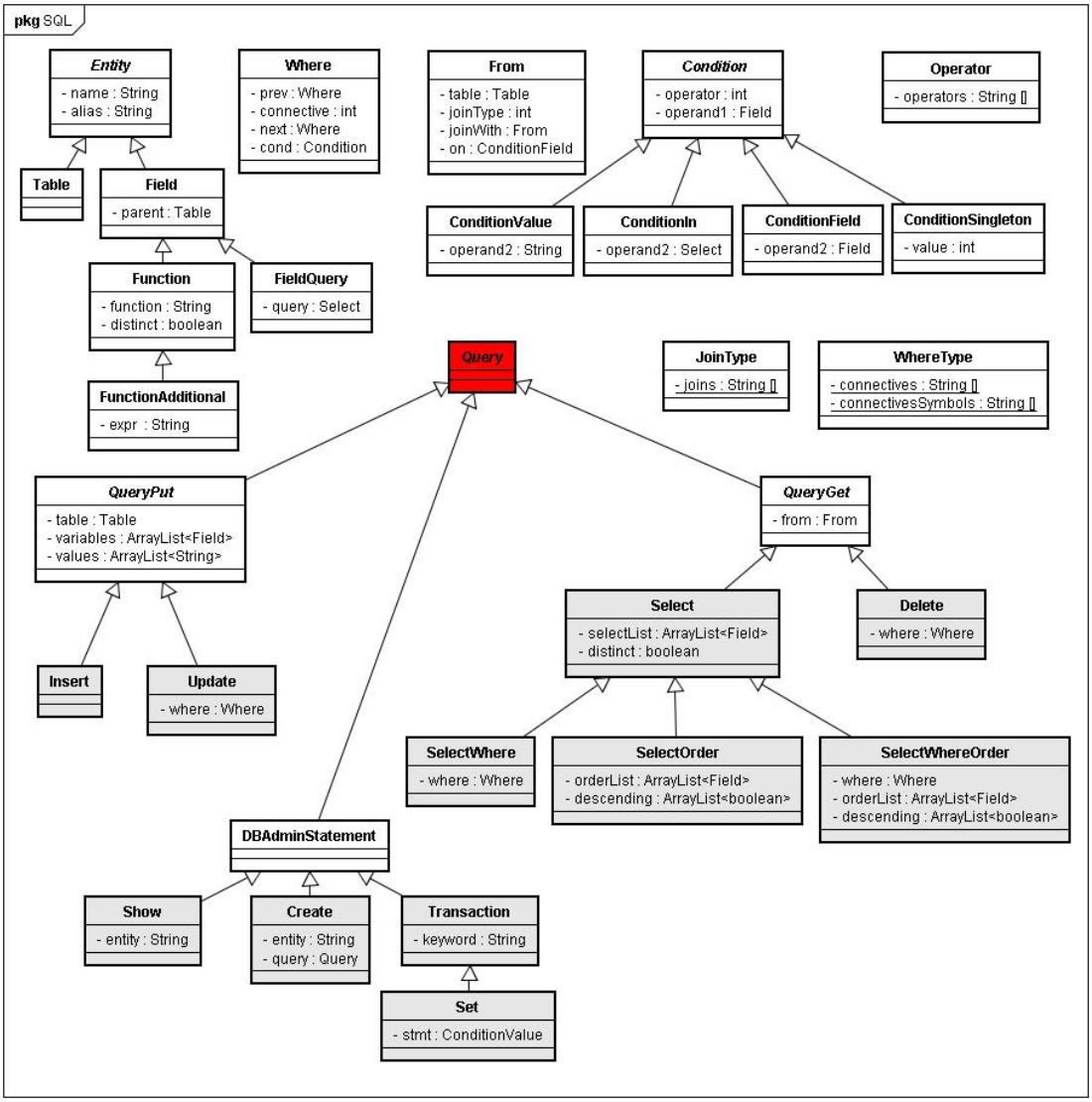

Figure 4.1: Class diagram showing design query structure

Class Entity: An `Entity` is a representation of a database object. It is an abstract super class. Every entity has a `name`, and sometimes an `alias`. If an `alias` is not given, then it is taken to be the same as the `name`.

Class Table: A `Table` is a representation of a database table. It is a subclass of the class `Entity` having its same attributes.

Class Field: A `Field` is used to model table column names which are sometimes given aliases; thus it is a subclass of the class `Entity`. Moreover the name of the table which contains this field is sometimes stated in the SQL statement; this is stored in the attribute `parent`.

Class Function: A `Function` is used to model the use of SQL functions such as `Count` and `Max`. These functions take as a parameter a column name (i.e. `Field`), in addition to the function type (as a `String`). Thus the `Function` class is subclass of the `Field` class. Moreover it is sometimes desirable to perform a count on the distinct elements of the column; thus the need arises for the uses of a `boolean` variable to indicate this.

Class AdditionalFunction: An `AdditionalFunction` is an extension of the `Function` class to include additional mathematical expressions that maybe required by the statement, such as the addition of a value `(Max(column_name) +1)`; here the addition `(+1)` is stored in a `String` named `expr`.

Class FieldQuery: It is used as a field in the selected expressions of a `SELECT` statement, defined as another `SELECT` statement. This class is a `Field`, but has an additional attribute to hold the nested query.

Class Where: It represents the `WHERE` clause of any SQL statement. According to the previously mentioned specification, it should support the ability to have conjunction `(AND)` and disjunction `(OR)` of conditional clauses. Thus it consists of either a `Condition`, or two other `Where` clauses connected by an `connective` (represented as an `integer`, and decoded by the `WhereType` class).

Class From: It represents the `FROM` clause of any `SELECT` or `DELETE` statement; it should support the possibility to add multiple consecutive joins. This is done by containing a recursive call of the `From` clause. Thus a `From` object is either a target `Table`, or another `From`, with a `Condition` on which they are joined and a specific `integer` representation of a `join type` (decoded by the `JoinType` class).

Classes Condition, ConditionField, ConditionValue, ConditionIn, ConditionSingelton: These hierarchy of classes are used to represent the `conditional_expr` clause. All

conditional clauses contain one operator and two operands. The first operand is always a `Field`, however the second operand is either a `Field`, a `value` or a set of columns as result of a `SELECT` statement to be used with the `IN` operator. Thus the need arose for three respective classes `ConditionField`, `ConditionValue` and `ConditionIn`, all extending a super class named `Condition`. Another subclass of `Condition` is the `SingletonCondition` class, which just contains a `value` evaluating to `0` or `1`. The `operator` attribute of the `Condition` class is an `integer` decoded by the `Operator` class.

Class Query: `Query` is a super class used as a structure for all SQL statements. All SQL statement classes extend this class. They can be divided into three types of statements: `QueryPut`, `QueryGet` and `DBAdminStatement`. The grouping of statements into these three classes was based on the grouping of their characteristics.

Class QueryPut: This class represents query statements that "put" data into the database. Objects of this type all have a destination `Table` to which the data is added, a list of the column headings (`variables`) to which the data is inserted, and the actual data (`values`) that will be inserted. Two classes extend this class: `Insert` and `Update`. Since an `UPDATE` statement applies changes to a particular set of rows, it requires the need for a `Where` attribute.

Class QueryGet: It represents query statements that "get" data from the database. Objects of this type must have a reference to the destination from which the data is obtained. This can be a table or multiple tables joined together; thus this class has a `From` attribute. Many classes extend this class; each adding attributes according to the nature of the statement, such classes are: `Select`, `SelectWhere`, `SelectOrder`, `SelectWhereOrder` and `Delete`. Notice that the `Select` class has a `boolean` attribute to represent if the statement contains a distinct keyword. Also some classes contain an array of `boolean` flags for the `OrderBy` columns to represent the desire of an ascending or descending arrangement for each column.

Class DBAdminStatement: This class is used as a super class to represent all the other statements: transactional, administrative and data definition statements.

Class Show: This class contains a `string` attribute to represent whatever is to be shown.

Class Create: This class contains a `String` attribute to represent the view name that is to be created and a `SELECT` statement as a means of holding the SQL statement to be executed for creating the view.

Class Transaction: This class contains a `String` attribute to represent the transaction

keyword.

Class Set: This class extends the `Transaction` class, where all objects here will have the `keyword` as "`SET`" and a `ConditionValue` attribute to contain the assignment variable and value.

4.5 Interoperability Layer

The generic query structure is implemented in Java. Java is a good choice of programming language since it is platform independent and it supports various database connections [11]. However, the interoperability layer needs a bridge to access Java classes from YAMM's PHP scripts.

4.5.1 Java/PHP Bridge

One available bridging solution is the use of the PHP/Java bridge [32]. This bridge is an implementation of a streaming XML-based network protocol. It is used as a connection mechanism between a native script engine, like PHP, with a Java virtual machine. It is a fast solution since it needs less resources on the web-server side compared to using Remote Procedure Call (RPC) via the Simple Object Access Protocol (SOAP); in fact its more than 50 times faster than RPC. Moreover it necessitates no additional components to invoke Java procedures from PHP or PHP procedures from Java, thus the bridge is faster and more reliable than direct communication via the Java Native Interface.

The bridge contains a PHP "Java" class implementation which utilises a VM bridge protocol, in order to connect running PHP instances with already running Java or .NET back ends. It is a two-way communication; Java components can call PHP instances using an interface that can connect to a running PHP server, and also PHP scripts can invoke Java based applications.

The PHP/Java Bridge is available as a standard Java EE web application, `JavaBridge.war`. It can be deployed to any standard Java servlet engine or Java application server. It is a distributable ZIP archive, containing example PHP scripts, the `JavaBridge.jar` Java library, and a MIT-licensed PHP file which may be included by PHP scripts.

The PHP file `Java.inc` is available from the web application's `java/` folder. This file is of high interest to the aim of our work, since PHP scripts can fetch it and use it to invoke Java methods using familiar PHP syntax. The fetching is done using the `require_once`

command. The interoperability layer depends on the `Java.inc` file, in order to create Java objects out of the generic query structure and to connect to the various database back-ends.

4.5.2 Design

The design of the interoperability layer and how the different components connect together is shown in Figure 4.2.

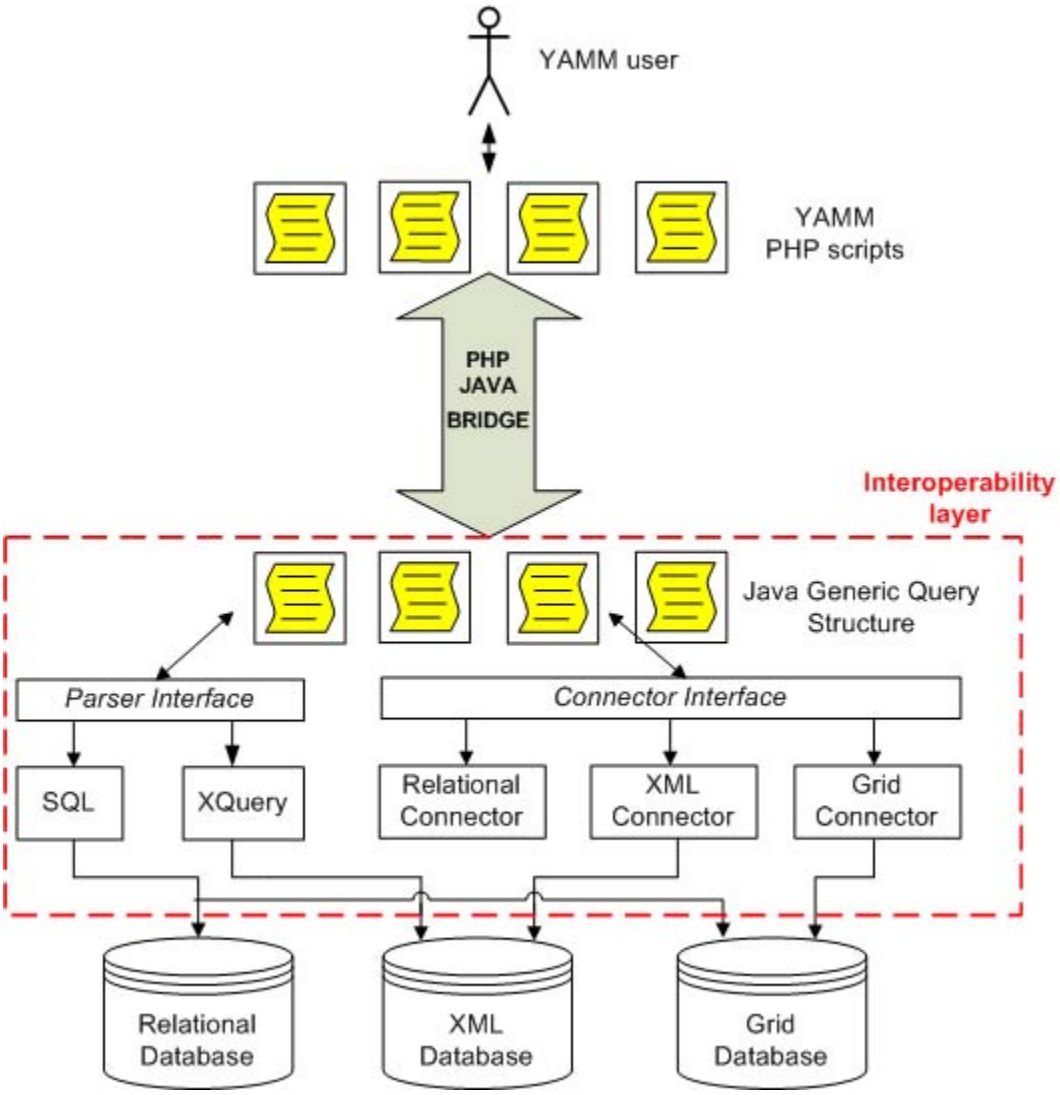

Figure 4.2: Connection of components of new system architecture

4.5.3 Implementation

The generic query structure, the generic connection interface, and the actual back-end connections are implemented in Java. YAMM's PHP scripts are modified to use the PHP/Java bridge in order to invoke Java methods and create Java query objects. These objects are passed to the connection interface and parsed into whatever querying language supported by the intended database back-end.

In order for the PHP scripts to be able to use the Java classes, the following commands have to be included in every script:

```
require_once("java/Java.inc");
//this is the location of the java classes
java_require("C:\.......\classes");
```

Additionally, the PHP/Java Bridge requires that a Tomcat server be running which has the `JavaBridge.war` deployed into it.

The introduced modifications performed in the PHP scripts to use the Java query objects will be highlighted in the following subsections.

4.5.4 Query Modifications

In this section several examples are given of how the modifications were performed in YAMM; in order to introduce the parsing of queries into the generic structure.

4.5.4.1 Query 1: SELECT WHERE

For a simple SQL `SELECT` statement that selects the `ID` from the `record` table given a single condition in the `WHERE` clause that checks the equivalence of the `EntityID` field to a PHP variable (`$entityID`), such as:

```
YDB::query("SELECT ID FROM record WHERE EntityID =".$entityID)
```

It should be changed into the following PHP script, which creates an `ArrayList` of desired fields for selection, a `From` clause containing the target table of selection upon which the query acts and a `Where` clause containing the condition having a value as a second operand:

```
$selectList = new Java("java.util.ArrayList");
$selectList->add(new Java("Field","ID"));
```

```
$from = new Java("From",new Java("Table","user"));
$whereClause = new Java("Where", new Java("ConditionValue","=",
                            new Java("Field","EntityID"),$entityID));
$metaQuery = new Java("SelectWhere",$from,$selectList,$whereClause);
YDB::query($metaQuery);
```

If it is required to specify that only unique `IDs` are extracted then a `DISTINCT` keyword is essential; this is specified by inserting a `boolean true` value as a last variable in the `$metaQuery` as seen below:

```
$metaQuery = new Java ("SelectWhere",$from,$selectList,$whereClause,true);
```

4.5.4.2 Query 2: SELECT ORDER BY

For a simple SQL `SELECT` statement that selects all fields from the `entity` table and orders them according to the `(Name)`:

```
YDB::query("SELECT * FROM entity ORDER BY Name")
```

It should be changed into the following PHP script:

```
$selectList = new Java("java.util.ArrayList");
$selectList->add(new Java("Field","*"));
$from = new Java("From",new Java("Table","entity"));
$orderList = new Java("java.util.ArrayList");
$orderList->add(new Java("Field","Name"));
$metaQuery = new Java("SelectOrder",$from,$selectList,$orderList);
YDB::query($metaQuery);
```

If it is desired to sort the fields in descending order of the name, then an array of descending flags corresponding to the fields in the order list is used; the `$metaQuery` is replaced by the following:

```
$orderList = new Java("java.util.ArrayList");
$orderList->add(new Java("Field","Name"));
$descending = new Java("java.util.ArrayList");
$descending->add(new Java("Boolean",true));
$metaQuery = new Java("SelectOrder",$from,$selectList,$orderList,
                        $descending);
YDB::query($metaQuery);
```

4.5.4.3 Query 3: SELECT WHERE ORDER BY

A complex SQL statement containing a complex `WHERE` clause consisting of a join of three tables is as follows:

```
YDB::query("SELECT attribute.*, attribute.Name AS AName, type.Name AS TName,
          type.ValueField, entity.SuperEntityID, entity.Name AS EntityName
          FROM attribute JOIN type ON attribute.TypeID=type.ID JOIN entity
          ON attribute.EntityID=entity.ID WHERE attribute.ID=".$attributeID
          "ORDER BY attribute.OrderShow");
```

To construct the nested `FROM` clause, it is built backwards with the last table first, followed by a backward nesting of tables. In other words, the `entity` table is joined to the `attribute` table, then to the `type` table. The PHP script to transform this SQL statement into the generic Java query is as follows:

```
$selectList = new Java("java.util.ArrayList");
$selectList->add(new Java("Field","*", new Java("Table","attribute")));
$selectList->add(new Java("Field","Name", new Java("Table","attribute"),
                          "AName"));
$selectList->add(new Java("Field","Name", new Java("Table","type"),
                          "TName"));
$selectList->add(new Java("Field","ValueField", new Java("Table",
                                                              "type")));
$selectList->add(new Java("Field","SuperEntityID", new Java("Table",
                                                              "entity")));
$selectList->add(new Java("Field","Name", new Java("Table","entity"),
                          "EntityName"));
$from3 = new Java("From",new Java("Table","entity"));
$from2 = new Java("From",new Java("Table","type"),"JOIN",$from3,
          new Java("ConditionField","=",
           new Java("Field","EntityID",new Java("Table","attribute")),
             new Java("Field","ID", new Java("Table", "entity"))));
$from = new Java("From",new Java("Table","attribute"),"JOIN",$from2,
           new Java("ConditionField","=",
             new Java("Field","TypeID", new Java("Table", "attribute")),
               new Java("Field","ID", new Java("Table", "type"))));
$whereClause = new Java("Where",new Java("ConditionValue","=",
```

```
                    new Java("Field","ID",
                      new Java("Table", "attribute")), $attributeID));
$orderList = new Java("java.util.ArrayList");
$orderList->add(new Java("Field","OrderShow",
                          new Java("Table", "attribute")));
$metaQuery = new Java("SelectWhereOrder",$from,$selectList,
                       $whereClause, $orderList);
YDB::query($metaQuery);
```

4.5.4.4 Query 4: SELECT - with Aggregate Functions

An example of an SQL statement that contains the usage of the `COUNT` and `MAX` functions is as follows:

```
YDB::query("SELECT COUNT(*), MAX(ValueInt)+1 FROM value
            WHERE value.AttributeID=" . $attributeID)
```

Transforming this into the Java object invocations means creating a `Function` object for the `COUNT` and an `AdditionalFunction` for the `MAX` (to hold the additional tailing expression). This can be done by inserting the following PHP script:

```
$selectList = new Java("java.util.ArrayList");
$selectList->add(new Java("Function","*","COUNT"));
$selectList->add(new Java("FunctionAdditional","ValueInt","MAX","+1"));
$from = new Java("From",new Java("Table","value"));
$whereClause = new Java("Where", new Java("ConditionValue","=",
            new Java ("Field","AttributeID",new Java("Table","value")),
                       $attributeID));
$metaQuery = new Java("SelectWhere", $from,$selectList,$whereClause);
YDB::query($metaQuery);
```

4.5.4.5 Query 5: SELECT with Nested Query

Some of the queries involve nested queries; in particular they contain a query in the `WHERE` clause and require the use of the `IN` operator. This is illustrated in the following `SELECT` statement:

```
YDB::execute("SELECT * FROM value WHERE AttributeID IN
```

```
(SELECT ID FROM attribute WHERE TypeID=3)")
```

A special type of conditional clause was designed especially to handle this type of nested queries: the `ConditionIn` object. This object contains a reference to the variable and the nested statement; i.e. the `AttributeID` variable and the `SELECT` statement. The translation to use the Java objects from PHP is performed by creating a `subMetaQuery` to hold the nested query and then this is given as a parameter to the `ConditionIn` object of the `$whereClause`. The means for this are shown below:

```
$subSelectList = new Java("java.util.ArrayList");
$subSelectList->add(new Java("Field","ID"));
$subFrom = new Java("From",new Java("Table","attribute"));
$subWhereClause = new Java("Where",new Java("ConditionValue","=",
                      new Java("Field","TypeID"),$vars["type"]));
$subMetaQuery = new Java ("SelectWhere",$subFrom,$subSelectList,
                          $subWhereClause);
$destTable = new Java("Table","value");
$whereClause = new Java("Where",new Java("ConditionIn",
                    new Java("Field","AttributeID"),$subMetaQuery));
$metaQuery = new Java("Delete",$destTable,$whereClause);
YDB::execute($metaQuery);
```

4.5.4.6 Query 6: INSERT

An SQL statement that includes inserting records into the `Filter` table of the YAMM database has the following format:

```
YDB::query("INSERT INTO Filter(AttributeID, FAttributeID, CAttributeID)
            VALUES (".$vars["aID"]."," ".$vars["faID"]."," ".$caID)
```

This can be transformed into a Java object by specifying the `ArrayList`s of column names and values; as in the following PHP script:

```
$destTable = new Java("Table","Filter");
$variables = new Java("java.util.ArrayList");
$variables->add(new Java("Field","AttributeID"));
$variables->add(new Java("Field","FAttributeID"));
$variables->add(new Java("Field","CAttributeID"));
```

```
$values = new Java("java.util.ArrayList");
$values->add(new Java("java.lang.String",$vars["aID"]));
$values->add(new Java("java.lang.String",$vars["fAID"]));
$values->add(new Java("java.lang.String",$caID));
$metaQuery = new Java("Insert",$destTable,$variables,$values);
YDB::query($metaQuery);
```

4.5.4.7 Query 7: UPDATE

An `UPDATE` SQL statement which is capable of altering rows in the `Institution` table of the YAMM database is given below:

```
YDB::query("UPDATE Institution SET Name='".$vars["name"].
                    "' WHERE ID=".$vars["institution"]);
```

The introduced modifications on the PHP side should be:

```
$destTable = new Java("Table","Institution");
$variables = new Java("java.util.ArrayList");
$variables->add(new Java("Field","Name"));
$values = new Java("java.util.ArrayList");
$values->add(new Java("java.lang.String","'".$vars["name"]."'"));
$whereClause = new Java("Where",new Java("ConditionValue","=",
                        new Java("Field","ID"),$vars["institution"]));
$metaQuery = new Java("Insert",$destTable,$variables,$values);
YDB::query($metaQuery);
```

4.5.4.8 Query 8: DELETE

An SQL statement that deletes a record specified by the PHP variable `$relationID` from the database table `relation` has the following format:

```
YDB::execute("DELETE FROM relation WHERE ID=".$relationID);
```

This can be transformed into the following PHP script:

```
$destTable = new Java("Table","relation");
$whereClause = new Java("Where",new Java("ConditionValue","=",
                                  new Java("Field","ID"),$relationID));
```

```
$metaQuery = new Java("Delete",$destTable,$whereClause);
YDB::execute($metaQuery);
```

Furthermore an SQL statement that deletes all records from the `relation` table, is as follows:

```
YDB::execute("DELETE FROM relation");
```

Since it has no `WHERE` clause, then it is transformed into a `Delete` object as follows:

```
$destTable = new Java("Table","relation");
$metaQuery = new Java("Delete",$destTable);
YDB::execute($metaQuery);
```

4.5.4.9 Query 9: Transactional

The few transactional statements can be easily transformed to use the generic query structure. Given below is the direct mapping for each of the possible statements:

`YDB::execute("BEGIN")` → `YDB::execute(new Java("Transaction","BEGIN"))`

`YDB::execute("COMMIT")` → `YDB::execute(new Java("Transaction","COMMIT"))`

`YDB::execute("ROLLBACK")` → `YDB::execute(new Java("Transaction","ROLLBACK"))`

`YDB::execute("SET AUTOCOMMIT=0")` → `YDB::execute(new Java("Set","0"))`

4.5.4.10 Query 10: Data Definition

The only data definition statement used by YAMM creates a view with a specific name (as in the variable `$name`) for a given select statement(`$sql`). Formally it was given as:

```
YDB::execute("CREATE OR REPLACE ALGORITHM = UNDEFINED VIEW `".$name."`
             AS ".$sql)
```

The new PHP script for this is as follows:

```
$metaQuery = new Java("Create",$name,$sql)
YDB::execute($metaQuery)
```

In order to introduce the interoperability layer, all YAMM queries had to be transformed as explained in the 10 examples presented above. Some queries have a fixed structure as the ones discussed; their mapping is a direct one following the transformation rules illustrated.

Other queries are dynamic in their creation; they depend on the values of certain variables, like those queries needed for a search query. The transformation of these queries is more complex, where special care has to be taken for the careful translation of the dynamic construction means of these queries. However in principle the translation rules are the same, since the generic query structure is fixed.

4.5.5 Connection Modifications

The old version of YAMM used the built-in functions of PHP in order to connect to a MySQL database. Therefore to introduce the generic query structure, it was required to change the database connections to use generic back-end connection functions implemented in the interoperability layer. The connection implementation should contain a connection mechanism such as the use of the Java Database Connectivity (JDBC) driver. The changes were made to the PHP script `YDB.php`, some of these changes are given in the following points.

- To facilitate the connections to the interoperability layer, a class variable named `$dbConnector`, is needed in the `YDB` script. Calls to any Java methods handling database back-end operations are performed using this variable.
- To connect to the database back-end, the `YDB connect` function simply calls the interoperability `connect` method. Given below is the means in order to connect to a relational back-end while instantiating a JDBC connection from the connector variable, (this connection is explained in Section 4.6.2):

```
function connect() {
   YDB::$dbConnector = new Java("JDBConnector");
   YDB::$dbConnector->connect();
}
```

- To execute a query, the `YDB execute` function simply calls the interoperability `execute` method and it returns the number of affected rows if the query was an update; this number should be returned by definition from the Java method. Given below is the newly implemented `execute` method:

```
function execute($metaQuery) {
   return $dbConnector->execute($metaQuery);
}
```

- Another necessary method is the `fetch_assoc` method, which is a method that fetches a result row from an executed query's result set and returns it as an associative array. An *associative array* is an array where any key or index is "associated" with each value. In this case, the row values are the values of the associative array, and the keys are the result set columns names from the returned query. The means of how this associative array data is extracted from the query result is the concern of the interoperability layer. However, the `YDB` script should assemble the associative array from the returned Java arrays. The Java/PHP bridge offers a method to evaluate a Java object; it is called `java_values()`. This method converts a Java object into an equivalent PHP value. Thus here it can be used to parse the Java arrays returned from the interoperability layer into PHP arrays, to be able to use their values and assemble a PHP associative array. The means by which this is done is shown below:

```
function fetch_assoc() {
   $rowHeader = java_values($db->getHeader());// getting column names
   $rowValues = java_values($db->fetchRow()); // getting row values
   if($rowValues == null)
      return null;

   $db->skipRow();   // moving row pointer to the next row to evaluate
   $data = array();  // a PHP associative array variable
   $i = 0;
   foreach($rowValues as $key => $value) {
      $data[$rowHeader[$i]] = $value;
      $i = $i +1;
   }
   return $data;
}
```

4.6 Interface for Various Database Back-ends

The main reason for introducing the Java interoperability layer is to have the possibility to use various database back-ends. This is dependant on the fact that there exists one generic structure for storing all queries and database interactions. A query translator is needed to produce queries specific to each back-end and to translate the results back from each data source into a common format.

The required functionality was split into two interfaces that act as a prototype for all the database interactions: a *parser interface* and a *connector interface*. Each interface sets an abstract requirement for the functionalities that it represents; being an interface it allows one to specify zero or more method signatures without providing the implementation of those methods. Database specific classes should then implement the common interfaces, thereby providing the relevant coding of the methods performing the intended functionality.

4.6.1 Parser Interface

The parser interface is used to parse the generic query structure into a querying language as specified by the respective database back-end. The interface specifies the essential methods needed to represent the components of the query structure. Methods are thus needed to parse each of the classes explained earlier. The interface written in Java is as follows:

```
interface QueryParser {
   // parses a SELECT statement
   String parseSelect(Query q);

   // parses a DELETE statement
   String parseDelete(Query q);

   // parses an INSERT statement
   String parseInsert(Query q);

   // parses an UPDATE statement
   String parseUpdate(Query q);

   // parses a Database Administrative statement
   String parseDBAdmin(Query q);

   // parses a FROM clause
   String parseFrom(From from);

   // parses a Conditional expression
   String parseCondition(Condition cond);
```

```
    // parses a WHERE clause
    String parseWhere(Where where);

    // parses a Field, Table, Function or Additional function
    String parseEntity(Entity e);
}
```

Querying languages are computer languages used to query databases and information systems. Depending on the target database system, the respective query language is chosen. For example, SQL is a well known query language used for relational database management systems [19]. In the frame of this work, a parser was made to transform the query structure into SQL syntax. This facilitates the use of a relational database such as a MySQL or Microsoft SQL database. A class named `SQLParser` was created that implements each of the methods specified in the common interface, to produce well structured SQL syntax.

To illustrate how this was performed and to explain the means in which the interface can be implemented for any other querying language, a few of the method implementations are given next.

SQL INSERT Parser

```
public String parseInsert(Query q) {
    Insert insert = (Insert) q;
    String str = "INSERT INTO \t";
    str += parseEntity(insert.getTable()) + " ("
         + parseFieldList(insert.getVariables()) +") \n"
         + "VALUES \t("
         + parseStringList(insert.getValues()) +")";
    return str;
}
```

This method returns a String representation of the `INSERT` query. It uses helper methods which were implemented to parse the components of the structure:

- `parseEntity(Entity e)`: to parse the name of the target table.
- `parseFieldList(ArrayList<Field> list)`: to parse a list of fields, which generates a comma separated listing of their names.

- `parseStringList(ArrayList<String> values)`: to parse a list of string values.

SQL SELECT Parser

The parser for a `SELECT` statement has to determine the extended type of the statement, to determine if it has any `WHERE` and/or `ORDER BY` clauses to handle them respectively. Moreover a check has to be performed on whether it is a `DISTINCT` selection query.

```
public String parseSelect(Query q) {
      Select select = (Select) q;
      String str;
      if (select.getDistinct())
         str = "SELECT DISTINCT \t";
      else
         str = "SELECT \t"
      str += parseFieldList(select.getSelectList()) + "\n"
          +  parseFrom(select.getFrom()) + "\n";
      if(select instanceof SelectWhere)
         str += parseWhere(((SelectWhere)select).getWhere()) + "\n";
      if(select instanceof SelectWhereOrder)
         str += parseWhere(((SelectWhereOrder)select).getWhere()) + "\n"
             + "ORDER BY \t"
             + parseOrderList(((SelectWhereOrder)select).getOrderList(),
                    ((SelectWhereOrder)select).getDescending())+ "\n";
      if(select instanceof SelectOrder)
         str += "ORDER BY \t"
             +   parseOrderList(((SelectOrder)select).getOrderList(),
                     ((SelectOrder)select).getDescending())+ "\n";
      return str;
}
```

The need rises for the use of a helper method to parse the order list (if given) in a way to facilitate the output of the desired sort orientation, i.e. descending or ascending. This is the `parseOrderList` method and its implementation is as given below:

```
protected String parseOrderList(ArrayList<Field> list,
                                ArrayList<Boolean> desc){
```

```
    String str = "";
    for(int i=0; i<list.size(); i++ ){
       str += parseFieldName(list.get(i));
       if(desc.get(i)) // placing descending orientation if desired
          str += " DESC";
       str += ", ";
    }
    str = str.substring(0, str.length()-2); // removing extra comma
    return str;
}
```

4.6.2 Connector Interface

The connector interface is used to connect the interoperability layer to the desired database. It specifies the methods that should be implemented for any database connection; it indicates the means to connect to the database, to retrieve data from it and operate on the obtained data. Any database connection class implements this connector, to ensure that it contains the methods required by the newly implemented YAMM interoperability layer. The interface written in Java is as follows:

```
interface Connector{
  // connects to the target database
  public void connect();

   // closes the connection with the target database
   public void close();

   // queries the database, returning a ResultSet with retrieved data
   public ResultSet query(Query query);

   // executes a query, returning number of affected rows if query
   // was an UPDATE
   public int execute(Query query);

   // returns a string array containing single record of the result set
   public String[ ] fetchRow();
```

```
    // returns the number of rows obtained in the result set
    public int numRows();

    // returns the ID generated for an AUTO_INCREMENT column by
    // the previous INSERT query
    public int getInsertId();

    // releases the last query and remove it from the array
    public void removeLastQuery();

    // escapes special characters in a string for safe use in the
    // querying statement
    public String makeSecure(String inputString);

    // returns the last query in array list
    public ResultSet currentQuery();
}
```

In the frame of this work, a connector class was implemented using JDBC to connect the interoperability layer to a MySQL database. JDBC is a Java API that enables the execution of SQL statements by Java programs, to interact with any SQL compliant database. The power of using JBDC lies in that most relational database management systems support SQL and that Java is platform-independent running on most platforms; this makes it possible to write a single database application that can run on different platforms and interact with different database systems.

A class named `JDBConnector`that implements the `Connector` interface was created. This class uses the JDBC driver to connect to the YAMM database. It contains implementations for each of the interface methods. Furthermore it contains an array (`ArrayList<ResultSet> resultSets`)) to store all the result sets and to keep store of them until they are needed or removed as required by the implementation. Another required class variable is an array (`String[ ] header`); it keeps account of the field headers of the last executed query.

The implementations for methods is quite straight forward, however it may seem important to highlight the implementations of the following methods:

JDBC connection - Connect method

In order to connect to a MySQL database using the JDBC driver, the connect method is as follows (note that the database password and user name are both "`YAMM`"):

```
public void connect(){
   resultSets = new ArrayList<ResultSet>();   // initialising result sets
   String url="jdbc:mysql://localhost/yamm"             // connection URL
   try {
      Class.forName("com.mysql.jdbc.Driver").newInstance();
      conn = DriverManager.getConnection(url,"yamm", "yamm"); // providing
   }                                          // the user name and password
   catch (SQLException E) {
      System.out.println("SQLException: " + E.getMessage());     }
}
```

JDBC connection - Querying method

```
public ResultSet query(Query query){
   rs = null;
   try{
      Statement stmt = conn.createStatement();
      rs = stmt.executeQuery(converter.parse(query));
      addResultSet(rs);               // to add rs to global array list
      if(rs!== null)
         setHeader();                 // call to store the column names
   }
   catch (SQLException E) {
      System.out.println("SQLException: " + E.getMessage());     }
   return rs;
}
```

The method requires the assistance of two helper methods:

- `addResultSet(ResultSet rs)`: adds the given result rest to the stored ones.
- `setHeader()`: sets the header string array to store the column names of the currently executed query.

JDBC connection - Securing method

Securing the SQL statements parsed before sending them to the database connector is a very important step in order to avoid SQL injection. SQL injection is a technique which exploits a security vulnerability occurring in the database layer of the application [2]. The risk is present when queries contain data taken as input from the user; input may not be strongly typed or certain characters may need to be present as escape sequence.

This functionality was performed in the earlier version of YAMM using the PHP function `mysql_real_escape_string()`; which calls a MySQL library function which prepends backslashes to the following characters: `\x00`, `\n`, `\r`, `\`, `'`, `"` and `\x1a` [31].

Thus it is required that the `makeSecure(String input)` method scans the string for the occurrence of the specified characters and prepends them with a leading backslash.

JDBC connection - Number of Rows method

In order to count the number of rows returned in the result set of the last executed query, this is performed using the following code snippet:

```
public int numRows(){
  try{
    int rowCount;
    int currentRow = rs.getRow();        // Get current row
    rowCount = rs.last()? rs.getRow():0; // Get number of rows
    if (currentRow == 0)                 // If there was no current row
       rs.beforeFirst();                 // next() should go to 1st row
    else                                 // If there was a current row
       rs.absolute(currentRow);          // Restore it
    return rowCount;
 }
 catch (SQLException E) {
    System.out.println("SQLException: " + E.getMessage());     }
 return -1;
}
```

4.7 Summary

This chapter included the main tasks performed in order to introduce the interoperability layer into YAMM. It started by examining the existing YAMM implementation and extracting the main back-end operations. Then the different forms of the various back-end operations were specified. The main highlight of the chapter was the design of a generic query structure that can be used to wrap any YAMM back-end operation. Following this, the main modifications performed in YAMM were summarised. In the end, a discussion of the interoperability interface was given, which provided an opportunity to have multiple back-end implementations. In this work, a relational back-end using a JDBC connection and a SQL parser were implemented. After inspecting the OGSA-DAI and AMGA Grid toolkits and verifying their adaptability with the newly introduced interoperability layer, it was concluded that using a Grid-Aware back-end is feasible. However due to the time frame of this work, this interface remains as a future task.

Chapter 5

Instantiation with the SEE-KID Metamodel

In order to use YAMM for any application, the application's data model must be transformed into the type accepted by YAMM. The data model and the data itself are then provided to YAMM as input using its graphical interface, as was illustrated in Section 2.2.2. At the time of designing the interoperability layer, an attempt was made by RISC/MI to automate the instantiation process. Using the provided automated input means, we discovered a problem with the current implementation of YAMM. Thus the need has emerged to re-engineer the structure of YAMM.

This chapter starts with a brief overview of the work done to automate the instantiation process. It follows by a presentation of the data model used in the SEE-KID project. A discussion of how this model can be transformed into one that is acceptable by YAMM is then elaborated. From this point the flaw in YAMM is revealed, and the chapter concludes by suggesting how this problem can be resolved.

5.1 Automating the Metamodel Instantiation

It is a requirement of YAMM for any data model which it encodes, that it consists of one root entity for which other entities and attributes are related to. Thus it is possible to give a nested XML structure to represent this model. Using such a structure would ease the import of any data model into YAMM, in addition to making the process of exporting the data or the model stored in a transmittable form easy to produce.

To illustrate the requirements for such a document, the example previously mentioned in Section 2.2.2 (Figures 2.4 and 2.5) can be revisited. This example contains car ownership data. The root element which is needed in its YAMM representation is the *Person* entity, since a person can contain the *owns* relation to a *car*. The data can be represented as the following XML document:

```
<person Person_ID="1234" Name="Jane Smith" Date_of_birth="9/1/85"
        Gender="Female">
    <owns>
        <car Car_ID="5512" type="Volvo" color="red" />
    </owns>
</person>
```

Its metamodel can be expressed using the following XML Schema Document (XSD):

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="Person">
   <xsd:complexType>
     <xsd:attribute name="Person_ID" type="xsd:integer" use="required"/>
     <xsd:attribute name="Name" type="xsd:string" use="required"/>
     <xsd:attribute name="Date_of_birth" type="xsd:date" use="required"/>
     <xsd:attribute name="Gender" type="xsd:string" use="required"/>
     <xsd:sequence>
      <xsd:element name="Owns">
       <xsd:complexType>
         <xsd:sequence> <xsd:element name="Car" type="CarType"/> </xsd:sequence>
       </xsd:complexType>
      </xsd:element>
     </xsd:sequence>
   </xsd:complexType>
 </xsd:element>
 <xsd:complexType name= "CarType">
   <xsd:attribute name="Car_ID" type="xsd:integer"/>
   <xsd:attribute name="type" type="xsd:string"/>
   <xsd:attribute name="color" type="xsd:string"/>
 </xsd:complexType>
</xsd:schema>
```

Since any data model can be expressed in a format similar to the one presented, having YAMM interpret and produce such structured documents is an advantage for future extensions of YAMM. At the time or production of this work, an importer and exporter was created for handling such documents to and from YAMM.

The importer works by taking a structured document as the one above and transforming it into model entries in YAMM. It is capable of extracting the data model characteristics and placing them in the respective entities for storage.

5.2 SEE-KID Data Model

The complex SEE-KID data model initially created in [29], represents the entities needed to store information regarding a patient and his/her simulation data. The diagram in Figure 5.1 shows the transport model of SEE-GRID.

The root element for which the data is stored is the *SoapPatient* entity. This is the SOAP representation of the most important element for which the whole SEE-KID simulations revolve around. The blocks represent the entities involved, and the arrows represent the relations between them. An arrow with a single name written on it is a one-to-one relationship, however an arrow with multiple names represents a one-to-many relationship.

To illustrate how this model is read, a discussion follows of three entities, their relations and attributes. A patient undergoing an eye simulation will require simulation data to be stored. The *SoapSimulation* is an entity that contains the simulation data of three eye data sets: *left, right* and *reference* eyes; each of these is a *SoapEyeData*. Every *SoapEyeData* has one *modelType* which is given by a *ModelTypeId* entity containing the respective model data, represented with the attributes: *SoapStringModel, SoapTapeModel, SoapSeeKidModel, SoapSeeKidActivePulleyModel* and *SoapOrbitModel*. It also has a relation to geometrical data with the *SoapGeometricalData* entity. This entity contains the attributes: *eyeX, eyeY, eyeZ, referenceDirection, gazeRotationVector, maxAngle* and *gazeQuaternion*.

5.3 Transforming into YAMM's Metamodel

SEE-KID's model can be easily transformed into YAMM's data model, by applying the automated mapping technique to an XML structured representation of the SEE-KID model.

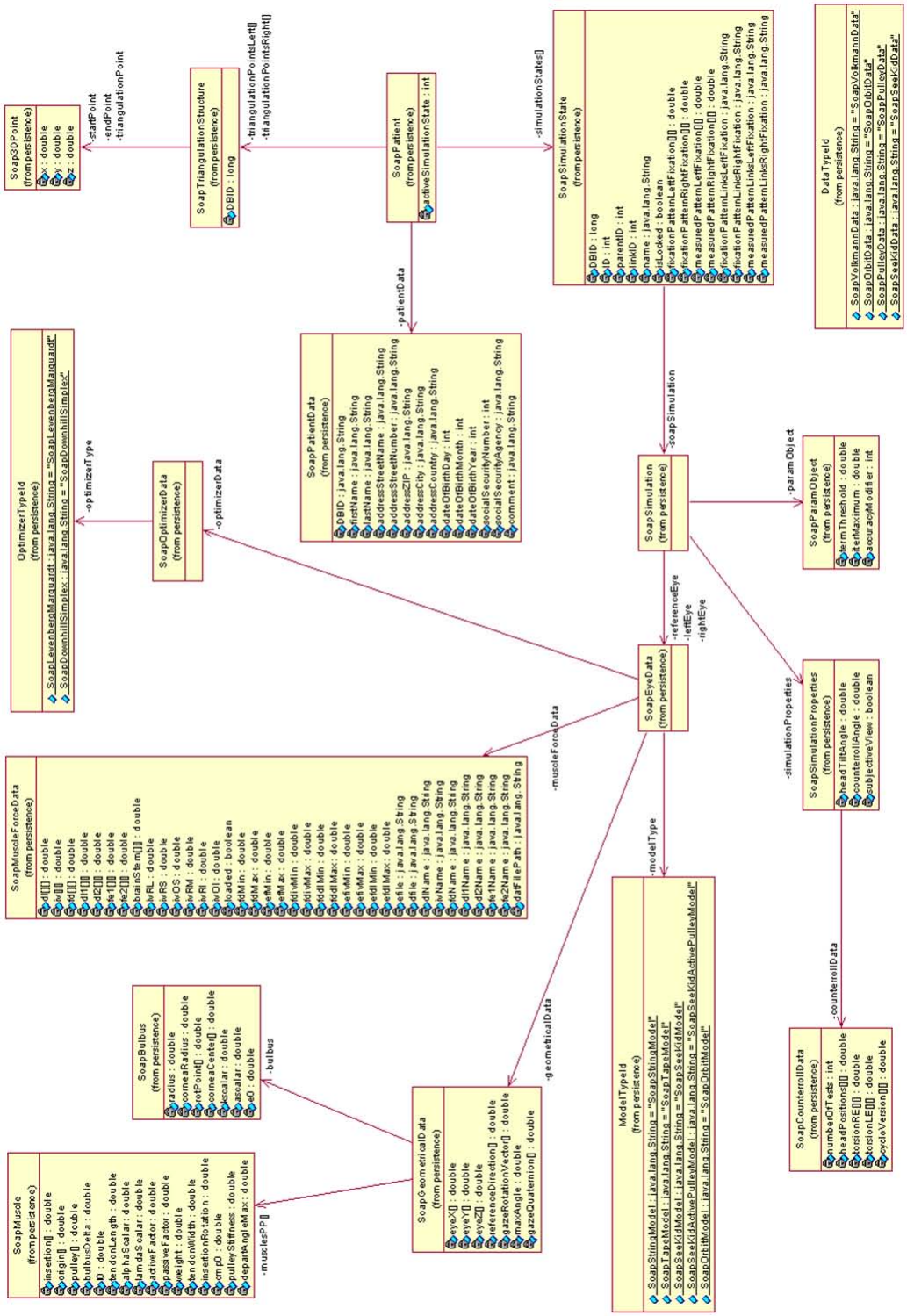

Figure 5.1: SEE-GRID Transport Model [25]

In fact, SEE-KID already uses gSOAP [1] to produce two exports of its transport data model. One is in the form of a WSDL (Web Services Description Language) document; containing the data types as SOAP methods. Another output in the form of an XSD (XML Schema Document); containing the data types and additional SOAP methods. The old SEE-KID persistence component uses the WSDL document. However for the purposes of YAMM, it would be easier to use the simpler XSD and just remove additional unnecessary data. Through this automated means, the metamodel can be easily instantiated and imported into YAMM. The only step needed is to make sure that all the data represented in the XSD can be reflected in YAMM, without any loss of information.

5.4 Problems with YAMM

During the inspection of the data model after parsing it into one interpretable by YAMM, we noticed that some vital information is missing in relation to the required relations. This information is lost when storing the model into YAMM, while this information is highly valuable for the needs of SEE-KID.

In particular, the problem was experienced in the translation of the *Eye Data* entity, since the Soap Simulation entity has three Eye data entities: *left, right* and *reference* eye data. In YAMM entities are stored using backward referencing; thus each eye data entry will have a reference to an eye simulation entry. In order to obtain the eye data for a particular simulation a join query must be performed to check for the ID of the eye simulation in all data relevant to eye data entries. The procedure is a sophisticated one requiring several joins, nevertheless it is possible and already implemented in YAMM.

Actually, the real problem is that the query will return back three eye data entries, without any indication of which entry represents which eye (i.e. left, right or reference). This vital information is lost during the mapping of the SEE-KID model into YAMM's metamodel. There is no possibility to additionally store this information into the original mapped model. Furthermore, there is no restriction on the number of eye data models that are represented for one simulation. It is possible that an error occurs, and fewer or more models are inserted than the exact required amount. This feature is not desirable by SEE-KID and its presence could lead to problems.

A similar problem is experienced in expressing the complex type *SoapTriangulationStructure*. A triangulation structure is an entity represented using three points: *startPoint, end-*

[1] `http://gsoap2.sourceforge.net/`

Point and *triangulationPoint*. Each of these points is a complex term of type *Soap3DPoint*. After the transformation into YAMM, the same problem explained above occurs here as well. Vital information is lost regarding which of the 3D points represents which point for a particular triangulation structure.

Consequently, we concluded that a flaw exists in YAMM in relation to the storage of complex terms and one-to-one relationships. This problem means that YAMM fails to map all data models precisely and therefore cannot be used for all applications without having to fix this problem first. Additionally, it would be a plus if the querying mechanism involving the multiple joins was simplified.

The tasks required to solve the problem of YAMM is out of the scope of this work. However the initial attempt to resolve this issue will involve adding a new entity to the YAMM entity relational model to maintain the relations between complex types and entities in a named and organised manner. This will also help increase the querying efficiency since it will remove the unnecessary numerous joins performed between tables.

Once the changes have been performed to YAMM, making it capable of representing the SEE-KID model without ambiguity, it can be instantiated and the use of YAMM as a back-end will be feasible. Due to the fact that the modifications of YAMM are currently under development, the task of re-introducing the new back-end changes to the new YAMM system and instantiating YAMM with SEE-KID data will have to be done in the future.

Formerly in the work of [27], profiling was performed for the SEE-GRID persistence component. Similarly, it would be of value to perform such quantitative analysis of the newly introduced back-end. This will help evaluate its performance and assess the benefit of its usage. The quantitative analysis was not possible in the scope of this work, since the instantiation of the metamodel was not yet feasible.

5.5 Summary

This chapter included investigating the instantiation of YAMM using the SEE-GRID data model. It started with an overview of the process of automating YAMM's instantiation. Then the SEE-GRID data model was explained. This followed by a discussion of how this model can be transformed into one that can be automatically instantiated with YAMM. During this step, it was discovered that YAMM introduces ambiguity in translating some required relations. Thus it was revealed that YAMM has to be modified before use with SEE-GRID. In the end, a few suggestions were given for future work related to these issues.

Chapter 6

Conclusions

To conclude the thesis, this chapter presents the overall achievements accomplished and a discussion of suggestions for the next steps to be followed.

6.1 Achievements

The main aim of this thesis was to re-engineer the back-end required by SEE-KID to a simpler solution that offers better efficiency and avoids the performance bottlenecks incurred. The existing back-end implementation was based on a metamodel which was too complex and over-engineered, causing a decrease in performance as the size of the processed data increased; thus hindering calculations.

Our solution approach involved using a metamodel framework (YAMM) to design and maintain a metamodel based back-end. YAMM offers a means to store both the data model and data in a relational back-end in the form of 11 database entities. The idea was to instantiate an instance of YAMM with the data model required by SEE-GRID. YAMM handles query construction and data retrieval in a more simpler and more efficient way compared to that of the current version of SEE-GRID. Thus it offered a promising means to solve the performance problems regarding the metamodel back-end.

Nevertheless another requirement for the evolution of SEE-KID, is to have a Grid-aware back-end for SEE++. Thus it should be possible to have the metamodel integrated into Grid environments, and usable by Grid data resource management tools such as OGSA-DAI or AMGA. This need required enhancing YAMM such that it allows the use of any database back-end.

Accordingly, the first task of this work involved investigating the structure of YAMM and analysing its back-end operations. We compiled a summary of the different back-end operations. From this, we suggested a generic structure to hold a representation of all the database back-ends. Several examples were documented in this thesis on how to create the queries using the newly created structure. YAMM is implemented in PHP and the interoperability layer (including the generic query structure) is implemented in Java. Therefore, the need for use of a bridge between PHP and Java arose. Such a bridge already exists, we investigated the usage of this bridge for the purpose of the intended connection and we integrated the bridge into YAMM.

Moreover, interfaces were made to translate the generic query structure into the syntax of any query language, and also to connect YAMM to the chosen back-end option. The introduction of the generic query structure would enable the use of various options. In this work, we realised the connection to a relational database (MySQL) using a JDBC connection and a query parser with SQL as the querying language.

Finally, the means to instantiate the newly created generic and multiple back-end YAMM with SEE-KID data were investigated. A brief discussion was given of how this procedure should be done. However during this step, we discovered a deficiency in YAMM, making it incapable of mapping the SEE-KID data model. It was found to be incapable of modelling the required relations and thus it has to be fixed.

This work presented the opportunity to get acquainted with Grid technologies such as OGSA-DAI and AMGA. The chance did not present itself to actually use the toolkits for the purposes intended, however we (the author) spent time researching on their operation. Moreover, the investigation of the data model used by the SEE-KID project, required the installation and running of SEE-GRID's persistence component and getting familiar with the different functionalities and implementation details. Furthermore, working with YAMM gave us the chance to understand the concept of a metamodel and comprehend its advantages. The inspection of the queries used by YAMM and clustering their types inspired us to derive the generic query structure that was generated in our work. A search for a bridging tool between PHP and Java was required to facilitate the communication between YAMM and the interoperability layer. Such a bridge was found, and its usage was integrated into YAMM.

6.2 Future Outlook

Since work on both projects, SEE-KID and YAMM, are still on-going; several suggestions were made through this work for their further development and to help guide the next steps in relation to fulfilling their aims.

In the direction of the YAMM project, it is required to enhance YAMM so that it overcomes the problem that it is currently facing. One idea for resolving the problem was mentioned in the thesis. Moreover once the new version of YAMM is available, it will be required to re-introduce the interoperability layer into YAMM so that it is able to use the multiple back-end interfaces.

From the Grid-environment point of view, the creation of a Grid-interface that implements the generic connection interfaces is desirable in order to compare the efficiency of using such a back-end for the purposes of the SEE-KID project.

On the other hand, regarding the efforts in relation to the SEE-KID project, it is required that once the previous tasks are performed; the instantiation of the SEE-GRID data model using YAMM is performed. It will be of interest to try multiple back-ends, like other relational, XML or Grid-aware databases. Hence, realising a quantitative analysis (like that that [27]) of the performance evaluation of the introduced YAMM, and of the use different back-ends will be of significant value to the SEE-KID project. Furthermore, a front-end will be needed to interface the SEE++ software with YAMM, preferably using PHP and web services to reduce the number of technologies used.

Bibliography

[1] Allcock, B., Bester, J., Bresnahan, J., Chervenak, A. L., Foster, I., Kesselman, C., Meder, S., Nefedova, V., Quesnel, D., and Tuecke, S. Data Management and Transfer in High Performance Computational Grid Environments. *Parallel Computing Journal 28* (2002), 749–771.

[2] Anley, C. Advanced SQL Injection in SQL Server Applications. *White paper, Next Generation Security Software Ltd* (2002).

[3] ARDA Metadata Catalogue Project. `http://amga.web.cern.ch/amga`. Last accessed, 28/4/2009.

[4] Bagui, S., and Earp, R. *Database Design Using Entity-Relationship Diagrams.* Auerbach Publications, June 2003.

[5] Bosa, K., and Schreiner, W. The Porting of a Medical Grid Application from Globus 4 to the gLite Middleware. In *Proceedings of DAPSYS 2008* (September 2008), Peter Kacsuk et al., Ed., Springer, pp. 51–61.

[6] Bosa, K., Schreiner, W., Buchberger, M., and Kaltofen, T. SEE-GRID, A Grid-Based Medical Decision Support System for Eye Muscle Surgery. In *Proceedings of 1st Austrian Grid Symposium* (2005), Austrian Computer Society (OCG), Hagenberg, Austria, pp. 61 – 74.

[7] Bosa, K., Schreiner, W., Buchberger, M., and Kaltofen, T. A Grid-Based Medical Decision Support System for the Diagnosis and Treatment of Strabismus. In *Fifth EGEE Conference, International Conference Centre Geneva, Geneva, Switzerland* (September 2006).

[8] Buchberger, M. *Biomechanical Modelling of the Human Eye.* PhD thesis, Johannes Kepler University Linz, Austria, 2004.

[9] Buchberger, M., Kaltofen, T., and Priglinger, S. *SEE++ User Manual*. RISC Software GmbH, Research Unit Medical Informatics, Hagenberg, Austria, May 2009.

[10] Chen, P. P. S., Ed. *The Entity-Relationship Approach to Logical Data Base Design*. The Q.E.D. Monograph Series, Data Base Management, Q.E.D. Information Sciences, Wellesley MA, 220 pages, 1979.

[11] Flanagan, D. *Java in a Nutshell, Fifth Edition*. O'Reilly Media, Inc, 2005.

[12] Foster, I., and Kesselman, C. *The Grid 2: Blueprint for a New Computing Infrastructure*. Elsevier, 2004.

[13] Foster, I., Kesselman, C., Nick, J., and Tuecke, S. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, June 2002.

[14] Foster, I., Kesselman, C., and Tuecke, S. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications 15* (2001).

[15] Fuhrmann-Koch, M. Grid Computing-Technologie: Rechenleistung wie Strom aus der Steckdose ziehen. `http://idw-online.de/pages/en/newsimage?id=60138&size=thumbnail` Last accessed, 5/7/2009.

[16] gLite Homepage. `www.glite.org`. Last accessed, 5/5/2009.

[17] Globus Toolkit Homepage. `www.globus.org`. Last accessed, 5/5/2009.

[18] Grid Cafe. `http://www.gridcafe.org/index.html`. Last accessed, 20/4/2009.

[19] Groff, J. R., and Weinberg, P. N. *SQL: The Complete Reference*, 2 ed. Osborne Complete Reference Series. McGraw-Hill Osborne, August 2002.

[20] GT 4.2 Grid FTP User's Guide. Globus alliance `http://www.globus.org/toolkit/docs/4.2/4.2.0/data/gridftp/user/`. Last accessed, 28/4/2009.

[21] GT 4.2 Grid RfT User's Guide. Globus alliance`http://www.globus.org/toolkit/docs/4.2/4.2.1/data/rft/user/`. Last accessed, 28/4/2009.

[22] GT 4.2 RLS User's Guide. Globus alliance`http://www.globus.org/toolkit/docs/4.2/4.2.0/data/rls/user/`. Last accessed, 28/4/2009.

[23] Habela, P. *Metamodel for Object-Oriented Database Management Systems.* PhD thesis, Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland, November 2002.

[24] Hibernate Homepage. `http://www.hibernate.org`. Last accessed, 20/4/2009.

[25] Kaltofen, T. Software Architecture of SEE-GRID Persistence-Component. Unpublished Presentation Slides, RISC Software GmbH, Research Unit Medical Informatics, Hagenberg, Austria, April 2006.

[26] Koblitz, B., and Santos, N. AMGA User's and Administrator's Manual, June 2007. `http://amga.web.cern.ch/amga/pages.html`.

[27] Matkó, I. Z. Grid-aware Database Support for Medical Software. Master thesis, International School of Informatics, Hagenberg, Austria, July 2008.

[28] Medical Dictionary from The Free Dictionary by Farlex. `http://medical-dictionary.thefreedictionary.com/aneurysm`. Last accessed, 28/4/2009.

[29] Mitterdorfer, D. Grid-Capable Persistence Based on a Metamodel for Medical Decision Support. Diploma thesis, Fachhochschule Hagenberg, Austria, June 2005.

[30] MySQL 6.0 Reference Manual, Chapter 12: SQL Statement Syntax. Sun Microsystems. `http://dev.mysql.com/doc/refman/6.0/en/sql-syntax.html`Last accessed, 1/6/2009.

[31] PHP Manual. Php documentation group, `http://www.php.net/manual/en/index.php`. Last accessed, 1/6/2009.

[32] PHP/Java Bridge Project. Jost Bökemeier and Jon Koerber. `http://php-java-bridge.sourceforge.net`. Last accessed, 1/6/2009.

[33] RISC Software GmbH, Research Unit Medical Informatics, H. A. YAMM Documentation. (Work in Progress), April 2009.

[34] Soanes, C. *Compact Oxford English Dictionary*, 3 ed. Oxford University Press, 2005.

[35] The OGSA-DAI Project. `www.ogsadai.org.uk`. Last accessed, 5/5/2009.

[36] Watzl, J. Investigations on Improving the SEE-GRID Optimization Algorithm. Master's thesis, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, June 2008.

23 Ebad Elrahman St.
Sheraton Heliopolis,
11361 Cairo, Egypt

(+2012) 245 7 245
amira.gamaleldin@gmail.com

Amira Gamaleldin Zaki

Education

2008 – 2009 Johannes Kepler University, Linz, Austria. **International Master** of Informatics: Engineering and Management

2007 – 2008 German University in Cairo, Egypt. **Master** courses in Computer Science and Engineering

Spring 2007 Visiting student at the Bioinformatics department in **Albert Ludwigs University**, Freiburg, Germany. Completed bachelor thesis with grade A+, "Identifying Key Regulators in Gene Regulatory Networks"

2003 – 2007 German University in Cairo, Egypt. Faculty of Media Engineering and Technology. **Bachelor of Computer Science and Engineering.** Accumulative GPA: 0.86, ranked **first** on faculty

1997 – 2003 Al Nahda National School for Girls, Abu Dhabi, United Arab Emirates. Completed **IGCSE** with an average of 119.3%

Languages

- Fluent English and Arabic
- Fair German

Programming Skills

Databases:
- Proficient in developing relational databases using SQL
- Very good capability of using XML, XSLT and XPath

Java:
- Proficient in working with Java 6 and previous releases
- Proficient in developing applications with J2SE and J2ME technologies
- Proficient in developing web applications using JSP and Java Servlets

C++:
- Capable of working using C++
- Familiar with the openGL and openCV libraries

Other programming languages:
- Very good knowledge of Prolog, LISP and Haskell
- Very good knowledge of using Constraint Programming and Constraint Handling Rules libraries with SICStus Prolog.
- Very good knowledge of the Hardware description language Verilog

Scripting:
- Good knowledge of XHTML, CSS and Javascript

Computer Skills

Platforms: Capable of working under Windows and Linux
Software: MS Visual Studio, Eclipse, Netbeans, Altova XML, SICStus Prolog
UML Tools: Rational Rose
Others: Microsoft Office Bundle

Internships

[March – May 2007] Chair for Bioinformatics, Albert Ludwigs University, Freiburg, Germany; involvement in researching in the field of Bioinformatics with focus on identifying key regulators in genetic regulatory networks and signalling networks, under supervision of Prof. Dr. Rolf Backofen (granted German Academic Exchange Service (DAAD) scholarship and GUC academic excellence award)

[July – August 2006] Programming Methodology and Compiler Construction Department, University of Ulm, Germany; involvement in researching in the fields of Constraint Programming (CP) and Constraint Handling Rules (CHR) under supervision of Prof. Dr. Thom Frühwirth. Implemented an automatic examination schedule generator using CP (granted German Academic Exchange Service (DAAD) scholarship and GUC academic excellence award)

Computer Science Department, GUC; Student teaching assistant :

[Spring 2008] Computer Programming Lab (Advanced game programming using Java) – Senior course co-ordinator responsible for preparing course material, overall course organization and instructing several classes

[Winter 2007] Data structures and algorithms lab (using Java)

[Spring 2006] Computer Programming Lab (Advanced game programming using Java)

[Spring 2005] Introduction to Computer programming (using Java)

[Winter 2005] Electronics Department, GUC; Lab Assistant for Electric Circuits lab

Extracurricular activities

Member of the **Pioneers Active Working** Group in the GUC; member of the Human Resources Management Department, Database Administrator of the Research Database and also a member in the Engineering academic work-team (www.pioneersawg.com)

Volunteer work in the **Zusammen** Club (Charity Club in the GUC)

As part of the Pioneers Active Working Group activities, I taught an **Introductory Course to Java** in the semester break to younger students. (February 2005)

Completed a one month **German language course** in Heinrich-Heine University, Düsseldorf, Germany, (granted German Academic Exchange Service (DAAD) scholarship), Summer 2004

Assisted researchers in the typing, organization and manual work of researches done for the **Faculty of Medicine in Ain Shams University**, Cairo, Egypt

Personal Information

Date of birth: 29/11/1986
Place of birth: Cairo, Egypt
Nationality: Egyptian
Gender: Female
Marital status: Single

References

Furnished upon request

Erklärung

Ich erkläre an Eides statt, das ich die vorliegende Masterarbeit selbststndig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Hagenberg, am 15. Juli 2009 Amira Zaki