Development of a Web-Based Application for Collecting Models and Supporting the Design of AMS Systems

Thomas Böhm

Delivered: 1st February 2007
Contents

1 Introduction ........................................... 1
   1.1 Motivation ........................................ 1
   1.2 Intention of the Thesis ......................... 2
   1.3 Organisation of the Thesis ...................... 3

2 Fundamentals of Software Engineering .......... 5
   2.1 Software Life Cycle ................................ 6
       2.1.1 Problem Definition ............................ 6
       2.1.2 Requirements Analysis and Specification .... 8
       2.1.3 Design ....................................... 8
       2.1.4 Implementation .............................. 9
       2.1.5 Acceptance Testing .......................... 10
       2.1.6 Deployment .................................. 10
       2.1.7 Life Cycle Models ........................... 11
   2.2 Unified Modelling Language (UML) ............ 13
       2.2.1 Use Case Diagram ............................ 14
       2.2.2 Class Diagram ................................ 15
       2.2.3 Activity Diagram ............................. 18
       2.2.4 State Machine Diagram ....................... 19
       2.2.5 Sequence Diagram ............................ 20
   2.3 Tools Supporting Software Engineering ........ 23

3 Fundamentals of Web-Based Applications ....... 25
   3.1 Web Application Architectures .................. 25
       3.1.1 Client/Server Architecture ................... 26
       3.1.2 Two-Tier Architecture ......................... 26
       3.1.3 N-Tier Architecture ........................... 28
   3.2 A Closer Look at the N-Tier Architecture ...... 28
       3.2.1 Data Tier ..................................... 28
       3.2.2 Business Tier ................................ 33
       3.2.3 Presentation Tier ............................. 39

4 Problem Definition .................................... 41
   4.1 Functional Requirements .......................... 41
   4.2 Quality Requirements .............................. 42
   4.3 System Requirements .............................. 42
   4.4 Process Requirements .............................. 42

5 Requirements Analysis and Specification ...... 45
   5.1 Validation of the Requirements ................ 45
List of Figures

1.1 V-model for the design process of a technical system [MV06] .................. 2
2.1 Phases of the software life cycle [Dum03] .................................. 7
2.2 Waterfall Model [Som01] .................................................. 11
2.3 Incremental software development [Dum03] ................................ 12
2.4 Software development as Prototyping [Dum03] ............................ 13
2.5 UML 2.0 diagram types .................................................. 14
2.6 Example of an Use Case Diagram [Obj05] ............................... 15
2.7 Example of a Class Diagram ............................................. 16
2.8 Example of an Activity Diagram ........................................... 18
2.9 Example of a State Machine Diagram ................................... 20
2.10 Example of a Sequence Diagram [Kec05] ................................ 22
3.1 Types of client/server architecture [DLWZ03] ............................. 27
3.2 Example of a two-tier architecture [KPRR06] ............................ 27
3.3 N-tier architecture [KPRR06] .............................................. 29
3.4 Terms of the relation model [SST97] .................................... 30
3.5 Example of an Entity-Relationship-Diagram ............................. 33
3.6 Java EE platform APIs [JBCE06] ....................................... 34
3.7 Sequence Diagram of the JSP life cycle ................................ 36
3.8 Example for the interaction between JSP and EJB ....................... 37
4.1 ModelLib Use Case Diagram .............................................. 43
5.1 Architecture of the ModelLib prototype [MV06] .......................... 48
5.2 Architecture of ModelLib with the new technology ...................... 51
6.1 ER Diagram of the ModelLib database (structural meta information) . . . . . . 55
6.2 ER Diagram of the ModelLib database (document meta information) . . . . . . 56
6.3 ER Diagram of the ModelLib database (user meta information) .......... 57
6.4 General concept of access control in ModelLib ......................... 60
6.5 Composition of the view .................................................. 61
6.6 Example of the DocumentACL table .................................... 64
6.7 ER Diagram of ModelClass .............................................. 65
6.8 Activity Diagram of the function called by the ModelClassEffectiveACLView 67
6.9 Activity Diagram of the recursive function to compute the effective rights . . . 68
6.10 Example of linked entities without an own ACL ......................... 70
6.11 Activity Diagram of the rights inheritance .............................. 74
6.12 Composing the table with the effective access rights ................. 75
6.13 Activity Diagram of the trigger function ................................ 76
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.14</td>
<td>Class Diagram of a possible business tier of ModelLib</td>
<td>78</td>
</tr>
<tr>
<td>6.15</td>
<td>Possible elements of the presentation tier of ModelLib</td>
<td>79</td>
</tr>
<tr>
<td>6.16</td>
<td>Possible layout for the main ModelLib Web page</td>
<td>79</td>
</tr>
<tr>
<td>6.17</td>
<td>Presentation of the meta information about design tools [MV06]</td>
<td>80</td>
</tr>
<tr>
<td>6.18</td>
<td>Presentation of the meta information about a model [MV06]</td>
<td>81</td>
</tr>
<tr>
<td>7.1</td>
<td>An example of a model class hierarchy</td>
<td>85</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Process perspective of CASE tools [Som01] ........................................... 23
5.1 Characteristics of Web applications and the relation to this project .......... 46
5.2 Evaluation of technologies in the business tier ........................................ 49
7.1 Possible ACL values and their theoretic result in a computation ............. 85
7.2 Possible roles with and without membership and their rights ................. 89
7.3 Results of the performance test of the triggers ...................................... 90
A.1 List of all tables in the meta information database .............................. 97
A.2 List of all views in the meta information database ............................... 102
A.3 Table of views depending on the RolesMembershipView ..................... 107
List of Acronyms

ACL  Access Control List
AMS  Analogue and Mixed-Signal
API  Application Programming Interface
ASP  Active Server Pages
CASE Computer Aided Software Engineering
CGI  Common Gateway Interface
DBMS Database Management System
DBS  Database System
DLL  Dynamic-Link Library
EDA  Electronic Design Automation
EJB  Enterprise Java Beans
ER  Entity-Relationship
GUI  Graphical User Interface
HTML Hypertext Markup Language
HTTP Hypertext Transfer Protocol
HTTPS Secure Hypertext Transfer Protocol
IC  Integrated Circuit
IDE  Integrated Development Environment
IP  Intellectual Property
Java EE Java Platform, Enterprise Edition
JDBC Java Database Connectivity
JMS  Java Message Service
JNDI Java Naming and Directory Interface
JSF  Java Server Faces
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSP</td>
<td>Java Server Pages</td>
</tr>
<tr>
<td>MDB</td>
<td>Message-Driven Beans</td>
</tr>
<tr>
<td>MOEMS</td>
<td>Micro-Opto-Electro-Mechanical System</td>
</tr>
<tr>
<td>ODBC</td>
<td>Open Database Connectivity</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>PHP</td>
<td>Hypertext Preprocessor</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>SoC</td>
<td>System-on-a-Chip</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>VHDL-AMS</td>
<td>VHSIC Hardware Description Language – Analogue and Mixed Signal</td>
</tr>
<tr>
<td>W3C</td>
<td>World Wide Web Consortium</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Motivation

Systems-on-Chips (SoCs), as combinations of computer and communication hardware and software equipped with autonomy based on perception, cognition and control capabilities, are key parts of a continuously broadening range of applications, from industrial equipment to personal appliances. The design of SoCs has currently to address a number of significant issues, namely:

- **Increasing complexity**, due to the integration of significant computing and communication power (intelligent systems).
- **Significant heterogeneity**, due to the variety of integrated components (analogue/RF/digital hardware, embedded software, sensors, actuators).
- **Increasing environmental awareness**, due to energy saving, battery operated systems, environmental monitoring capabilities, and continuous interaction with the working environment.
- **Increasing impact of modern silicon technologies**, due to deep sub-micron and nanometer technological processes.
- **Increasing re-use of subsystems**, due to ever shrinking time to market and rapid product obsolescence.

The fast progressing advances in manufacturing technology allow the integration of more and more functionality from different disciplines into a single complex heterogeneous SoC. This leads to a continuously growing in the needed design effort where at the same time product cycles get shorter. The resulting increase in the “design productivity gap” is especially notable in semiconductor industry. There the technological production capacity (measured by the number of available transistors) has increased since 1985 yearly between 41% and 59% whereas the design capacity (measured by the efficient use of transistors) has increased only at a yearly rate of 20% to 25% [OTDRG05]. To allow the control of the design costs and prevent them to get prohibitively expensive, new design technologies have to be continuously introduced, like block reuse or Integrated Circuit (IC) implementation tools.

Analogue and Mixed-Signal (AMS) hardware systems are a predominant part of today’s SoCs used in telecommunication, automotive, and multimedia application areas, which is likely to increase in the future. AMS hardware systems typically include a digital core, consisting of embedded software running on a microprocessor, and AMS interfaces to the outside world, e.g., audio/video/RF interfaces, which are including components such as PLLs, filters, and amplifiers, or Micro-Opto-Electro-Mechanical System (MOEMS) sensors or actuators.

The design of AMS hardware systems has to cope with the heterogeneity of their components and with different design methodologies. For example, the design of a sensor component (e.g., an accelerometer) involves dealing with physical information such as mechanical forces and electrostatic field distribution between the electrodes, while the design of an electronic component...
1 Introduction

Recursions
Requirements analysis
Component design
System design
Acceptance test
System test
Component test

Problem definition
Acceptance test
Functional models
System performance
System test
Component performance
Component test

Input
Output

Repetitions
Interactions

Figure 1.1: V-model for the design process of a technical system [MV06]

(e.g., a filter) involves dealing with more abstract voltages and currents. In all cases, models of devices, components or of the whole system under design are created or used for either exploring the performances of possible architectures or validating an existing realisation in reasonable simulation times. Executable models, that is models that can be processed by a design tool such as a simulator or a synthesiser, are defined using a hardware description language such as VHDL-AMS [IEE01, PLV05] Verilog-AMS [Acc04, PLV05] or SystemC-AMS [VGE05, Ein05]. They can describe structures and behaviours of hardware components in various levels of details. They also provide an excellent documentation of the designer’s intents.

One important aspect of the AMS design flow (see Figure 1.1) is therefore an efficient management of models promoting reuse. Designers usually only reuse their own models or those provided by their direct design environment. An exchange of models between designers is complicated by the fact that they are often not aware if and where a similar model already exists. Furthermore models need to be properly documented. Essential information includes the model’s interface, its implementation, the effects covered, how it has been verified, and properties supporting the design such as parameters for guiding the synthesis process (e.g., component topology selection to meet system-level specifications).

1.2 Intention of the Thesis

The objective of this Master thesis work is to contribute to the development of a web-based application that is able to collect models of AMS hardware systems in various modelling languages (e.g., VHDL(-AMS), Verilog(-AMS), SystemC(-AMS)) and store meta information about the models and their related modelled hardware components. The application shall provide flexible ways to register new models, to browse through the existing model library, to allow collaborative review and improvement of models in the library, and to support complex queries about models in the library that come from the AMS design process. The last kind of task may be performed either by the designer through a Graphical User Interface (GUI), or by an Electronic Design Automation (EDA) tool through a specialised Application Programming Interface (API).

Starting from a first prototype, developed by Torsten Mähne and Alain Vachoux [Mä06], a widely improved application shall be designed and implemented. This new version has to consider extended requirements to the system, e.g., extended database scheme for model meta information, a fine grained access control mechanism. An appropriate architecture and technology for such a Web-based application have to be chosen to meet the requirements. Thereby this thesis describes the realisation (especially the access control mechanisms) from the perspective of a software developer.
and not from the perspective of an AMS designer. This means that the focus is put on the application itself and not its support of hardware system design.

1.3 Organisation of the Thesis

After a description of the motivation and the intention of this thesis in the introduction, Chapter 2 gives an overview of the relevant aspects of software engineering to cover methodical approaches for the realisation of software projects. Chapter 3 contains the theory about Web-based applications. Some system architectures and technologies are introduced there. After these two chapters of theoretical fundamentals, follows the description of the development of the system, which is based on software engineering methods. Chapter 4 presents the requirements for the system to be developed. Chapter 5 validates the requirements and discusses the decision for a certain architecture and technology to realise them in an appropriate way. Afterwards, the design of the application and its components is explained in Chapter 6. It contains, among other things, the structure of how meta information about design models are stored. Also chosen aspects, e.g., a tuple-wise access control mechanism, or recursive functions to extend SQL queries, are described. Some examples of the implementation and test cases are the topic of Chapter 7. Summarising this work, Chapter 8 presents a conclusion and an outlook of possible future work addressing the application.
1 Introduction
2 Fundamentals of Software Engineering

Since the late 1960’s it is well known that successful development of software products requires a systematic and disciplined approach. This approach is the field of software engineering. Its goal is to reach cost effective, high quality products. This chapter will give an overview of the software engineering topics, which are relevant to this thesis.

Web-based systems are still software products, though slightly different from "classic" software, so they need basically the same approach. As described in the paper [KMP+04] these differences are:

**Application-Related Characteristics** – e.g., the content-driven nature of the system, frequently changing content, non-linear structure in usage and the visual presentation as central qualifying factor.

**Usage-Related Characteristics** – e.g., a potentially high number of users, who differ in social and cultural background, age, skills, intentions, hardware and software. They all have a demand of service on any time immediately.

**Development-Related Characteristics** – e.g., often applied multi-disciplinary development teams, the involvement of open source products and communities, integration of heterogeneous legacy systems and volatility of technologies for the development itself.

It has to be emphasised that some of these characteristics can occur in traditional applications, but all together they represent the specifics of web-based systems, even if the degree of each characteristic may vary depending on the aim of the web application. However, every development process of software should be seen as a problem to be solved by the use of appropriate methods. That includes well defined aims and requirements, systematic development process divided into phases, diligent design of these phases and continuous monitoring of the whole process. Thus this work will not further explicitly distinguish between engineering of a web-based or non-web application. The following definition shall express the understanding of web engineering of the author:

> “Web Engineering is the application of systematic and quantifiable approaches (concepts, methods, techniques, tools) to cost-effective requirements analysis, design, implementation, testing, operation, and maintenance of high-quality Web applications.” [KPRR06, p. 3]

It is also helpful to define the term Web-based application/software:

> “A Web application is a software system based on technologies and standards of the World Wide Web Consortium (W3C) that provides Web specific resources such as content and services through a user interface, the Web browser.” [KPRR06, p. 2]
2.1 Software Life Cycle

This section describes how a systematic development process should look like. Therefore the typical phases of the software life cycle and their dependencies will be given and explained. Further some of the common models of the software life cycle are presented.

The software life cycle itself is seen as a process to develop software from the first step to the final running product and its maintenance up to its replacement. Figure 2.1 shows an overview of different phases of the software life cycle. Due to the focus of this work software use and maintenance are just mentioned. For more details about these phases can be found in the corresponding literature [Bel00, Dum03, Pre01, Som01].

2.1.1 Problem Definition

The first phase of the software development life cycle is the problem definition. [Dum03] In most cases it starts with the order of a customer for a certain software. That means that a strong communication between customer and developer takes place to work out a brief description about the product. The client gives his wishes about the software to the software engineer in an informal way, mainly text documents. From this the requirements are derived.

Before going on, it has to be mentioned that the term requirement is not used throughout the software industry consistently. In this paper it is understood as the a property to be met or a service to be provided by a system [KPRR06]. To take a closer look at the requirements the categories are used as they are published in [Dum03]. The book separates:

**Functional Requirements** pool all functions expected from the system. Furthermore information about interfaces to other tools or the user are given, as well as information about input or output data the system has to work with or to produce. Ideally developer and client together devise significant use cases, so that both sides clearly understand what the system should do.

**Quality Requirements** include all goals of quality the software should meet or quality standards to follow. This could be criteria as usability (ergonomics, documentation, learning effort), reliability (stability, fault tolerance), efficiency (performance, resources, system behaviour), maintainability (testability, readability, changeability) and portability (substitutability, compatibility).

**System Requirements** describe possible hard or software technologies to be used in the product or for the process.

**Process Requirements** eventually define details about project specific topics, such as the time schedule, milestones, financial resources, and organisational resources, as well as personnel resources.

Other sources use different categories for requirements, e.g., functional and non-functional (with product, organisational, ethical) requirements [Som01] or functional and data requirements as well as constraints (e.g., for cost, hard-,software, delivery date) and guidelines [Bel00]. They are also an adequate way to structure the wishes of a customer.

All structured requirements together are the foundation, to speak as an engineer, on which both sides agree to go on with the next step of development.
2.1 Software Life Cycle

Figure 2.1: Phases of the software life cycle [Dum03]
2.1.2 Requirements Analysis and Specification

The next phase is the requirements analysis and specification. Some literature refers to it requirements engineering [Pre01, Bel00]. The subject of this phase is the validation of the requirements regarding their correctness, completeness, conformity, consistency, and feasibility. Also requirements have to be realistic, need by the customer, verifiable, and traceable. This phase is often underrated, even though it is very important for the success of a development process. For example, a review of completed software projects had shown that 13.1% failed because of incomplete requirements and 12.4% because of insufficient involvement of the customer [Dum03]. The requirements analysis adopts some methods and techniques to serve its purpose, for example brainstorming and interviews with the customer. This of course is difficult for web-based applications, because brainstorming or interviews with millions yet unknown customers is hard to do. Another technique is the systematic elimination of ambiguous terms and the definition of a domain glossary if needed. On the formal site effort estimation methods are used to make statements about costs and time resources.

After that, or sometimes in parallel, the actual software development begins with the so called specification. The earlier defined requirements are transformed into models, which depict the software in its whole functionality. The modelling is the structural, operational and informal transformation of requirements in a form, so that the result is easy to understand and interpret likewise for developer and client. The structural modelling is the abstraction and partitioning of requirements, but without the loose of information. The operational modelling is based on computer aided methods like simulation or animation. Interviews, analysis and investigations are the methods of informal modelling. Sometimes layouts for the user interface are drawn or a first flat prototype is implemented to clear details of the system. However, all kinds of modelling aim to a global conceptual model representing the system. In most cases different types of models are used in a consistent relation to each other to handle the complexity of a product (e.g., function model, data model, state model, work-flow model). [Dum03]

Beside this the specification should describe scenarios for testing of units, modules, the complete system, and its acceptance. Also a concept for the technical and user documentation should be formalised inhere [Som01].

To summarise, the specification is the complete and detailed definition of the functional and some quality requirements in form of models. It does yet not include the system requirements.

It has to be emphasised that under some conditions the design is directly following the requirements analysis. This makes sense whenever a system is very simple or heavily bound to a certain architecture [Dum03]. Web-based applications often show the characteristic that a legacy system has to be integrated in the new product. In this case the work of detailing should be shifted to the design phase, because of its influence to the new architecture [KPRR06].

2.1.3 Design

In the design phase all models from the specification are converted to an architecture, which takes the system requirements into account. One differentiates between product implementation requirements, coming from the problem definition (Section 2.1.1) and predefined platform conditions, which effect the design of a particular system architecture [Dum03].

The designed architecture contains the structure of the system or its components (also referred as modules) and their interfaces as well as its relations and interactions with each other. A component is a encapsulation of functions. In some projects components are an already existing software (so
2.1 Software Life Cycle

called legacy system) to add to the new system (e.g., a new system working together with a commercial database). Thereby the integration of these external components ranges from simple interaction via an interface to complete embedding. In general, interfaces should connect the modules and hide the internal structure, but this is not always practicable, because of efficiency reasons.

Once the components and their relations are modelled in an hierarchical way, a decision about a certain design technique to be followed can be mad. Typical methodical approaches to reach a high quality design are Top Down, Bottom Up, Hardest First or Trail and Error.

The act of design itself is done by semi-formal or formal methods. A various number of such methods, like Structured Programming, Flow Charts, Function Charts, Event Chart, Pseudo Code, Petri Nets, Semantic Object Model, Architecture of Integrated Information Systems or the UML (see Section 2.2) exists and are supported by an even higher number of tools, the so called Computer Aided Software Engineering (CASE)-tools. Some of the methods are already in use during the phase of specification (Section 2.1.2). To describe them all would go far beyond the scope of this work. Therefore the reader is referred to the appropriate sources (except for the UML because of its application within the development project as part of this thesis).

As result of the design the developer should be able to implement the system directly out of the documents generated during this phase. Further, more detailed test cases as well as user and development documentation should also be part of the output. [Bel00]

2.1.4 Implementation

The phase of implementation follows after the design. As its result a runnable software system is coded. To do so basically four techniques exists [Dum03]:

- **Editing** is the manual writing of source code for a software, mainly by the support of an editor or a development environment.

- **Generating** means the computer aided form of code production. It is a time reducing procedure, but only if the design is sufficient enough. Of course, an adequate code generator is required too. In general the so called templates are generated as a body, in which the final code will be edited.

- **Adapting** is a technique where an already existing code is more or less modified.

- **The Reuse** is the unmodified adoption of existing source code. This method, of course, requires a fundamental knowledge and understanding of the original code. Reuse is often done in form of the usage of source code libraries, which provide solutions for common problems.

The developer should follow coding style guides as well as naming, structuring and commenting conventions to ensure the quality of the product.

Another part of the implementation is the testing of the software, to ensure that it is free of faults, errors and failure (even though all three terms have a different meaning they are equally used in here), if that is achievable at all [Som01, Dum03]. Of course, the correction should be on a minimum, because faults are avoided since the problem definition.

The testing is separated into two kinds of test methods:

**Static Program Testing** are all methods, which directly validate the source code of a program (e.g., check lists, reviews or symbolic execution).
Dynamic Program Testing are all methods, which validate the code in its executable form on a computer. An example of such a test is the so called black-box testing. Thereby the program is treated as a closed unit. It is analysed only by input and the resulting output. Due to the high range of possible input values, the analysis is done with a small set of significant values:

- normal values (occurring during a “normal” use of the software)
- extreme values (values on an edge of the range, but still valid)
- invalid values (values that should not be accepted by the program itself, e.g. character for a required integer).

Another example is a validation method named white-box test (alias glass-box or clear-box testing). This is an approach where tests of usually small units are derived from knowledge of the software’s structure. The knowledge is used to find out how many test cases are needed to guarantee that all statements of the unit are executed at least once during the test. [Som01]

Now, that the units and components are tested, they have to be integrated into a whole and then tested again. Therefor the grey-box test, a mixture of black- and white-box testing, is a common approach that validates components as well as the system as a whole [Dum03]. Depending of the component’s hierarchy the test can be either top-down or bottom-up. If all components are tested at the same time it would be a big-bang test. Once the integration tests are finished it is recommended to test the user interface and the performance behaviour. The stress testing is a stepwise exceeding of the boundaries set in the requirements (Section 2.1.1) or design (Section 2.1.3). For example a system is designed for 100 transactions per second. With the stress testing one figures out how it behaves in case of 110, 120 and so on. Doing so can identify problems that would have been left undiscovered during normal use. This is especially important for Web-based systems, because of their uncertain and unstable number of users. [KPRR06]

The use of software metrics can provide additional information about the quality of the system. It is a too extensive and too complex topic to go into details here, hence it is just mentioned (for more information see [Bel00, Dum03, Pre01, Som01]).

As part of finishing the implementation a complete user and developer documentation has to be written.

2.1.5 Acceptance Testing

The phase after the implementation is the acceptance testing, which simulates the usage of the just implemented system under real conditions to proof the matching of the costumers requirements. This is done with or under supervision of the client. Incorrect or incomplete implemented requirements are listed and revised. [Dum03]

2.1.6 Deployment

The revised software will then be deployed or delivered to the costumer in a final version. Sometimes it is also installed as part of the provided service.

In many cases client and developer agree on a contract of maintenance or future cooperation. This leads to the phases of software life cycle use and maintenance as shown in Figure 2.1. As mentioned before they are not the centre of interests of this work and therefore left out of further discussions, even though they are quit important in commercial software development projects.
2.1 Software Life Cycle

2.1.7 Life Cycle Models

The introduction above gives the phases of development one by one, but one can think of a various number of scenarios, in which phases are applied in different orders. The long term experience of software engineering in praxis teaches that several models of the software life cycle exists [Dum03]. This section gives an overview about some of the most common ones.

Two groups can be distinguished which contain themselves a number of models:

**Sequential life cycle models** are typically following the phases step by step. Examples are the Waterfall Model, the V-Model or the Clean-Room Engineering. They all clearly separate the phases from each other. [Dum03]

**Non-sequential life cycle models** have the characteristic that they are cyclic. They allow the developer to go more or less back in between the phases. Examples are evolutionary development with Prototyping or the Spiral Model, incremental development with the the Whirlpool Model or the Fountain Model. [Dum03, Pre01]

The Waterfall Model (Figure 2.2) is a good example for a sequential model. It was one of the first models in software engineering. Only if a phase is complete it is allowed to start the next one. The end of each step marks a document on which the next one builds. That is why the Waterfall Model is often referred as a document driven model [Som01]. If errors or faults are detected in the progress a single step back to the previous phase is allowed to correct them. Every know and then it can happen that a single step back is not enough to eliminate an error. In this case a whole new process must be initialised to develop a bedder new version. But this is not intended in the Waterfall Model. This conflict is by far the biggest weakness of sequential models in general.
The possibility of such complete renewing is the intention of non-sequential models. Figure 2.3 shows the technique of incremental software development. At its beginning all requirements are defined and modelled, but not all of them are implemented. At first some features are programmed. This early version will then delivered to the client. The lessons learned from its usage have an influence on the process and on the next version, which implements additional features. This will be iterated until a satisfying product level is reached. The incremental model enable the engineer to concretise fuzzy or unknown requirements stepwise. Another advantage is that every iteration produces a runnable product. The client can use this and instead of long waiting time he gets something in short periods.

An analogue technique is the Prototyping that also belongs to the group of non-sequential models. The improvement of the cooperation between client and costumer is a major goal of this approach. Therefore it is shortening the time of iterations to a minimum [Dum03]. There is a distinction between:

- **Horizontal Prototyping** – implements all required functions, but only in an adumbrated way (e.g. the layout of the user interface without its functionality).

- **Vertical Prototyping** – completely implements some chosen required function (e.g. the user account management component is fully detailed and ready to use, but user interface does not yet exist).

In both cases the client devises his requirements, then the developer quickly implements a prototype reflecting how he has interpreted the requirements. Afterwards both sides analyse and validate the result together and implement the improvements into the upcoming version. A prototype can either be used to receive perception about the feasibility (explorative Prototyping) or as an experimental...
2.2 Unified Modelling Language (UML)

Section 2.1 alludes from time to time the modelling of requirements or the system. Especially during the design phase (see section 2.1.3) models play an important role. But it is not explained how methods to model a system actually work. This section will explain the modelling, using the Unified Modelling Language (UML) as an example. Though over the past decades a various number of methods for modelling were invented and implemented in an even higher number of tools, UML is the most widespread one [Dum03].

“…UML which is emerging as a standard modelling language, particularly for object-oriented modelling.” [Som01, p. 150]

Hence it is used for modelling the system within the development project, which is subject of this thesis.

UML is a graphical language for describing object-oriented models. In 1997 it was released in its first official Version 1.0. Reputable organisations as Microsoft, Oracle, IBM, Rational Software, were involved in this release which united the work of Grady Booch, Jim Rumbaugh and Ivar Jacobson. They tried to create a unified approach and notation to model software development. [Kec05]

Since 1999 the Object Management Group (OMG) as standardisation committee is responsible for the UML and declared it as an official industrial standard. Since that time the UML has been evolved. This has lead to the current version 2.0 released in 2004. The version 2.0 provides 13...
types of diagrams (Figure 2.5). They are distinguished by static (the group of structure diagrams) and dynamic (the group of behaviour diagrams) characteristics. Some of these diagrams will be briefly explained below. Due to the complexity of the UML specification [Obj05], and thus the UML itself, this can only serve as an introduction to enable the reader to understand the basic concepts.

2.2.1 Use Case Diagram

The Use Case Diagram (Figure 2.6) provides the means to describe of the system functions in an abstract way. Thus it is used to capture a general vision on a system from a clients point of view. With its help the requirements are modelled in the phase of Problem Definition regardless any technical solution (e.g., particular programing language). The following enumeration lists the elements of the diagram.

- **Actor** – is an object interacting with the system (subject).

- **Association** – is a relation between an actor and a use case.

- **Extend** – is a relationship from an extending use case to an extended one, which specifies how and when the behaviour defined in the extending use case can be inserted into the behaviour defined in the extended use case.

- **Include** – is a relationship which defines that a use case contains the behaviour defined in another use case.

- **System name** – is apparently the name of the system.
2.2 Unified Modelling Language (UML)

Figure 2.6: Example of an Use Case Diagram [Obj05]

- **System boundary** – marks the border of a system.

- **Use case** – is the specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholder of the system [Obj05].

As an example a very simple simulator for driving a car is used in Figure 2.6. It is easy to realise that the driver is the actor, who is interacting (association) with the simulator (system) by executing the “car driving”-use case. This use case includes the use case “speed control”. Thereby it is meant that driving always and completely encloses controlling the speed. Of course one could argue that steering the way in which the car is driving would also be a use case while driving. It is right, but may be in this simulator the speed is also influenced by external effects (e.g., lading or weather) other than the direction of the vehicle. Hence the speed control is a little more laborious to implement. That is why it is explicitly modelled. On the other side parking is a special situation only occurring at the end of a ride. So, if the car is about to be left, it has to be parked. Therefore the “car parking”-use case extends the driving.

The example shows that a Use Case Diagram gives an impression of what the system basically shall do, without going into detail how. This helps the user and the developer to validate if the requirements are correctly understood.

2.2.2 Class Diagram

As a structure diagram the Class Diagram describes static aspects of a system. It gives information about the classes (mainly objects in a object-oriented context), types and their relationships. Depending on the software life cycle phase it contains platform specific details (Design) or not (Specification). The example in Figure 2.7 shows the basic elements as given in [Obj05] which are:

- **Class** – defines a set of objects that share the same features, constrains and semantics. A class has a name and can possess a set of attributes and operations.
Figure 2.7: Example of a Class Diagram
2.2 Unified Modelling Language (UML)

- **Interface** – represents a set of public features that are coherent to the ones realised in an object. Hence an interface is an abstract connection point to a class or with the words of the OMG

  “An interface specifies a contract; any instance of a classifier that realizes the interface must fulfil that contract.” [Obj05, p. 82]

- **Package** – is a collection of elements to group them in a structured way. A package also provides a name space for these elements.

- **Stereotype** – extends the existing elements or type of the meta model to classify and, if desired, define a new general element. This is sometimes necessary to meet requirements for a certain programming language.

- **Association** – describes a semantic relationship between objects. Thereby the name, the role of each object, or the cardinality (also known as multiplicity) can be specified in form of text alongside the object connecting dash. The addition of these information is also possible for the specialised association types Aggregation and Composition.

- **Aggregation** – defines a relationship between objects in the kind of “part of a whole” or in the other direction “made of”. This means that an aggregated class has instances of other classes as part of its own instance.

- **Composition** – is a stronger kind of aggregation, because unlike the parts of an aggregation the ones of a composition only exists together with the whole and not as an object itself.

- **Generalisation** – names the relationship of objects in the sense of inheritance. Each instance of the specific object is also an indirect instance of the general object. Thus, the specific object completely inherits the features of the more general object. If this object is abstract the inheriting object has to implement the abstraction.

- **Realisation** – describes an abstraction relationship of two elements, one representing a specification (the supplier) and the other represents an implementation of the latter (the client).

The example (Figure 2.7) depicts a package “car” with a class of the same name as its central piece. Car is also the name of the interface realised by the class. The car consists of an engine and at least four but at most six wheels. This is modelled using the “has a” aggregation. Further the class is marked as abstract through the italic style in which the name is written, because car is a generalisation of a bus and a truck. Hence “truck” and “bus” inherit the attributes (cartype, max. speed, speed) and the operations (accelerate, stop), but may implement them differently. Moreover, they add their own attributes and operation. In addition “truck” has a composition relationship to one or more goals which means that a truck contains one or more instances of destinations. It is a composition, because it is assumed no destination exists without a truck to go there (from the system view). The “driver” is associated with the “car” which simply says that one of them can instance the other. Due to the exemplary character only tree classes are modelled with attributes and operations.

Once again, this example claims neither to be complete nor ingenious, but should transmit how a Class Diagrams are modelled.
2.2.3 Activity Diagram

The Activity Diagram describes the sequence and conditions for the behaviour of a system or one of its parts on a detailed level. All activities are referring to either an use case or a class. According to [Obj05] its main elements are:

- **Action** – is the function unit within the activity or data flow from edges to other edges. An action will execute if all its conditions are satisfied. The completion of an action may enable the execution of other actions. Such successors use the output for their input. An action has a name and optional pre or post conditions.

- **Decision node** – is a control node which chooses between outgoing flows. It has one incoming and at least two outgoing edges. The opposite of it is a *merge* node which has several alternate incoming flows and one outgoing. It is not used for synchronising but for selecting one among the alternate flows. Both type of nodes react on conditions that specify the decision.

- **End node** – marks the final node that terminates a flow.

- **Flow** – means both object (data) and activity flow. The first describes the way of objects through an activity. The latter depicts its procedural chain.

- **Fork node** – is a control node that splits a flow into multiple concurrent flows.

---

Figure 2.8: Example of an Activity Diagram
2.2 Unified Modelling Language (UML)

- **Initial node** – marks a node at which an invoked activity starts a flow. It is possible that an activity has more than one initial node.

- **Join node** – synchronises multiple flows into one.

- **Object** – is a particular class instance available or required at a particular point in the flow. Beside a name objects can possess additional values such as their state.

Figure 2.8 is a simple example of an Activity Diagram. Therein an instance of the driver class from the Class Diagram in Figure 2.7 starts the car after the activity has been initialised. After splitting the activity flow two parallel actions follow, the controlling of where the car goes by the “controlWay” action and the “accelerate” action. As long as the place where the driver currently is equals not the destination, he keeps driving. Analogous for the acceleration, if the speed is not proper he keeps adapting it either by positive or negative acceleration. At the moment he arrives at his destination with the right speed he will stop and change his landing, which also terminates the activity.

Activity Diagrams are much more at the level of algorithm and thus mainly used in the design phase. Often the textual syntax of the diagrams elements equals the one of the programming language chosen for the succeeding implementation [Kec05].

### 2.2.4 State Machine Diagram

The subject of State Machine Diagrams are the state of objects and the functions to change this state as a way to model the object discrete behaviour. The diagram describes a hypothetic machine which is in a finite set of states at every point in time [Obj05]. The basic elements are:

- **Decision (pseudo) state** – marks a point where the path of an outgoing transition is decided in the moment of the execution of the decision state. Conditions, under which a path is selected, are written in square brackets alongside the transition. The addition of “pseudo” means, that it is a kind of abstract state, because it is different in notation and behaviour from a element state.

- **Final state** – is the element for the end of a modelled state machine.

- **Initial (pseudo) state** – depicts the beginning of a state machine. Its transition initialises the first state of the machine.

- **State** – defines a discrete situation, during which an object holds an invariant not empty set of its attributes. A state in a diagram optionally possesses a number of attributes such as:
  - entry – behaviour on an entry of the state,
  - exit – behaviour on an exit of the state,
  - do – behaviour performed as long as the object remains in the modelled state.

Beside simple states, the diagram specification also names Composite state, state list, and submachine state, which are more or less composed of simple states and thus not necessary to explain.

- **Transition** – is a directed relationship between a source state (or pseudo state) and a target state (or pseudo state). Thereby it transforms the machine from one state to another. Alongside the edge of it are optional to specify
Figure 2.9: Example of a State Machine Diagram

- trigger – multiple actions that fire the transition,
- guard – in square brackets written conditions that control the transitions firing, but in contrast to decision states without an alternative,
- effect – additional behaviour to perform when the transition fires syntactically separated by a slash after a guard or a trigger.

Most of these elements are shown in the example in Figure 2.9. The state machine there starts with a ready truck. Once the driver has started the car (condition expressed by a guard) the transition fires with the start car action which leads to the state of a started truck. As to see it performs the behaviour of driving (do attribute) while the machine remains in this state. Via transition the truck stops if the destination has been reached, then the lading is changed. The transition new lading either leads to the final state, in the case that the lading variable of the truck class possesses the value empty, or back to the state of a started truck.

As an advantage of state machines counts its formalism, which allows a designer to evaluate a model or rather a system component in a mathematical way (e.g. testing for deadlocks, reachability or liveness). They are used in the specification phase as well as in the design phase, though a more detailed in the latter. Similar to Activity Diagrams (Section 2.2.3) it is close to the implementation especially in combination with the syntax of the targeted programming language.

2.2.5 Sequence Diagram

The most common (though with the most complex specification) variant of interaction diagrams is the Sequence Diagram, which focuses on interactions via messages interchanged between lifelines of objects [Obj05]. As the name indicates it is sequential and thus time-dependent. In general it describes the dynamic intra and inter system communication by using the following elements:

- **Action** – names which interaction is performed with a message.

- Combined fragment – defines a fragment of interaction, which enables the modeller to describe a number of traces in a compact and concise manner. The operator specifies the type of fragment and sometimes has a guard to express when or if the fragment is performed. The operator is one of:
2.2 Unified Modelling Language (UML)

- alt – for at least two alternative sequences,
- assert – for an assertion that must be valid,
- break – for a breaking scenario,
- consider – for designating which message should be considered within this fragment,
- critical – for a fragment that can not be interleaved,
- ignore – for indicating that messages are not shown,
- loop – for a defined number of iteration,
- neg – for negative or invalid defined fragments,
- opt – for optional behaviour where either the optional fragment is performed or nothing,
- par – for parallel interactions,
- seq – for an unordered sequence respectively where the order is undefined,
- strict – for a strict sequencing.

- **Execution** – is a unit of behaviour or action within the lifeline. It is possible to specify conditions, under which they participate in the interaction or the duration of an execution.

- **Interaction use** – refers to another interactions.

- **Lifeline** – represents an individual participant of an Interaction.

- **Message** – defines a particular communication between lifelines. The different kind of messages are
  - asynchronous,
  - found as they are received from a non specified source and are modelled as a small black circle at the starting end,
  - lost as they are send to a non specified target and are modelled as a small black circle at the arrow end,
  - synchronous with a call and the corresponding reply.

- **Sequence** – depicts a sequence identified by its name. A Sequence Diagram may contain multiple interacting sequences.

- **Termination** – marks a termination of a lifetime through using a cross at the lifelines end.

The example in Figure 2.10 shows a sequence of pegging a picture. Therein the participants are a person, a hammer, a nail which has to be straight by condition, a thump and a picture. Until the nail is in a steady position (condition on the loop) tries the person to pound it with the hammer. Either the hammer hits the nail as intended or the thumb. In the latter the interaction for medical treatment for the thumb is performed, but is not further specified and just referred. Once the loop is ended the person pegs the picture to enjoy it afterwards. This should give an impression how behaviour of a system is modelled in a Sequence Diagram.

All other members from the group of Interaction diagrams use more or less a similar concept as the one introduced above. They are focusing more on the exact time (Timing Diagram), object communication (Communication Diagram), or combined interaction overall (Interaction Overview Diagram). However, they all are used for modelling during the specification and rather the design phase, because of their appropriate representation of how a system is executed or interacts during execution.
Figure 2.10: Example of a Sequence Diagram [Kec05]
To summarise this section, the UML is a very powerful, though complex language to support software development, especially if used with a tool. Such tools usually provide the very supportive service of code generation. They are grouped under the term of CASE tools, which is the subject of the next section.

**2.3 Tools Supporting Software Engineering**

The subject of this section is to give an overview about technologies, which support the software life cycle as described at the beginning of this chapter (see Section 2.1 and aides the development process. They are accumulated under the expression of Computer-aided Software Engineering CASE tools and include design editors, data dictionaries, compilers, debuggers, system building tools, etc. [Pre01] The advantage of these technologies is their automation of some development process activities. The use of CASE tools can lead to improvements in software quality and productivity.

“... improvement were likely if integrated CASE environments were used. In fact, the actual improvements which have been achieved are of the order of 40 per cent.” [Som01, p. 64]

The CASE technologies can be classified according to the process phase they are specialised for. Table 2.1 presents this classification. CASE tools supporting the early phases up to the design are called upper CASE tools. On the opposite site, lower CASE tools refer to those tools, which support the phases of implementation and testing [Dum03].

Another classification is possible according to how they are integrated to more complex systems, for example:

- **Tools** – support single process tasks such as compilation, file comparison, code editing, etc. They are mostly stand-alone tools.

- **Workbenches** – are composed of tools and support phases or activities such as design, validation, etc.

<table>
<thead>
<tr>
<th>Class of tools for</th>
<th>Specification</th>
<th>Design</th>
<th>Implementation</th>
<th>Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re-engineering</td>
<td></td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td></td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Debugging</td>
<td></td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Program analysis</td>
<td></td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Language-processing</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Method support</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prototyping</td>
<td>×</td>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td>Configuration management</td>
<td>×</td>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td>Change management</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Documentation</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Editing</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Planning</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>
2 Fundamentals of Software Engineering

- **Environments** – support all or at least a substantial part of the development process by integrating several tools or workbenches into a single system. It is not uncommon for them to feature in some way the possibility to add new functionality via plug-ins.

Especially the last group, also known as Integrated Development Environment (IDE), are essential for today's large software development projects to be effective. Typical representatives are the Eclipse Platform of the Eclipse Foundation, NetBeans of Sun Microsystems, C++ Builder of Borland, etc. A more extensive list can be found under [Wik06a].

Still software engineering remains a highly creative work, thus these CASE technologies are not silver bullets to eliminate all difficulties from the development process.
3 Fundamentals of Web-Based Applications

The previous chapter gave an overview of software engineering methods. This chapter is about understanding how Web-based application may look like, respectively their architecture, and what technologies are used to implement such architectures. Because of the broadness of the domain of web applications, this part will concentrate on architectures and technologies which are in the scope of this thesis.

3.1 Web Application Architectures

Before describing any architecture it seems to be appropriate to define the term itself. Though there is no unique definition, the author favours the following to give the intention how the term is understood within this document:

“The architecture, in the sense of Web application development, is the software or hardware related structure of a system to develop, which contains the components of the structure, their extern interfaces and the relationships between the components.” [DLWZ03, p. 141]

An architecture of a system is primary influenced by the functional and system requirements (e.g., operating system or integration of legacy systems).

In general, Web applications are a kind of distributed systems. Some of these distributed systems are based on the client/server model and some are not. The following brief description should give an overview of architectures addressing the distribution of data and messages [KPRR06]:

Peer to Peer (P2P) – describes the direct communication between two participants (the peers) without a server as mediator (e.g., point to point connection) and the way how they discover each other. Thereby the peers are acting equally. Examples include JXTA and Xmiddle.

Distributed Object Middleware (DOM) – is an infrastructure to access remote objects transparently. It is based on the Remote Procedure Call (RPC) mechanism. Some systems under DOM also enable objects to interact beyond their own boundaries with different platforms, e.g., Common Object Request Broker Architecture (CORBA). Further examples include Microsoft’s Distributed Component Object Model (DCOM) or Sun Microsystems’ Enterprise Java Beans (EJB).

Virtual Shared Memory (VSM) – is a technique in which distributed processes access common data. Thereby an appropriate middleware is used to allocate the data, which can be stored “anywhere”. Examples of VSM systems are Corso and Equip.

Message Oriented Middleware (MOM) – offers functionality for communication via asynchronous messaging, which means that the transmission is sent to the receiver regardless of its status. In the case of an unavailable participant MOM ensures that messages are delivered nevertheless. For instance, the Java Messaging Service (JMS) and the Microsoft Message Queue ( MSMQ) represent MOM.
Service Oriented Middleware (SOM) – enhances DOM by the idea of services. In this context it means that a number of objects and their behaviour are made available for other by using a well defined interface. SOM specifies communication protocols between services and provides location- and migration-transparent access to them. Thus it is an approach that overcomes boundaries of platforms. All architectures that emerged within the field of Web services belong to the SOM group. One example of a SOM is Sun’s Jini system.

As mentioned above these architectures principally belong to distributed systems and are not limited to Web-based applications. This is why they are not discussed in detail. Even though some of them are based on the client/server model, the focus is moreover on the common and widely successful approaches of client/server architectures for Web applications. They will be the subject of the next section.

3.1.1 Client/Server Architecture

The concept of this architecture is the separation of system components in two roles based on the request-response principle [Som01]:

Client is a piece of software that sends a request for a particular service to a component.

Server is a piece of software that implements a certain functionality and responds to a request regarding this functionality.

Within the client/server architecture one can distinguish between architecture types based on the point where the client is connected to the server and the following layers [DLWZ03]:

- **Presentation** – to present the data,
- **System control** – to control the workflow of the system,
- **Application control** – to exchange and prepare data,
- **Data management** – to provide functions to access the database,
- **Database** – to store and manage the data,

The Figure 3.1 illustrates the different types. Thereby the network separates the client’s functionality (all units above the network) from the server’s (all units below the network). One speaks of a thin client system if just the presentation layer is implemented by the client and of a fat client in the case, in which the client additionally prepares data or controls the workflow.

3.1.2 Two-Tier Architecture

A client/server architecture consisting of a single server and at least one client is depicted as two-tier architecture. Thereby the network strictly divides the layers of the client from those of the server. The whole interaction takes place through the network, like Figure 3.2 shows. The server includes all necessary functionality, such as database management, application and system control (Business Logic). [KPRR06]

Thus the two-tier architecture is suitable particularly for simple Web applications. Maintaining the server may become a difficult and complex burden once the application reaches a certain level with a higher number of components.
3.1 Web Application Architectures

Figure 3.1: Types of client/server architecture [DLWZ03]

Figure 3.2: Example of a two-tier architecture [KPRR06]
3 Fundamentals of Web-Based Applications

3.1.3 N-Tier Architecture

In Contrast, the N-tier architectural approach is more suitable for larger applications with complex business logic, which are accessed by a large number of concurrent clients. The application is organised in an arbitrary number of layers, typically the following three. [KPRR06]:

- **Presentation tier** – renders the result of a request in the demanded output format.
- **Business tier** – hosts the business logic of an application in an application server. It provides the workflow, the data access, the connection to legacy systems, etc.
- **Data tier** – contains the data and its management, usually a database server with a database and a database management system.

Additionally, mechanisms for the security (e.g. firewalls) or for the caching (e.g., proxies) can be integrated into the request-response flow if needed. Note that the term tier and layer as well as N-tier and three-tier (N-tier and three tier divide the business tier into several layers) are often used synonymously.

The main difference between two- and N-tier architecture is how the latter embeds functionality within the application server component. To make services available to all Web applications they are held in the application server context. For instance, services like customisation respectively personalisation might be used by various Web applications. As another advantage counts that three-tier architectures profit from the load balancing mechanisms of applications servers. Further, so-called connectors are available to integrate external applications or legacy systems. The Figure 3.3 illustrates the N-tier architecture and its components. The N-tier architecture is more complex to design, because of the difficulties to reach a clear separation of the layers.

The described advantages of the three-tier over the two-tier architecture led to its success in large Web-based applications and it is not surprising that a lot of application servers are based on this architecture (e.g., J2EE, WebSphere, .NET). [KPRR06] Some might wonder if there is a three- and a two-tier architecture exists there also a one-tier approach. It exists but the special case of a one-tier architecture, also known as host/terminal system, will not be discussed herein. It is not considered as a client/server system, because the terminal is a simple input/output device without own memory and processor capacity. A terminal is not requesting a service. All workload is done by a monolithic host.

3.2 A Closer Look at the N-Tier Architecture

Due to the relevance of the N-tier architecture for this thesis a more detailed look at its layers and the corresponding implementation technologies will be given in the following. It has to be emphasised that the topic contains an uncounted number of different technologies, pattern, frameworks, etc., of which only a small set will be presented. Of course, the introduced technologies are not exclusive to a N-tier architecture. In fact, some of them have been developed completely independent from the Web. This is the case for technologies in the data tier.

3.2.1 Data Tier

Since the mid 1960’s the efficient and structured storage of large data sets is the domain of databases. The development of relational databases, SQL and Entity-Relationship (ER) in the years from 1970
to 1975 signify undisputed one of the most successful and long living technologies in the history of software [HS97]. Thus database systems are the leading technology to manage data in a Web application.

*It turns out that the vast majority of the world’s corporate data is now stored in relational databases. They are the starting point for every enterprise application with a lifespan that may continue long after the application has faded away.* [KS06, p. 1]

It must be admitted that content management systems are moreover used to manage documents and the whole Internet presence (e.g. layout, configuration, site mapping, versioning, etc.) than that they handle single data [DLWZ03, KPRR06].

According to [CFMS95, p. 3]:

“A database is a collection of mutually correlated data permanently stored on persistent storage supports. . . In a database, data (entities) and entity associations that characterise an organisation are described. Data management functions, usually supplied by an operating system, are extended through a set of programs globally called the Database Management System (DBMS). Great quantities of data can be accessed rapidly and efficiently by a DBMS, which supplies a set of targeted functions, such as schema management, concurrent transaction management, data access control, logging, recovery of the database after system failures.”

The definition also sorts the term Database Management System (DBMS), which builds together with the database the Database System (DBS) [SST97].
Relational Database Model

The relation database model is based on organising data in form of tables (relation). Thereby the following expressions (see Figure 3.4) describe a relation.

- **Attribute** – is a column of a table and by that a name a set of values values.
- **Attribute value** – is a cell of a table, thus an atomic value.
- **Relation** – is the table containing of a set of tuples.
- **Relation name** – names a table.
- **Relation schema** – describes the structure of a relation by its attributes.
- **Tuple** – depicts a row of a table and is a set of attribute values (It is required that tuples are unique within a relation).

The uniqueness of a tuple is also referred as the key and is part of the integrity ensuring [HS97]. One differs into

**Primary key**, which names a attribute or a attribute combination to identify every single tuple of a relation; and into

**Foreign key** which is a reference to a primary key of another relation or sometimes the same relation.

One may notice the similarity to an algebra. This results from the formal way, in which the relation model relies, the so called relational algebra.

The relational algebra enables a flexible, independent modelling by providing a variety of operations and queries that are not bound to the underlying physical features. It formalises the relation schema $R$ as a set attributes $(A_1, A_2, \ldots, A_n)$:

$$ R = \{A_1, A_2, \ldots, A_n\} $$

It formalises the relation $(r)$ as a set of n-tuples $(a_1, a_2, \ldots, a_n)$.

$$ r = \{a_1, a_2, \ldots, a_n\} \quad \text{with} \quad a_1 \in A_1, a_2 \in A_2, \ldots, a_n \in A_n $$
3.2 A Closer Look at the N-Tier Architecture

As an algebra it provides a number of operators, all applying on relations and giving relations as result. The main operators on sets of the relational algebra are (assuming the two relations \( r \) and \( s \)):

- **Union** – applied to \( r \) and \( s \), returns a new relation \( t \) defined as a set of n-tuples belonging to \( r \) or to \( s \):
  \[ t = r \cup s \]

- **Difference** – applied to \( r \) and \( s \), returns a new relation \( t \) defined as a set of n-tuples belonging to \( r \) and not to \( s \):
  \[ t = r - s \]

- **Intersection** – applied to \( r \) and \( s \), returns a new relation \( t \) defined as a set of n-tuples explicitly belonging to \( r \) and to \( s \):
  \[ t = r \cap s \]

- **Cartesian product** – applied to \( r \) and \( s \), returns a new relation \( t \) defined as a set of n-tuples that are a combination of the k-tuples of \( r \) and the m-tuples of \( s \):
  \[ t = r \times s \quad \text{with} \quad n = k \times m \]

- **Projection** – applied to \( r \), returns a new relation \( s \) containing the n-tuple of \( r \) defined on a subset of attributes of \( r \). It implements the vertical decomposition of \( r \):
  \[ \pi_{\text{attribute}_1, \ldots, \text{attribute}_n}(r) = s = \{\text{attribute}_1, \ldots, \text{attribute}_n\}, \quad s \subseteq r \]

- **Selection** – applied to \( r \), returns a new relation \( s \) formed of the n-tuple of \( r \) verifying a predicate on a subset of attributes of \( r \). The predicate is expressed through a formula involving constants, attributes, and comparison operators (one of \(<, \leq, \geq, >, =, \neq\)). Also predicates can be combined by connectors (one of \(\neg, \land, \lor\)). The selection implements the horizontal decomposition of \( r \):
  \[ \sigma_{\text{predicate} \oplus \ldots \oplus \text{predicate}}(r) = s \quad \text{with} \quad \oplus \in \{\neg, \land, \lor\} \]
  \[ \text{and} \quad \text{predicate} = \text{attribute} \otimes \text{constant} \quad \text{with} \quad \otimes \in \{<, \leq, \geq, >, =, \neq\} \]

- **Natural join** – applied to the relations \( r \) (with the attributes \( A \) and \( B \)) and \( s \) (with the attributes \( B \) and \( C \)), such that \( AB \) intersected with \( BC \) results \( C \), returns a new relation \( t \) defined on the attributes \( ABC \). The relation \( t \) consists of the set of n-tuples resulting from the concatenation of k-tuples in \( r \) with the m-tuples in \( s \) that have identical values for the attribute \( C \):
  \[ r \bowtie s = t \quad \text{with} \quad AB(r) \cap BC(s) = B \]
  It is also possible to use predicates in a join. A join, in which the intersection is an empty set, becomes a Cartesian product.

- **Renaming** applied to the relation \( r \) renames an attribute \( A \) to \( A' \). It is necessary, because a join requires equal names of an attribute to be executed:
  \[ \beta_{A \rightarrow A'}(r) \]
One can imagine that such algebra is implemented in a various number of query languages (e.g. SQL, Query By Example (QBE), and QUery Language (QUEL)). The SQL marks one of the most important. It is by far the most widespread [SST97] and will be introduced in the following. Though it is neither intended nor possible to cover the complete standard. For more information about SQL the reader is referred to the appropriate sources (e.g. [SQL06, Pos07, HS97]).

SQL allows the definition of tables with its attributes and the corresponding data type. It is also possible to define a primary and additional foreign keys. This short example should give an imagination of the syntax to define data:

```sql
CREATE TABLE table_1
  ( attribute_1 integer,
    attribute_2 varchar NOT NULL,
    attribute_3 date
  PRIMARY KEY (attribute_1),
  FOREIGN KEY (attribute_2)
    REFERENCES another_table (another_attribute) )
```

The command NOT NULL forces defined values in every tuple of this attribute. This also indicates that every data type contains an undefined value symbolised by NULL. SQL enables a alternation of tables through the ALTER TABLE-command. For example, it supports the renaming of a column, the change of number of columns, the change of referential integrity, etc. The DROP-command drops tables, column, etc.

Beside the definition of relational databases, SQL supports the querying for data. Thereby the basic concept is the so called SELECT-FROM-WHERE-block [Pos07, SQL06], short SFW-block. It is a statement to query a database. The single parts of the block mean:

- **SELECT** – specifies the column(s) of a table.
- **FROM** – specifies the relation(s) to which the block is applied.
- **WHERE** – specifies the condition, which has to be valid to select a tuple.

A statement without a WHERE-clause is analogous to the projection in relational algebra. The WHERE-clause itself would represent a predicate in relational algebra. The clause may also involve several encapsulated SFW-blocks and other operations such as UNION and EXCEPT (difference). Further it is possible to use the JOIN-command or one of its derivate in the FROM-clause. The example below shall illustrate the use of the SFW-block. Therein the column 2 from relation a and 4 from relation b are selected if the tuples have the same value in column 1 and 3, and if the value in column 2 is bigger than the value 5.

```sql
SELECT table_a.column_2, table_b.column_4
FROM table_a, table_b
WHERE table_a.column_1 = table_b.column_3 AND table_a.column_2 > 5
```

The manipulation of data works in a similar way. It uses the commands:

- **DELETE** – deletes tuples from the specified table. An optional condition in WHERE-clause is possible to specify certain tuples.
- **INSERT** – inserts a new tuple or values into the table.
- **UPDATE** – changes the value of cells, also with an optional WHERE-clause.
Nowadays, SQL is supported in many database systems, e.g., DB2, MySQL, PostgreSQL, Sybase, Informix, Microsoft SQL Server.

To design relational databases the ER-model is a proven technique. It uses a graphical notation to draw an abstract model of the database. The Figure 3.5 shows the diagram and its elements in the way it is introduced in [HS97, SST97]. It is based on the concept of three main elements:

- **Entity** – describes an object from the real world or an imagined world, about which information have to be stored. Entities with common characteristics are grouped under an entity type. The entity type is symbolised with a rectangle.

- **Relationship** – describes a relationship between entities. It is is symbolised with a rhombus that is connected to the participating entities. It may posses additional attributes to characterise the relationship. Sometimes a cardinality is specified to express the quantity of a relationship.

- **Attribute** – is a characteristic of an entity or a relationship. It is symbolised with a rectangle with rounded corners. The underlining of an attribute specifies it as a primary key.

The simple, but very formal, concept of the ER-model is a reason for it’s success. Furthermore, the design via ER-models is supported by many software tools, which generate SQL-commands from the ER-diagrams, and execute them to construct the database.

### 3.2.2 Business Tier

The business (or logic) tier implements the logic on a server as described above. Therein different concepts can be applied to handle the tasks.

For example, the interaction with a relational database system (see Section 3.2.1) is often provided through the use of a standard interface such as:

**Open Database Connectivity (ODBC)** – is an interface developed by Microsoft to enable the access to different databases with SQL-commands without the necessity to change the application above. ODBC contains a driver manager, which decouples an application from a certain database driver. The driver manager is responsible for the loading of the appropriate database driver and delegating of all commands towards it. The interface to the driver manager as well as the one to the database driver follow a well defined specification. [SST97]
Java Database Connectivity (JDBC) – is an API from Java with a concept similar to the one of ODBC. It is a standard interface to interact with a SQL-database. Hence it is heavily reliant on the query language. JDBC allows the use of SQL-statements within a Java program [KS06] by offering an abstract and neutral interface. The different database vendors offer a JDBC-driver for their product. This driver respectively the needed packages are dynamically loaded during runtime and handled by a driver manager provided by Java. The manager is then used to control the connections and the interaction with the database. Though using the same concept as ODBC, JDBC is considered to be easier to handle and to have the better clearness [SST97].

Java offers also other APIs, which can be used in the business tier. In fact, Java provides a complete specification for Web-based applications, the Java Platform, Enterprise Edition (Java EE) [JBC+06]. It is a specification of a software architecture not a ready to use software. The Figure 3.6 illustrates the Java EE specification Due to the fact that JavaEE is a very extensive specification only the core components Servlets, Java Server Pages (JSP), and Enterprise Java Beans (EJB) are introduced in the following:
3.2 A Closer Look at the N-Tier Architecture

**Servlets** – are a component technology designed to serve the needs of Web developers who need to respond to Hypertext Transfer Protocol (HTTP) requests by returning dynamic generated content. They are the complement to the Common Gateway Interface (CGI) in the world of Java. Servlets are invoked by a special Uniform Resource Locator (URL) to process incoming requests. The requests are sent from the client and forwarded by the server. Then a Servlet generates Hypertext Markup Language (HTML) response pages on the fly [FC05]. Servlets have to run in a special environment which is fully integrated into the Web server. This environment is the Servlet (or Web) container. Servlets provide a multi-threading capability so they are able to handle many requests concurrently in different threads, though a single thread is sequentially processed. Thereby the sequential steps are:

1. reading all data from the client,
2. looking up the information, which is embedded in the HTTP request,
3. generating result, if required through interaction with an other component, database, or external system,
4. putting the results into a document form,
5. setting response parameter (e.g. cookies, caching parameter, document type, etc.), and
6. sending the result document to the client.

In addition, a large number APIs are available to integrate supplementary of features (e.g., Java Authentication and Authorisation Service (JAAS), Java Architecture for XML Building (JAXB), JavaMail). [JBC ++ 06] Moreover, Servlets offer an elegant way of session tracking by using cookies and the HTTP-Session object to access session data [FC05]. Servlets are mainly used to generate dynamic content, to track the session, and to implement the business logic. Their advantage lies in their procedural style, the clearly encapsulate Java code, whereas the disadvantages are the complexity coming along with the Java technology and the inert reaction (if the Servlet is very complex through the including of many templates) [FC05].

**JSP** – are a Java component technology designed to simplify the programming of dynamic, graphical sophisticated HTML pages. They extend the conventional static document with snippets of Servlet code through special JSP-tags [JBC ++ 06]. It is the attempt to combine the readability of HTML-tags and the strength of the Java API. Since the JSP specification is based on the Servlet specification, JSP support within a Web server provides a translation layer. Individual JSP documents are stored as text files on the server. When the JSP document is first requested the JSP engine uses the JSP file to generate the source code for a Servlet. The generated code is then compiled, installed into the Servlet engine, and used to service the request. Once the JSP has been compiled, the compiled version is saved and used for fulfil additional requests according to the standard Servlet specification [FC05]. This procedure increases the performance of the request-response process. In case of a modification on the JSP file the server detects this change and rebuilds the corresponding Servlet. Figure 3.7 illustrates the JSP processing in a Sequence Diagram. Beside the improvement of the development of dynamic content, the JSP technology aspires a better separation of functional and presentational parts of a system by its tags. The tags represent a certain command to the JSP engine. It is possible to add customer defined tags. Thus, the functionality can be excluded into the custom tags, which allows the division of labour into designer and developer. The first are responsible for the presentation (e.g. HTML elements for layout, navigation, etc.) through the use of tags, the latter for the business logic (custom tag library). Once a tag library is established, it is much likely that it will be reused in many projects. This
makes JSP, in combination with the platform independence of Java, a highly reusable and maintainable technology [KS06]. JSP provide a better abstraction then Servlets, because they look more like a HTML document and less like a program. Their strength is the separation of presentation and functionality. A big drawback is that ‘dirty’ development is possible. It mixes presentation and logic, which makes the code very hard to read and to maintain [KS06].

**EJB** – are a Java component technology designed for distributed business applications. The EJB specification offers a standard model for building server-side components that represent business processes. It is by far the most powerful, though the most complex specification, of the three mentioned here [KS06]. EJB are exclusively responsible for business logic and require a special container on a Web server, the EJB container. The specification describes three kind of Beans:

- **Entity Beans** – provide a generic, object-oriented way to manage persistent data without worrying about the details (e.g. a certain database system) of what’s underneath. This means the data to be stored, along with transaction characteristics, security mechanisms, etc. are described in an Entity Bean. The container, especially the persistence manager, then takes care about translating it into whatever database or other storage mechanism is used [BMH06]. As an alternative way the persistence can be managed by the Bean itself. However, the advantage is that Entity Beans only need to implement simple ‘get’- and ‘set’-methods to retrieve and store data, instead of query statements.

- **Session Beans** – embody a process or a task flow that defines how the Bean interacts with other Beans. If the Session Bean is stateful it receives this state from the client. Stateless Session Beans are completely without any state. Both types of Session Beans are not persistent. Thus they ‘live’ only as long as the client needs them [BMH06]. They realise an interface, over which all clients outside the EJB container interact with the system. This hides the actual implementation of the function from the client and
allows different kinds of clients (e.g. Servlets, JSP, stand alone applications, legacy systems, other EJB, etc.) to use the application [KS06].

- **Message-Driven Beans (MDB)** – process messages asynchronously from systems like the Java Message Service (JMS), legacy systems, and Web Services. Similar to Session Beans, MDB coordinate task flows involving other Beans. They differ just in the way they are accessed. They do not use an implemented interface, that defines which methods can be invoked, but subscribe to or listen for messages. Thus they handle functionality that does not require an immediate response [JBC06].

Since the EJB3.0 specification, which is used in this work, EJB are simple Java classes and interfaces. The classes are extended with annotations, which declare a class or interface as a Bean. Annotations are a language feature that allows structured and typed meta data to be attached directly to the source code. Annotations overcome the onerous handling of additional deployment descriptor files, which contain the meta data in earlier EJB specifications. Deployment descriptors can still be used optionally [KS06]. For example, these meta data carry the information for an entry in the Java Naming and Directory Interface (JNDI). This entry enables all other Beans or Clients to ‘find’ a certain Bean. Assuming that one has developed and deployed a JSP-site, a Stateful Session Bean, and an Entity Bean on a Java EE-server, a client may send a request. The JSP site needs the functionality of a Bean to respond. It looks up the needed Bean in the JNDI and contacts the Bean container to use the Bean interface to invoke the Session Bean’s functionality. Then the Stateful Session Bean is instanced, so that the Session is bound to the client. Since the Session Bean implements the logic and works with Entity Beans, it provides the services to instance an Entity Bean by whether creating a new one or getting one from a underling data source. Afterwards it executes all required operations so that the JSP is able to respond. Figure 3.8 shows this process. The technology of EJB brings the benefits of a very high level of separation between data, logic, and presentation. The concept of the container is a powerful instrument, especially in perspective of load balancing. The container manages the EJB objects and the resources.

Figure 3.8: Example for the interaction between JSP and EJB
for the Beans, and additionally provides services such as transactions, security, concurrency, and naming at runtime. Once EJB are developed, they can be deployed on any Java EE compatible server, even redundant on many servers, and will work out of the box [BMH06]. Thus, they have a big advantage in scalability and portability. They can bridge the gap between relational databases and object-oriented applications via object-relation mapping. Further, EJB provide an excellent way to integrate external tools and legacy systems [FC05]. All these positive points go along with the disadvantage of the complexity. One can easily spend a lot of time to understand and apply EJB correctly. It is also required to use a Java EE compatible server. Today exists a hand full of free servers (e.g. GlassFish from Sun Microsystems, JBoss from JBoss Inc., Geronimo from Apache Software Foundation, JOnAS from the ObjectWeb consortium, etc.). The maintenance of such a server and therefore the use of EJB will cause some extra costs. Like any other technology they have their costs and benefits. EJB should only be used when their benefits outweigh their disadvantages. The Bean concept is suitable especially for large, growing applications that need to interact with various external applications [CK03].

Beside the vast of Java APIs, there exist other solutions for the Web application development respectively for the business tier. For example:

**CGI** – is a standardised interface between Web servers and applications to forward data in an HTTP request to such applications. The application program is addressed by stating a URL in the form tag of the calling HTML page. The parameters from the request are written to and processed by the program. The program creates an output, usually HTML documents. Any programming language available on the server platform can be used, e.g. Perl, C, C++, Python, etc. A major drawback of CGI technology is the limitation in scalability, because an independent process has to be started for each incoming request. This can cause a considerable strain on resources, especially in case of many concurrent users [DLWZ03].

**Server-side scripting** – is a generic term for approaches such as Active Server Pages (ASP) of Microsoft, Hypertext Preprocessor (PHP) of the PHP Group [PHP06], ColdFusion or ColdFusion Markup Language of Macromedia, etc. All of them are script languages allowing the creation of HTML pages on the fly. Therefore a special preprocessor (or script interpreter) is required to be integrated in the Web server. The script source code is embedded within the page source code. A page with an embedded script, usually identified through the special file extension, is forwarded to the preprocessor when a Web client requests it. The preprocessor executes the script, produces and delivers the output HTML page containing the dynamic content. In general script languages are easy to learn, hence very suitable for simple, small applications. Some of them have a lack of standardisation or are not as well structured as the Java technologies. Some may value standardisation very low, but its benefits manifest in calculation of risk, design patterns, ease of communication and interchangeability. Also the clear separation between presentation and logic is left in the duty of the developer. Like CGI, server-side scripting does not go well along scalability issues. [KPRR06] One might argue that JSP also fits into the group of server-side scripting, but the difference is the translation into a compiled Servlet, so that they are not interpreted by a preprocessor.

**ASP.NET** – is a component technology of Microsoft representing the new generation of ASP. ASP.NET provides better separation of presentation, logic and view, and supports Web services and distributed applications. It is improved in performance compared to ASP...
through a precompilation mechanism similar to the one for JSP. It still reaches not the functional amount of Java EE, but is therefore also far from being as complex as it.

In general, the briefly discussed technologies have their benefits and drawbacks. In the end it is in the hand of the developer or the manager to chose the technology, which is most suitable for a certain project.

### 3.2.3 Presentation Tier

The presentation tier is the interface to the user. The presentation is responsible to receive requests from a client, eventually render results, and present them to the user as the response. Thereby some of the above mentioned technologies are used, such as server-side scripts, JSP, CGI. They all provide the possibility to either implement presentation and logic separated, or mixed. For example, a page has to show a table depending on its dynamic content. That means the number of rows and columns is set by parameters, which get their values from an underlying data source. A clear separated design would handle the code to build the table form in one file, and the code to get the data and set the parameters in another one. In a ‘dirty’ design just one file would exist.

Assuming that a developer wants to avoid ‘dirty’ applications, the technologies are reduced to their presentational parts. Thereby are meant all tags, annotations, commands, etc. that serve the presentation of data. This set is extended with technologies that are particularly developed for presentational tasks. Like technologies in the business tier, a lot of concepts have been developed in recent years (e.g. DHTML, Portlets, Ajax, Struts, Applets). Only some of them can be introduced here.

Client-side technologies are left out from the beginning, because of their disadvantages. The performance and the functionality are heavily depending on the client’s software and hardware equipment. The time to load and transfer the required data becomes a problem, especially in larger systems. Many aspects are not under control of the server. This is a problem for high security relevant systems. In contrast, simple client interfaces, e.g. HTML-based, are extremely lightweight, do not require prior installation, and do not suffer from security restriction (e.g. firewalls). In addition, HTML documents can be modified and enhanced at their source without concerning about the client. The deficits weigh much heavier than the possible use of proprietary GUI-Elements, the load relieving of the server, or the including of data from distributed sources in a client application. Thus client-side application are much better suitable for an use in Intranet-system [RV03].

**Java Server Faces (JSF)** – is a Java standard for GUI components. JSF builds on the foundation provided by Java Servlets and JSP. It’s model is based on JSP tags and allows the creation of user interfaces from a set of predefined components through the use of a tag library. The JSF model also includes state management and page navigation facilities that support the development of applications under a N-tier concept [BHR07, FC05]. The components for a presentation (or view) page are sorted into a hierarchical tree structure (e.g. page, form, grid, component). This component tree can be instanced and bound to a state, which allows to save values in the components and their reuse for every single client. To use JSF the deployment descriptor files of the Servlet/JSP have to be extended with the necessary entries. A server then applies the deployment descriptor information and the tags in a JSP file to process a request. This is done in six basic steps:

1. restoring the component tree and its values for the client or instance a new tree,
2. filling the components with values from the request,
3 Fundamentals of Web-Based Applications

3. validating the values via a predefined validator (the view is reloaded with error messages in case of a detected fault),
4. updating the new values in the data model (e.g. calling a setter method of an Entity Bean instance),
5. invoking of any business logic actions that correspond to the current request,
6. rendering a response.

The JSF component formatting is independent from the output response. What a page does look like to a client is left to a special renderer. This allows an easy way to internationalise applications. Altogether, JSF are a step further in the direction of tier separation and ease of Web application development. The features of component tree, value validation, navigation control, instancing of pages for clients, and the internationalisation provide large benefits to developers [BHR07].

Tapestry – is an approach from the Apache Software Foundation that uses HTML templates. These template are normal HTML elements with special attributes that define it as Tapestry element. It runs within a Servlet container. Tapestry elements are embedded in the document and a descriptor file is used to map between the marked HTML elements and the corresponding Java components. In contrast to PHP, ASP, or JSP it does not use special tags. Thus Tapestry documents are bit easier to read, but the additional descriptor files have to be managed.

Like the technologies for the business tier (especially since some are usable in both), a developer team must choose those presentation technologies, which most fit the project goals.
4 Problem Definition

The problem definition and therefore the requirements of the ModelLib framework are the result of discussion within the ModelLib project team as well as the extraction from the paper [MV06] published by Torsten Mähne and Alain Vachoux. Also the prototype described within the paper was a serious source for the requirements, because it already implements some essential features. According to Chapter 2.1.1, where the requirements for software projects are formalised, they are divided into four groups and presented in this section.

4.1 Functional Requirements

The software called ModelLib Framework should provide the service for accessing and storing AMS design models from different locations all over the world. Therefore the preferred way is to develop a web–based application, because it makes the functionality easily accessible via HTTP or Secure Hypertext Transfer Protocol (HTTPS) using a web browser.

The user has to have the opportunity to query directly for a model and its characteristics or to browse for it through a hierarchy of categories (e.g. physical domain, organisational unit and own models). Information stored in the ModelLib (e.g. about interfaces, architectures, testbenches, design languages, and design tools) shall be editable. This includes also the subjoin of these information.

The up- and downloading of additional files or documents associated with a model has to be a feature of the software. Furthermore committing models to the application and checking them out is a basic function of ModelLib. The committing of a new model or the editing of information are sensitive operations. That is why an automatic or at least semi automatic review process is needed, that validates the submission to prevent data from being inconsistent in a global perspective. A version management is also required as part of the framework to keep track of the modifications over time of the models. This allows designers to efficiently work together in teams on the same task, even if they are divided through long distances. Due to the team orientated character of design projects, there is a need for a platform to discuss issues regarding the AMS design process, models, and documents or the ModelLib itself.

Models and documents are part of the Intellectual Property (IP) of a designer or company. They maybe freely available or underly restricting license terms. This coercively requires an effective authentication and access control mechanism limiting the user access only to the information, which explicitly was made available to him or a group he belongs to. That means the security of IP is the most important concern in perspective of user acceptance and therewith the success of the project. No company or designer will risk its trade secrets and market profit just to accelerate their design process.

User and software communicate through a GUI. This also includes an online help. Besides the direct user-ModelLib communication, there is a perspective to provide EDA tools a direct communication with the model library to carry out advanced tasks, like selecting a suitable model
for a specific design task. Therefore the foundations (communication concept, basic interface) for an API have to be created.

4.2 Quality Requirements

For the quality of the software a few points are important. To ensure the usability of the ModelLib framework, an intuitive and easy to handle GUI is necessary and likewise an online help. It must be noted that especially the intuitive and easy to handle GUI can only be verified fully by the user. During the development the software engineer is only able to keep as close to ergonomic GUI standards as possible.

Reliability in the case of ModelLib aims at stability and running for an undefined time. Wrong, incomplete, or inconsistent values or data shall not lead to a system failure or worse a system crash. ModelLib should be available around the clock every day.

The response time to a request is the essential efficiency requirement of this project. It should be less than 3 s if the connection capacity is high enough between client and server.

A good comprehensive documentation of the application source code is indispensable for maintenance. Beside the source code documentation there has to be handed out a dedicated user guide. A modular architecture has to be chosen to ensure changeability, extensibility, and portability of the ModelLib framework in future times.

4.3 System Requirements

The most important requirement is the open source nature of the project. The software itself shall be made available as open source. It is intended that everyone should be able to download and enhance ModelLib for his own use, by using the same technology the creators did. That is why only open source tools are allowed to be used during the development.

Another requirement for the ModelLib framework is its platform independence. But that is already ensured through the access via HTTPS and a web browser. This implies that the ModelLib framework has to serve its information through a web server.

4.4 Process Requirements

The only things to mention regarding the process requirements are the date of delivery of the ModelLib framework as described above, which shall be January 31st 2007. The work will be done by a single programmer supervised and supported by a project manager.

The Use Case Diagram (section 2.2.1) shown in figure 4.1 summarises some of the requirements above.
4.4 Process Requirements

Figure 4.1: ModelLib Use Case Diagram
4 Problem Definition
5 Requirements Analysis and Specification

According to Section 2.1.2, the requirements analysis validates the requirements regarding their correctness, completeness, conformity, consistency and feasibility likewise if they are realistic, need by the customer and verifiable. The specification is a the detailed description of the system about to be developed, though in some case this description is shifted to the design phase.

5.1 Validation of the Requirements

The validation was done by the use of the first ModelLib prototype and interviews with the client. Especially the interviews lead to the agreement that the functional requirements are correct, conform, consistent, realistic and needed in the way they are drawn up in the previous chapter. Since the supervisor of the project is also one of the future costumers, the priority of complete and highly detailed requirements is not given. That means during the design the project manager will influence the system realisation to complete the functionality.

Investigations on similar problems and their solution as well as dealing with the theory of software engineering (Chapter 2 and 3) enables the author to conclude the feasibility of the project via analogy method. Moreover the existing prototype proofs that the requirements are feasible. It has shown that storing information and browsing them by a client communicating via HTTPS with a server is possible. Also the versionised document or file uploading and downloading can be provided by existing software solutions as well as a discussion platform.

Furthermore it is obvious that a Web-based system can be made available around the clock on every day for a various number of costumers. A careful design and testing of the developed software can assure that it works properly. The writing of documentation is more a concern of the discipline of the developer than a problem of feasibility. The response time is more a recommended than a precise value. It is not sure that this requirement will be met, but the system should aim this time span.

Experience, investigation, and the prototype have led to the opinion that enough adequate open source solutions exist on the market to fulfil the requirement to use such technologies during the development of ModelLib and for the ModelLib platform itself. Especially for free open source application server and database systems some products are available on the Internet that work well for many professional and non-professional applications.

As mentioned before the usability and an easy handling of the system depend heavily on the GUI. To fulfil the corresponding requirements the development has to orientate and follow reliable proven standards in this domain.

The aspect regarding an interface that enables a communication with extern (mainly EDA) tools, are yet not detailed enough to give a statement. It seems to be possible, but it can not be cleared out how and how much effort it will take to provide such service. An analogue conclusion must be drawn in case of the required elaborate access control mechanism and the semiautomatic review process. Both are for sure feasible, but its effort is uncertain.
5 Requirements Analysis and Specification

Table 5.1: Characteristics of Web applications and the relation to this project

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Fulfilling</th>
<th>Not fulfilling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content driven</td>
<td>Content from the database</td>
<td></td>
</tr>
<tr>
<td>Frequently changing content</td>
<td>Form of data is stable</td>
<td></td>
</tr>
<tr>
<td>Multi-media content</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-linearity of presentation</td>
<td>Hypertext</td>
<td></td>
</tr>
<tr>
<td>Non-linearity of usage</td>
<td>Browsing</td>
<td></td>
</tr>
<tr>
<td>High number of potential user</td>
<td>Naturally</td>
<td></td>
</tr>
<tr>
<td>Users with different social, cultural background</td>
<td>Primarily engineers</td>
<td></td>
</tr>
<tr>
<td>Different intention, skills of the users</td>
<td>As required</td>
<td></td>
</tr>
<tr>
<td>Demand of service at any time</td>
<td>As required</td>
<td></td>
</tr>
<tr>
<td>Multi-disciplinary development team</td>
<td>Single developer</td>
<td></td>
</tr>
<tr>
<td>Open source products</td>
<td>As required</td>
<td></td>
</tr>
<tr>
<td>Integration of legacy systems</td>
<td>Components of the prototype</td>
<td></td>
</tr>
<tr>
<td>Volatility of technology</td>
<td>Not in this version</td>
<td></td>
</tr>
</tbody>
</table>

In perspective of the process requirements, explicit the small development team, it is scarcely conceivable if all requirements are fulfilled by the date set in the corresponding section (see 4.4). Thus it can not be determined which functionality or component of the system will be finished. Therefore the development team set priorities for the tasks.

Before continuing it should be described in which way this project fits the peculiarities (see Chapter 2) of Web application development and in which way not. The Table 5.1 illustrates this. Thereby it is noticeable that important peculiarities like frequently changing content (the content is growing, but the form of storing and presenting it is stable), multi-media content or the high variance of user characteristics are not relevant to ModelLib. Hence a more classic oriented process has been chosen.

Further, at this point it seems to be right to argue for an appropriate software development process model guiding this thesis project. As explained earlier, (see Section 2.1.7) it is strongly recommended to follow a well defined process to reach a quality product. The already existing prototype and the fact that this is a Web application development lead to the conclusion that (rapid) prototyping or the incremental software development are well fitting approaches. In a global view at the whole ModelLib project, the incremental model is most suitable. The technology, on which the prototype is based is different from the one for the version developed during this thesis (reasons are described in Chapter 6). In addition, the prototype (except the database) was an ad hoc development with minimalist modelling. The requirements got more and more demanding during the time, because more and more people contributed their ideas. These three points demand an extensive revise in a new iteration. Due to the fact that the success of ModelLib is heavily bound to the quality and security of the stored data, the team decided, that it will profit more from the thoroughness of the incremental model then from the quickness of prototyping. First all requirements are formalised and then step by step implemented regarding their priority. The further chapters will evince this decision.
5.2 Specification

As already explained in Section 2.1.2 the detailed modelling of the system under some reasonable circumstances is shifted to the design phase. This is done in the development project accompanying this thesis, because the chosen architecture and moreover the technology have a big influence on the system. Thus it does not make sense to develop technology independently. In the following it will be explained what architecture and which technology has been chosen for the system.

5.2.1 Architecture Decision

Since the beginning of the ModelLib project it was intended to develop a Web-based system. The requirements (see Chapter 4) reflect this fact. Thus there was no alternate choice beside a client/server architecture. The question was whether to choose a fat or a thin client. It was decided to build on a thin client architecture for the following reasons:

- Reaching a maximum of usability through a maximum of independence from operating system, browser, etc. on the side of the client,
- A minimum of effort to install software to make ModelLib available for the client,
- The idea of a central library,
- Enabling a central maintenance and evolution of the ModelLib system,
- Enabling a central management of documents and files in different versions,
- Security aspects.

Especially the security and independence have to be taken into account. Considering that some users belong to a company, it might be very difficult for them to install ModelLib components in a fat client scenario, because many companies have contracts about which software is granted to use within the organisation.

The prototype as developed and introduced in [MV06] already set an approach for an architecture. It is shown in Figure 5.1. Therein the user has the Subversion client to check files out or commit them to the Subversion file repository (see [CSFP04] for more information), where the source code for an AMS design model, the testbench files, and additional documents are stored. The client uses a Web browser to communicate via HTTP with the server. WebSVN is a server-side component that allows browsing the Subversion repository content through HTML-documents. The client either uses this or WebDAV, which is an extension of the HTTP designed to turn the Web into a read/write medium, in combination with Subversion. WebSVN counts to the user Web interface as well as the ModelLib application core and the YaWiki [Jon05] component. The latter is intended as a platform to discuss and jointly develop the documentation or HOWTOs of the models. The application core contains a number of PHP-files, which control the application and provide the access to the PostgreSQL [Pos07] database server. Therein the meta information about models are stored and managed in one database and the textual data from YaWiki in another. All together the ModelLib server as a whole provides access to or retrieves information from the user created documents.

Though the Three-tier architecture of the prototype is a good solution, the requirements make a more clearly separated N-tier architecture look more suitable. Especially the communication with external EDA-tools votes for a more precise business tier. Also the better possibility to develop and improve the tiers independent from each other count for this approach, particular in perspective of

47
5 Requirements Analysis and Specification

Figure 5.1: Architecture of the ModelLib prototype [MV06]

the stepwise development of the project. Considering that ModelLib starts with a few users and than grows, the advantage of scalability and load balancing is also valuable for the future. An exact plan of the architecture chosen for ModelLib will be given at the end of this chapter.

5.2.2 Technology Decision

The named architectures and technologies in Chapter 3 give an overview of what is applicable to implement a N-tier architecture. At the start of the thesis project it was discussed between developer and supervisor if the prototype technologies are capable enough for the next version of ModelLib. The results are presented in this section.

Technology in the Data Tier

Investigations (e.g. [WE06, Hor06, Pos07, MyS06]) and the experience, made with the prototype, led to the consensus that the PostgreSQL database in version 8.1 is the best fitting free relational DBS for Linux. Particularly in comparison to MySQL it is more qualified for the usage in applications serving many concurrent users and complex queries [Hor06].

The Subversion file repository is also kept. It works well in the prototype, hence there is no reason to burden a migration to another system. The only drawback brings the required client-side installation of the Subversion client, which eases the management of working copies in the repository. It is an optional feature. All files in repository can be retrieved with any Web Browser using HTTP. ModelLib will still be partially usable, if someone does not install the Subversion client.

Since the two third party application components (a.k.a. Commercial of-the-shelf) are kept, any eventually new technology in the other tiers must be compatible to them.
Table 5.2: Evaluation of technologies in the business tier

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Java EE</th>
<th>ASP.NET</th>
<th>PHP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compatibility to the Commercial</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>of-the-shelf</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Open source</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Platform independence</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Standardisation</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Separation of tiers</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Integration of external systems</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Evolution of the technology</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Scalability</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Performance</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Possibility of customisation</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Learnability</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Technology in the Business Tier

It shall be described how some of the technologies in Section 3.2.2, have been validated in perspective of the project goals. Thereby it was just ranked in which way technologies are positioned to each other in perspective of a project goal. It was not a purpose to validate how big the difference is. Further, the validation is limited to Java EE, ASP.NET, and PHP as representative of script languages, because it is used in the prototype. The criteria and the ranking is shown in Table 5.2.

Of course, the **compatibility to the Commercial of-the-shelf** is the highest in case of PHP, because they already run together in the prototype. PostgreSQL is running on Linux platform and so does Java EE. In contrast, ASP.NET is running only on Microsoft platforms, and though PostgreSQL is available for Windows, it would cost some time to migrate it.

Java EE is equally ranked with PHP in the category of **open source**. ASP.NET is free, but not open source.

The **platform independence** shows the same results. Java EE as well as PHP run on multiple servers and on multiple platforms. ASP.NET is restricted to Microsoft platform products.

In the field of **standardisation** are Java EE and ASP.NET leading against PHP. The Java specification with their APIs and component structure is easily outclassing PHP. The latter has a bit the problem of uncontrolled growth, which led to a vast of predefined functions and methods. They are often overlapping and make it difficult for developers to use them efficiently. Also the way atomic data types are implemented and work are different from version to version, which makes it hard to update older applications [Wik06b].

For a **separation of tiers**, Java EE is the best, because the structure of JSF, JSP, Servlets, and EJB is innately separating layers of an application. ASP.NET is also structured, but the components are not as well defined as in Java.

The technology of Java EE features the **integration of external systems** in the best way of the three, followed by ASP.NET and than PHP, which provides almost no mechanisms for integration.

Regarding the **evolution of the technology**, Java is considered to be the leader, because of its standardisation combined with its open source character. In the past, it has been fast in adapting new approaches whenever they came up with success. PHP and ASP.NET are behind, though the Microsoft product is better in this category than PHP.
5 Requirements Analysis and Specification

Also in scalability ASP.NET is much better than script languages, while Java EE is dominating both. EJB can be distributed redundant on many server, without any changes for clients or Servlets/JSP.

The same must be said about load balancing.

In the field of performance is ASP.NET ranking first, because the ASP are precompiled directly into DLL files, which are faster than the byte code of Java, that is interpreted and executed at runtime. PHP is slowest of the three, because it is just interpreted for every request.

The possibility of customisation, for example internationalisation is hard to implement with script languages, while ASP.NET and JSF provide such service.

The learnability of PHP is by far the best, while the complexity of Java EE makes it the hardest to learn.

The most influence on the decision about a future technology had the compatibility to Subversion and PostgreSQL, the open source character, the platform independence, the integration of external tools, and the learnability. Java EE turned out to be the most suitable technology, despite its complexity.

Technology in the Presentation Tier

Since Java EE, or correctly EJB have been chosen to implement the business tier, the technologies in the presentation tier are also Java EE components. It is JSP with or without JSF. However, JSF initially was a bit overlooked, because its features are not very important in the current project phase. That is why this thesis will not further discuss JSF. They can be added in the next version of ModelLib.

To use the Java EE specification with all newest APIs the Java EE 5 SDK was downloaded from http://java.sun.com/javaee/downloads/index.jsp [Java06] and installed on a Debian Linux machine. The Java EE 5 SDK also contains all necessary files to install and run the Sun Java EE 5 Application Server PE 9, better known as GlassFish.

Figure 5.2 shows the designated architecture of ModelLib with the chosen technologies. The YaWiki platform for discussion is planned to be substituted with a corresponding Java-based solution to run within the application server. This will allow the reuse of the Wiki-engine to format the free-text, which will still be stored in the same PostgreSQL database. To keep the WebSVN component, it is necessary to keep the Apache 2 server. Apache and GlassFish run together on the same machine. GlassFish is fronted with the Apache 2/Tomcat connector. An incoming request will be forwarded to GlassFish. There it is linked to the Apache server, if the request is addressing the WebSVN module. All other points regarding the Subversion repository, and therefore the storing of the file revisions of documents, remain unchanged. A client accesses the ModelLib application with his Web browser via HTTP, respectively HTTPS. Confirming to the intention of the Java EE specification, the client interacts with the system through the presentation tier, which is build of JSP/Servlets. The workflow and the handling of data are managed by EJB. They are connected via JDBC to the underlying PostgreSQL database, which is kept without changes.

The Chapter at hand explained how the decision for an appropriate architecture and the use of Java EE as the core technology for the ModelLib platform was made. With the result of this process, some requirements have already been met. For instance, the GlassFish server provides access via HTTPS and is potentially available all around the clock. It is platform independent. The other components (PostgreSQL, Subversion, Wiki) are open source, as well as the Application server itself. The checking out and the committing of files is also supported as it was with the prototype.

The next chapter describes the further realisation of the requirements.
Figure 5.2: Architecture of ModelLib with the new technology
6 Design

As explained in Section 2.1.3 the design is the detailed modelling of a system concerning a certain architecture. At first the focus will be set on the data tier. It shall be explained which problems emerged during the realisation and how they have been solved, especially in perspective to the protection of IP. This is followed by the design of the business and the presentation tier.

6.1 Design of the Data Tier

A central aspect of the ModelLib platform is its function as a library that contains meta information describing the properties of models. The model might be part of the IP of a designer or a company. Thus, two points have to be emphasised addressing the data tier:

- an appropriate way to store meta information,
- the protection of these information.

Both are reflected in the requirements (see Section 4.1).

For the first point, the prototype contributed a good foundation. The relation schemas of the ModelLib database, in which the meta information is stored is well structured, so that only a few changes have been made, summarised as follows:

- Out of 19 original entities 11 have been changed (added/deleted attributes or relationships) and 1 entity have been removed.
- 9 new entities have been added.

Regarding the second point, the prototype worked as follows. A user had to authenticate himself to the system. The authentication is the processing of validation of the user who is trying to access a secured system. The server verifies that the user actually exists and has provided the correct password. Once a user is authenticated the authorisation is determining whether he is allowed to execute a certain function or has access to a certain data. Due to the intended simplicity of the prototype, it was just used what the technologies (Apache 2 server, YaWiki, PostgreSQL) already provided for authentication and access control. The access to the Subversion repository is controlled by the Apache server. YaWiki is using its own access control management. The access to the core application of ModelLib, respectively the provided data, is controlled by the use of PostgreSQL’s user management. The fact that ModelLib’s three main components handle their access control independently from each other is cumbersome to handle, but does not directly mean an implicit security risk.

It is assumed that the common approach of authentication with account (identity) and encrypted password (credential) is secure enough, because it is successfully used in many systems (e.g., Linux operating systems, Windows operating systems, eBay online auction, GMX Internet portal, etc.) [Rö05]. The authentication takes place on the application level, which means that a client is usually accessing ModelLib from a remote place via HTTPS connection to the server,
and is authenticating himself before he is able enter restricted areas. This mechanism works well for the Subversion repository and the YaWiki, because an authorisation is always granted or denied to a file/folder (Subversion) or a page (YaWiki). In case of the core application, which provides access to the meta information database, it is a major lack of security and can cause some serious damage. The reasons will be examined below, but before this the database structure is introduced in the following section.

6.1.1 The ModelLib Meta Information Database

To understand how the data in the database is secured it is important to give a complete overview about the structure of the database. The Figures 6.1, 6.2, and 6.3 show the relation schema in the notation of an Entity Relationship (ER) Diagram (see Section 3.2.1). It must be noted that a not specified cardinality indicates a \([0, \ast]\) relationship cardinality.

The entities describe meta information of an AMS design model as follows:

* **ModelClass** – stores a domain to which an interface is sorted, e.g., physical domain, organisational unit. The model class can be hierarchically structured (e.g., in Company, Section, Group).

* **Keyword** – stores keywords, so that they can be associated with entities and used to search for models.

* **Interface** – stores information about the external view of the model, i.e., the information that is made available to users of the model. Information includes ports, parameters, and assertions.

* **Parameter** – stores information about a model parameter. Parameters make a model more generic or universal as the same behaviour or structure may be adapted according to actual parameter values (e.g., an amplifier gain or a number of bits for logic operands).

* **Port** – stores information about the way the model exchanges data with its environment. A port may convey signals of various types depending of the level of abstraction of the model (e.g., logic signals or continuous-time waveforms).

* **PortDirection** – stores the direction in which the data flows through a port (e.g., input, output, bidirectional, or none).

* **PortClass** – stores the class which a port belongs to. The class is related to the mode of operation of the modelled component and to the level of abstraction of the model, e.g. digital/logic, analogue/conservative, or analogue/non-conservative/signal-flow.

* **Assertion** – stores facts about the model that must be true in order to use the model correctly (e.g., the amplifier gain must be kept within some range to allow a linear I/O relationship).

* **Severity** – stores the level of severity of an assertion, e.g., in increasing order of severity: note, warning, error, fatal.

* **Architecture** – stores information about the internal view of the model, that is how the behaviour and/or the structure of the modelled component is described.
Figure 6.1: ER Diagram of the ModelLib database (structural meta information)
Figure 6.2: ER Diagram of the ModelLib database (document meta information)
6.1 Design of the Data Tier

Figure 6.3: ER Diagram of the ModelLib database (user meta information)
• **DesignEntity** – stores information on a possible/legal combination of an interface and an architecture. There may be several architectures associated to the same interface, so each possible interface/architecture combination makes a distinct design entity.

• **DesignLanguage** – stores information about the design or modelling language used to describe a design entity. Examples of design languages are: SPICE, SystemC, VHDL-AMS).

• **DesignLanguageVersion** – stores information about the version of the design language used to describe a design entity. The version may be, for example, a single number or a year.

• **Vendor** – stores information about the provider of a design tool.

• **DesignTool** – stores information about a tool that has been used to process the model. Examples of tools are simulators such as ADVance MS and Modelsim from Mentor Graphics or SMASH from Dolphin Integration.

• **DesignToolVersion** – stores information about the version of the design tool used to process a model.

• **File** – stores information about a file that contains a full model written in some design language or part of it.

• **Document** – stores information about documentation associated to a model (e.g., application note, user’s guide).

• **DocumentType** – stores information about types of documents, so that it defines which attributes (e.g., year, chapter, ISBN, URL, etc.) a document (e.g., book, paper, manual, etc.) must contain.

• **Author** – stores information about the author of a document.

• **UserDirectory** – stores information about the user of ModelLib (e.g., name, affiliation, phone number, email address).

• **Message** – stores a comment, which is associated with a design entity, document, design tool version, or design language version (e.g., how helpful a document is, or how a design entity worked).

• **ModelClassACL** – stores the rights a certain role is given to a certain model class.

• **InterfaceACL** – stores the rights a certain role is given to a certain interface.

• **ArchitectureACL** – stores the rights a certain role is given to a certain architecture.

• **DesignEntityACL** – stores the rights a certain role is given to a certain design entity.

• **FileACL** – stores the rights a certain role is given to a certain file.

• **DocumentACL** – stores the rights a certain role is given to a certain document.

This brief description should be enough to give an overview about the entities and their meaning to the ModelLib library.
6.1 Design of the Data Tier

6.1.2 The Problem of Tuple-Wise Access Control

The library character of the ModelLib platform requires an uniform structure to store meta information about AMS design models. The tuples in the tables of the ModelLib database contain these meta information. Thereby a table contains tuples with public information as well as highly sensitive information.

For example, a company has developed a secret new interface with ports and parameters. These information are stored in the ModelLib database and shall be only accessible to the members of the company. ModelLib also contains information about common interfaces and their ports and parameters, which are far from being a secret and thus accessible for everyone. Both, the secret and the public information are stored in the same table. Assuming the application controls which information are presented to the user and which not, the problem is as follows. A user authenticates himself to the application. Since the user accounts and the passwords are stored and managed by PostgreSQL, the application connects to the database server and forwards the users account, password, and the query for data. The database management system verifies the user and sends the query results to the application, which is then verifying the user as no member of the company, and thus only showing the public information.

The application filters the data. The database management system is not aware of which tuple a user is allowed to see. It is only able to give rights to complete tables. If a user is valid and granted to access a table, he will be able to see all tuples of this table.

This makes it possible that any user can read and change every information. He just has to find the database server, what is not very difficult, and than use his own valid account and password to connect to the database and to gain full access.

To deny access to the table is not a practical solution, because all users need access to the public model information. Also, it is not a solution to create a different table for every user, because this would be impossible to manage once ModelLib reaches a certain number of users.

This explains why it is highly needed to find a way to protect every single tuple of every single table and to prevent the bypassing of such a mechanism. Therefore the following possibilities exists:

- Excluding the account and password management from the database, so that a user is no longer able to connect directly to the database.

- Shield the tuples inside the database to protect them.

It would not be efficient to reimplement the basic functionality of account management, which comes with PostgreSQL for free. Especially since a DBS would be the best solution to handle the account data. Another argument against the first approach is, that the account management in the database is much more capable to ensure data integrity. The strongest argument for the implementation of the access control in the database is, that it can be kept even if the rest of the platform is modified or completely replaced. That is why the development of a mechanism to control the access tuple-wise inside the database is chosen. Additionally, this eases the development of connectors for extern tools, which cloud communicate directly with the database, while still been put under access control.

The thesis of Frank Hertel [Her04] describes a similar problem. Two of the four solutions explained there fit into the first possibility (excluding access control), which makes them inappropriate for ModelLib. The other two are based on a feature of the commercial Oracle9i DBS used in Mr. Hertel’s project. This feature allows to declare access privileges for single tuple of a table.
6 Design

Figure 6.4: General concept of access control in ModelLib

PostgreSQL neither provides such a feature nor are the ModelLib project members able or willing to buy an Oracle9i DBS.

6.1.3 The General Solution for Tuple-Wise Access Control

As mentioned in the previous section, the general solution of the tuple-wise access control is a mechanism inside the ModelLib meta information database. Figure 6.4 illustrates this concept. The two levels are defined as follows:

**Application level access control** – is a set of mechanisms, implemented in the application, responsible to verify, which functions in the ModelLib platform is accessible by a certain user.

**Tuple level access control** – is a set of mechanisms, implemented in the database, to verify, which tuple of a table is accessible by a certain user.

The question is, how to realise such a tuple-wise access control in a relational database? Before this question can be answered, it is necessary to introduce the concept of a view in a relational database:

**View** – is a virtual (not materialised) relation containing values, which are persistently stored in one or more real relations. Views are defined through a query statement, which is executed in the moment of its call. This means that the values in a view are composed on demand. Views are also described as named/stored queries, because the definition of a view is a query, which is stored under the name of the view. [HS97]

To come to the solution, one could think of two tables/relations:

- The first relation is \( r \). It contains the attributes \( ID \) as the primary key, \( A_1 \), and \( A_2 \). It symbolises the data, which shall be protected.

\[
  r = \{id, a_1, a_2\} \quad \text{with} \quad id \in ID, a_1 \in A_1, a_2 \in A_2
\]
6.1 Design of the Data Tier

The second relation is \( acl \). It contains the attribute \( USER \) as the name of a user and \( REF \) as the foreign key referring to \( r \) plus the attribute \( G \) as a boolean value representing whether a granted (true) or a denied (false) access. The attributes \( USER \) and \( REF \) are defined as primary key for \( acl \). This relation symbolises the access control list for the table \( r \).

\[
alcl = \{ \text{user}, \text{ref}, g \} \quad \text{with} \quad \text{user} \in \text{USER}, \text{ref} \in \text{REF}, g \in G = \{ \text{true}, \text{false} \}
\]

If one selects all tuples from \( acl \) where the value of the attribute \( USER \) equals the current user and the value of the attribute \( G \) equals true, he will receive the relation \( acl' \), which contains all tuples where the current user has a granted right.

\[
\sigma_{\text{USER}='\text{current user'} \land G='\text{true'}}(r) = acl'
\]

The join of \( r \) and \( acl' \), of course after the renaming the attribute \( REF \) to \( ID \), produces the relation \( s \).

\[
r \bowtie \beta_{ID \leftarrow \text{REF}}(acl') = s \quad \text{with} \quad s = \{ id, a_1, a_2, \text{user}, g \}
\]

The projection of the attributes \( ID, A_1, \) and \( A_2 \) of the relation \( s \) produces a relation \( t \), which contains all tuple from \( r \) that the current user is granted to see.

\[
\pi_{ID, A_1, A_2} = t \quad \text{with} \quad t = \{ id, a_1, a_2 \}
\]

A view that is defined as the stepwise composed relation \( t \) returns only the tuple from \( r \) the user, who is currently querying the view, is granted to see. The following steps and the example in Figure 6.5 illustrate these composition of the view without the use of relational algebra:

1. Selection of all tuples form \( acl \) where the value of \( USER \) equals the current user (in the example it is ‘James’) and the value of \( G \) equals true.

2. The join of the result with the table \( r \).

3. The projection of the three columns \( ID, A_1, \) and \( A_2 \) from the result of the join.

The reason that this works is the concept of the view. Since definition of the view is a query executed at the moment of its call, it will always return the tuple of the protected table the current
user is granted to see due to the corresponding entries in the access control table. This is the first part of the problem solution.

The second part is the setting of global access rights to tables. The specification of SQL provides the possibility to set privileges for accessing a complete table to every user independently. This is also possible for a view. An administrator, responsible for the developed database, is the only one, who is allowed to access the tables (in the little example \( r \) and \( acl \)). He defines a view as above and afterwards grants access for all users to this view. As the result, a user is allowed to access the view, that is per definition showing only the tuples from the basic table, which the current user is granted to see. The view is accessing the corresponding tables using the rights of the one who has defined the view, the administrator. Whenever a user is trying to access a table directly, the DBMS will deny the access.

To summarise, the general solution for a tuple-wise access control inside the database, that is founded on the account management provided with DBS, is the concept of:

- One table containing sensible data,
- Another one containing the rights for tuples,
- A view joining the two tables,
- Granting rights for the view, but
- Denying rights for the tables,

It uses features that are part of SQL and therefore implemented in every relational database, which is compatible to SQL. It remains the subject of the next section to explain, how the developed solution is applied to the ModelLib meta information database. This is much more difficult then simply joining two tables, because problems with recursive dependencies, a limited number of joins, the inheritance of rights, and the use of internal schema information occurred during the realisation.

### 6.1.4 The Special Solution for Tuple-Wise Access Control in ModelLib

Figure 6.3 shows that not every entity is related to an access control table. Only six entities (\( ModelClass \), \( Interface \), \( Architecture \), \( DesignEntity \), \( File \), \( Document \)) have a direct table to control a tuple-wise access. Most other entities depend on them or do not need a special access control. At first it is described how such a table is structured and why.

#### Structure of a Table for the Access Control

The ER Diagram shows the structure of the tables containing access information. The term of an Access Control List (ACL) is used, because they are based on the similar concept in modern file systems (e.g., NTFS, XFS). They possess an attribute \(*ACLId*\) (the ‘*’ is a place holder for the actual entity name) as a primary key. This is chosen due to uniformity reasons with other entities, though the combination of the \(*Id*\) and the \( Rolename \) attribute would also be a valid primary key, because it is and must be unique.

The \(*Id*\) attribute is a foreign key referring to the primary key of the corresponding entity. This ensures data integrity, because it guarantees that the access control is actually referring to an existing tuple.
The *Rolename* column contains the role, for which a right to a tuple is specified. It might be helpful to describe the role concept as it is implemented in PostgreSQL 8.1. Where a

**Role** – can be a single user or a group of users. Roles can own objects of a database (e.g., tables, views) and assign rights to those object (not tuple) to other roles. Furthermore, it is possible to inherit rights from one role to another by making the latter a member of the first. [Pos07]

For example, there are the roles *James*, *Michael*, and *Vince* representing real users. There is also a role *Admin* representing a superuser, who owns the objects. If the role *Admin* is granted to the role *James*, *James* would have all the rights *Admin* has. If than *James* is granted to *Michael*, he would inherit the rights of *James*. Thereby, it is possible to specify whether *Michael* also inherits indirectly the rights of *Admin* or only the rights of *James*. *James* can also be granted to *Vince*, because a role can be set to multiple other roles. The roles and their passwords are managed by PostgreSQL as well as the access to objects.

The other three columns of the *ACL* contain the actual rights to execute functions, which manipulate the data of a table. This is similar to read/write access specified in an ACL for a file system. According to Section 3.2.1 these functions, respectively their rights, are:

- **SelectRight** – specifies whether a role is granted to see a tuple or not.
- **UpdateRight** – specifies whether a role is granted to change existing values of a tuple or not.
- **DeleteRight** – specifies whether a role is granted to remove a tuple or not.

If is not necessary to specify a right for the execution of the *INSERT* statement, because adding a new row does not harm any existing tuple.

A right is specified through the use of the boolean data type. The special case that SQL requires a ‘unknown’ or *null* value for every data type is a lucky fact and very important, because it enables a three state standard data type. A right can be either:

- **true** – for a grant,
- **null** – for an unspecified right, or
- **false** – for a deny.

Basically, an unspecified right means that a role is not able to execute the corresponding function until the membership to another role (which has a grant on that function) allows it. In contrast, a denied right means, that the role and all of its members are never allowed to execute this function. Hence, a deny is more powerful then a grant, which is more powerful then an unspecified right:

\[
null < true < false
\]

The following example is used to illustrate the functionality of the *ACL* tables a bit more compressed. There are a table *Document* containing the documents with the *ID* 1 to 5 and the table *DocumentACL* containing roles and rights as shown in Figure 6.6. Both, *James* and *Michael* are member of a company and of the role of the same name. Both inherit the rights of *Company*, so that the rights for every role results as follows:

- **Company** is allowed to select the tuples with the *Id* 1 and 2 from the *Document* table, but the role has no further rights. A tuple in *Document*, which has no corresponding entry for the role in *DocumentACL* is the same as all rights are *null*.
Design

<table>
<thead>
<tr>
<th>DocumentACLId</th>
<th>DocumentId</th>
<th>Rolename</th>
<th>SelectRight</th>
<th>DeleteRight</th>
<th>UpdateRight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Company</td>
<td>true</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Company</td>
<td>true</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>James</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>James</td>
<td>true</td>
<td>true</td>
<td>null</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>James</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>Michael</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Figure 6.6: Example of the DocumentACL table

- *James* is allowed to select document 1, 2, and 4. He is also allowed to update values of these documents and to delete the tuple with the *Id* 1 and 4. Though he is member of *Company*, it does not affect his permissions, because none of the rights of *Company* is stronger than his.

- *Michael* would have the right to select document 1 and 2 via his membership in *Company*, but he is denied for all rights regarding document 2.

The Three Basic Roles of ModelLib

The previous section introduced the role concept. Section 6.1.3 mentioned the possibility to set global rights to database objects and how this supports the protection of data. This concept is another aspect, which is used in the ModelLib meta information database to guarantee the security of data.

ModelLib knows three basic roles, which are the following:

- **mladmin** – is a high level role with all possible rights, e.g., creating, dropping of all kind of database object, and full access to all tables, views, etc. A member of the *mladmin* role is always operating on the real table, thus it is not necessary to specify any rights in any ACL.

- **mluser** – is an authorised user, who has been verified positively. Members of this role are not able to create or drop any database object. They have absolutely no rights on any table (except a select right on some tables with non-sensitive data, e.g., *DesignTools*). What they have is the permission to *SELECT*, *UPDATE*, *DELETE*, and *INSERT* on views, and only on views.

- **mlguest** – is a role, which is declared for every unverified user of ModelLib. Like the *mluser* role, this role is not able to create or drop any database object and has absolutely no rights on any table. This role has only the permission to *SELECT* on views. As the name is indicating it is mainly used for guests, so that they are allowed to see public information about models, but are unable to provide data.

The roles *mlguest* and *mluser* are conform to the actors in the Use Case Diagram in Figure 4.1. Every user must be member of at least one of these three roles otherwise he would not be able to use ModelLib.

Data Manipulation in Views

The reader may have noticed, that this chapter often speaks of *UPDATE*, *DELETE*, and *INSERT* operations in the context of views. According to the definition of views, which declares them as not
6.1 Design of the Data Tier

materialised relations, they do not contain any data to manipulate. In fact, every operation will fail
on a view. This conflict can be solved through rules.

The rule system is part of the PostgreSQL 8.1 DBS. The concept is similar to stored procedures
and triggers, only that rules are not a programmed function. They are a simple query rewriting
mechanism, which modifies queries before their execution and completely founded on SQL. Thus
rules have the same boundaries of what they are capable to do, e.g., no access to internals, on
scanning of results, etc. Their advantage is, that rules do not need a set of data to work with,
because they just rewrite the query. [Pos07]

For example, a rule that reacts on an UPDATE on a view can replace the addressed view with the
real table, so that the query is passed to the data source of the view instead. Thereby, it is possible
to use conditions in a WHERE clause, so that the ACL information are considered.

This should give an idea of how the manipulation on views works. More details will be given
once the first view and its rules are discussed.

The previous sections explained the concept of the combination of an ACL and a view to provide
a tuple-wise access control, the structure of such an ACL, the three basic roles in ModelLib, and
introduced how to operate on views. In the following, it is described in detail, by the use of
examples, how all these fits together.

Recursive Computation of the Model Class Access Rights

Figure 6.7 shows the ModelClass entity and its relations. The special aspect about model classes
is their hierarchical structure, which is unfolding like a tree. This has much influence on the
access control. Like in file systems, the user’s rights of a parent node apply for the child node(s),
too. This means that a child model class first inherits the access rights of its parent model class
and than merges it with its own. Thereby, the transitive order for the values remains as defined

Figure 6.7: ER Diagram of ModelClass
above \((null < true < false)\). A child class keeps its rights until a parent has a stronger value. On any given node the access rights have to be computed recursively up to the root of the tree.

For example, five classes of models belong to a class \textit{special switches} in the hierarchy \textit{electrical domain, mixed-signal, switches, special switches}. If someone wants to protect the five classes from someone else, he simply denies all rights on the class \textit{special switches}. In another scenario, a whole new branch is added to a class, containing only public classes. Without any further specification of rights this branch simply inherits all the rights from its parent under which it was added.

The problem with such a structure is that a view on the \textit{ModelClass} table cannot just simply join the \textit{ModelClassACL} table. It is necessary to look up if the current model class has a parent model class in the \textit{ModelClassHierarchy} table and what are its rights and than does this one have a parent again, and so on. Usually, the iteration depth is unknown or changing. A normal SQL query is not capable to use a parameter or any dynamic value, which is required for a recursive computation of the effective access rights for a model class node. The SQL99 standard specifies the \texttt{WITH} clause to have a recursive query, but unfortunately PostgreSQL 8.1 does not feature this clause [Pos07]. Thus a recursive function had to be written.

Figure 6.8 and 6.9 show the Activity Diagrams of two functions developed to compute the effective access control rights for every model class. The first one is executed in the moment of a query for the \textit{ModelClassEffectiveACLView}. It sets initial parameters, which is necessary, because the view itself can not provide a row of the \textit{ModelClassACL} table. Thus the view is not able to call the recursive function directly. After setting the parameters, the function loops through a set of \textit{Ids} from all model classes. Thereby it calls for every element of the set the second function, which is going up the hierarchy, computing the rights from the root, and then returning them.

The second function does the actual work. It receives a row in form of an ACL tuple as input. First it checks if all rights are already denied, which means that the values will not change, no matter what is specified in any parent node. Thus it will lead to the termination of the algorithm with the return of the row. If no entry in the \textit{ModelClassACL} table for the current role and model class exists, the algorithm detects the parent model class, calls itself with the parent model class \textit{Id}, and overwrites the result with the original \textit{Id}. If no parent node is detectable, the algorithm terminates by returning a row of an ACL tuple. In case of a positive check for an existing entry in \textit{ModelClassACL}, the algorithm checks for a denied \textit{SelectRight} in the current variables, than for one in the ACL, than for a granted right in the ACL. Then, this is processed analogously for \textit{UpdateRight} and \textit{DeleteRight}, which is addressed by the fact, that Figure 6.9 shows a grey dotted box around some activities to compute the \textit{SelectRight}. The same gray dotted line can be seen in the activities \textit{check update rights} and \textit{check delete rights}. This shall symbolise that the algorithm to compute the update and delete rights, is exactly the same as for the select rights, only that every \textit{Select} must be replaced by \textit{Update} and \textit{Delete}. This is not strictly UML conform, but simplifies the figure.

Once the two functions have terminated and \textit{ModelClassEffectiveACLView} contains the effective rights for the current user and for all model classes, it is possible to join the \textit{ModelClass} table with this view (like the normal solution without a tree like structure is intended). The result is a relation that contains only the tuple the user is granted to access. Additionally, the \textit{ModelClass} table has an attribute \textit{Owner}, which contains the role name of the owner of the model class. It ensures that the owner of a model class is always able to access his class. It is intended to present own model classes in a separate category ‘own model classes’, so that it is not effected from the hierarchy and any specified ACL entries. It also prevents model classes from ‘getting lost’ in the case that they
6.1 Design of the Data Tier

get all ModelClassIDs from ModelClass

loop
  current user
  set Rolename =
  set ModelClassId =
current Id
  set SelectRight = null
  set UpdateRight = null
  set DeleteRight = null
  call modelclassacl_recursive _compute_function(row)

return

called

modelclassacl_compute_function()

get all ModelClassIDs from ModelClass

loop

end of list

write result in list

of all Model ClassIDs

of ModelClass ACL

of effective ModelClass ACL

Figure 6.8: Activity Diagram of the function called by the ModelClassEffectiveACLView
6 Design

Figure 6.9: Activity Diagram of the recursive function to compute the effective rights
6.1 Design of the Data Tier

are not correctly linked into the hierarchy. The following algebra term defines the ModelClassView in its final version:

\[
\text{ModelClassView} = \pi_{\text{ModelClassId}, \text{Name}, \text{Level}, \text{CreationDate}, \text{Owner}}(s \cup r) \quad \text{with}
\]

\[
s = \sigma(\text{ModelClass}) \bowtie \sigma_{\text{SelectRight} = \text{true}}(\text{ModelClassEffectiveACLView})
\]

\[
r = \sigma_{\text{Owner}=\text{current user}}(\text{ModelClass})
\]

As mentioned above, rules have to be defined to allow data manipulation on the view, such as \texttt{UPDATE} and \texttt{DELETE}, and \texttt{INSERT}. These rules replace the query statement on the view according to their definition. They use the condition to verify if the operation is allowed for a certain tuple or not. Assuming a tuple \( t \) of the relation ModelClassView shall be updated. The corresponding rule is defined to do this update on the ModelClass table instead, but only if the ModelClassId to update is an element of the set of ModelClassIds in the ModelClassEffectiveACLView, which the current user has a granted update right. The equivalent algebra expression is the following:

\[
\text{UPDATE} (t(\text{ModelClassView})) = \begin{cases} 
\text{UPDATE} (t(\text{ModelClass})) & \forall t \in r \\
\emptyset & \text{else}
\end{cases}
\]

with

\[
r = (q \cup s)
\]

\[
q = \pi_{\text{ModelClassId}} (\sigma_{\text{RoleName}=\text{current user} \land \text{UpdateRight}=\text{true}}(\text{ModelClassEffectiveACLView}))
\]

\[
s = \pi_{\text{ModelClassId}} (\sigma_{\text{Owner}=\text{current user}}(\text{ModelClass}))
\]

The rule for a delete statement is analogously defined with a \texttt{DELETE} instead of the \texttt{UPDATE}. The rule for an insert statement is simply changing the addressed relation, because everyone is granted to add new tuples.

\[
\text{INSERT} ((t(\text{ModelClassView})) = \text{INSERT} (t(\text{ModelClass}))
\]

The corresponding SQL statements of the ModelClassView and its rules are shown in Section 7.1 as part of the chapter about the implementation.

This section described how a particular ACL is structured and handled, and how rules are used to manipulate data in a view. The next section is setting the scope a bit wider by showing how entities without an own ACL can be controlled by the ACL of an entity they are linked to.

Access Control of Entities without an own ACL

Looking at Figure 6.3, one may wonder why only six entities have an ACL table and the rest not. This does not mean that the data they are containing is public. Most of the data therein needs to be secured as the one in the tables with an own ACL.

Most of the entities have a strong relationship to the entities with an own ACL, so that they are indirectly depending on this ACL. The topic of this section is to describe how the security for such linked entities is realised. The ER Diagram in Figure 6.10 shows a typical example of an entity (Interface) with an ACL and entities (Port, Parameter), which are linked to that superior entity and thereby need to be under the scope its ACL. Similar to entities with an own ACL, the access control of a linked entity is based on the ACL-view-rule concept, except that linked entities use the rights of the entity they are linked to. Therefore the join includes not only the data table
itself and the ACL table, but also the relationship table and the linked table. This makes the rules more complex, but not less efficient.

For instance, every tuple of the table Parameter is linked to either one Interface tuple or one Architecture tuple (the cardinality is specified for a particular relationship, so that from the interface-parameter-perspective parameter tuples exist, which are not linked to an interface). Thus a Parameter tuple is only selectable if the current user has a select right on the corresponding Interface tuple. This includes the InterfaceParameter table, which contains the attribution of parameters to an interface via their Ids. In the following it is explained how the view of the parameter table and its rules are composed:

**ParameterView (pv)** – Since the InterfaceView (iv) already contains only interface tuples the current user is allowed to see, the InterfaceView can be used to be joined with the InterfaceParameter (ip) table and than with the Parameter (p) table. This is expressed by the following algebra term:

\[ pv = p \Join (\pi_{InterfaceId}(iv) \Join ip) \]

**Update Rule** – An update on a tuple of the ParameterView (pv) is not directly granted by a right in an ACL table. A user has indirect rights to change tuples of an interface view. Parameter data, which belong to an interface are also part of the information about this interface. If someone changes the values of a parameter tuple, it also effects the interface. Thus, any change on parameter information updates an interface. This is why a user should only be able to manipulate (update, delete, or insert) a parameter tuple if that particular user is granted an update right in the InterfaceACL (ia) of the corresponding interface. Otherwise a change
of the parameter information could have a destructive effect on the stored definition of the model interface. The algebra expression for the update rule is the following:

$$UPDATE(t(pv)) = \begin{cases} UPDATE(t(p)) & \forall t \in r \\ \emptyset & \text{else} \end{cases}$$

with

$$r = (p \bowtie pi \bowtie (q \cup s))$$
$$s = \pi_{InterfaceId}(\sigma_{owner='current user'}(Interface))$$
$$q = \pi_{InterfaceId}(\sigma_{UpdateRight='true'(rm \bowtie ia)}) - \pi_{InterfaceId}(\sigma_{UpdateRight='false'(rm \bowtie ia)})$$
$$rm = \pi_{Rolename}(\beta_{Rolename=MemberIn}(\pi_{MemberIn}(\sigma_{Rolename='current user'(rmv))))$$

The relation rmv is an abbreviation for the term RolesMembershipView. It depicts a view which uses internal schema information of the DBMS to determine of which role the current user (who is a role by himself) is a member. Though it is necessarily used here, it is describe in detail in the next section. The previously mentioned ModelClassEffectiveACLView contains only access rights for the current user, because it is computed by a function, which returns only values for the current user. In all other cases of the security system of ModelLib, the rights are determined directly from the ACL tables. Thus, more then one entry (depending on how many other roles a user is member of) are determining a specific right for a user. According to the concept of roles (see Section 6.1.4), a user can be member of various other roles, so that a right is granted for one role, but denied for another role. In such a case, the user has no permission to access the particular tuple. This is the reason why first the set of InterfaceIds with granted rights is retrieved and then the set of InterfaceIds with denied rights is subtracted (difference operator) from that set. The result set q is united with the set of InterfaceIds the current user owns, and than joined with the ParameterInterface relation and the Parameter relation. The result is a relation of tuples linked to an interface, which the current user is permitted to update.

**Insert Rule** – For an insert of a tuple in the ParameterView apply the same conditions as for an update. A tuple can only be inserted and linked to an interface, if the user is granted the according update right. The algebra expression is analogously defined to the one for an update, thus not presented here.

**Delete Rule** – A delete of a tuple in the ParameterView is also only possible, if the deleting user is granted to update the corresponding interface tuple.

This concept of access control is applied to every view of the linked entities. Linked entities are all entities, which have no own ACL, but have a strong semantical relationship to an entity with an ACL. A user is only permitted to manipulate these entities if, he is permitted to update the corresponding entity. The Port entity is one of these linked entities, but is showing an additional characteristic. It shows a relationship to the entities PortDirection and PortClass. Both entities belong to a small group of entities, which cannot be modified by a guest or an authorised user. They are used to specify a closed set of data for an entity attribute. For example, a port must specify a direction, which can be one of input, output, input/output, or none. These values hold true for every port. This group of global entities include:

- DesignLanguage,
6 Design

- DesignLanguageVersion,
- DesignTool,
- DesignToolVersion,
- DocumentType,
- PortDirection,
- PortClass,
- Severity, and
- Vendor.

They are managed by the content manager (see Figure 4.1). Since they are not bound to a single tuple and also not changeable by the user, they do not need a view or a special ACL. The rights are set global, which means that the basic roles mluser and mlguest have only a select right on these tables, while members of the mladmin role have full rights on them.

This section described how entities with an ACL and such without interacted through their relationships to ensure an effective access control mechanism.

Internal Schema information for Roles Membership

The previous section mentioned the so called RolesMembershipView (rmv), which provides information about the membership the users or roles have in other roles. Therefore it is necessary to access meta information of a database, which contain data about:

- which roles in the database exists, and
- which role is member of which role.

The PostgreSQL stores all kind of information about a database in tables of an internal schema, which is accessible by the owner of the database. This so called information_schema contains, amongst others, the tables enabled_roles (er) and applicable_roles (ar). Querying the first returns all roles of the database, while the second provides role names and the names of roles they are a member of. After duplicating the enabled_roles (er) table by joining it with itself, it is possible to unite this duplicated relation with the projection of the first two columns of the applicable_roles (ar) table. The result is the RolesMembershipView (rmv). The definition in algebra form is as follows:

\[ rmv = (r \cup s) \]

\[ r = \pi_{rolename, memberin}(er \bowtie (\beta_{memberin \rightarrow rolename}(er))) \]

\[ s = \pi_{grantee,rolename}(ar) \]

The advantage of the union operation for \( r \) and \( s \) instead of a join is that the view gets tuples, which express that a role is member of itself. This is very helpful, because a query for the membership of a role returns a single column instead of all membership roles in one set and the origin role in a separate one. This makes it much handier to use such query for access control checks, because this single column can be compared to the RoleName column in an ACL table.

Though this view is very simple, it is very important. The RolesMembershipView is the most used view in the security mechanism, it is part of almost every query addressing an ACL table. The SQL implementation of this view can be found in Section 7.2. It also gives an idea of how the internal schema can be used.
6.1 Design of the Data Tier

Handling the Complexity of ACLs

Thinking of the number of tuples and users, which will grow over time, the access control may become very complex. Hence, a mechanism is required, which allows as much control as needed, but as less specification effort for rights as possible. A first step is the role system provided by PostgreSQL to group users, so that rights can be specified for various users at once. A second step is kind of inheritance of rights from one entity to another. This ease of handling the ACL tables is the topic of this section.

The inheritance of rights affects those entities with an own ACL, which also have a kind of a ‘part of’-relationship. This includes the following tables:

- **DesignEntity** – which is always an implementation of a certain architecture,
- **Architecture** – which is always the internal description of an interface,
- **Interface** – which is always part of at least one model class, and
- **ModelClass**.

Thereby, the idea is to inherit the specified ACL rights of the belonging entity, as long as no rights for the inheriting entity are specified. For instance, in case of no specified access rights for a DesignEntity tuple, the rights from the corresponding architecture have to be applied to control the access. Are these rights also not specified, the rights from the corresponding interface have to be applied, otherwise the tuple can not be accessed. Figure 6.11 illustrates this concept.

The relationship between ModelClass and Interface does not fit to this concept, though the rights of Interface are related to the one of ModelClass. It is different from the relationship between Interface, Architecture, and DesignEntity, because an interface can only be seen if it is element of a model class the user is allowed to see. That means whenever the ModelClassView contains the corresponding model class. For all other operations (delete, insert, update) the InterfaceACL is used to determine the rights.

This kind of inheritance of rights makes it possible that a new interface can simply be sorted into a public model class and become available for many users. This model is then selectable, without any further rights specification. At a later time, the owner of this interface is free to decide to grant additional rights. These rights will apply to the architecture and the design entity. At one point the owner decides to deny, for example, the update right to some of the architectures for his interface. At the moment one entry (in the ArchitectureACL) for a certain architecture is made, this architecture does not inherit rights any longer. Only the now specified rights in the ArchitectureACL apply to this architecture. Thus it is not possible to ‘forget’ a role.

The Activities Diagram in Figure 6.11 would be easy to implement in a complete function like in the case of the recursive computation of the effective rights for the model classes (see Section 6.1.4). It was decided not to do this, because the use of such an extra function would cost flexibility and platform independence. Another approach, which used the view concept to compose the effective rights, caused a major loss in performance. The approach made it necessary to join more than 10 views, while each of them would be queried on demand. This led to a dead-end solution, because the composition, for example, for the effective architecture rights took more than 6 minutes, even if the ACLs had less than 25 tuples each. A much more efficient but similar approach was developed. It uses triggers and one additional table, each containing the effective rights for the inheriting relations (Architecture, DesignEntity).
access a tuple
of
determine right for a
certain DesignEntity

not specified
determine right for a
certain Architecture

specified

not specified
determine right for a
certain Interface

specified

not specified
check if Interface is in
a selectable ModelClass

specified

only show tuple

no

yes

apply rights

Figure 6.11: Activity Diagram of the rights inheritance
6.1 Design of the Data Tier

Trigger – A trigger is a procedural function automatically firing on a prior defined event, e.g., an manipulation statement. The trigger is reacting whether once per affected row of a table or once per complete statement. It can fire before or after the execution of a statement. Triggers are part of the SQL specification. [CFMS95]

Triggers are used to fill the two additional tables, called ArchitectureEffectiveACL and DesignEntityEffectiveACL. The data of these two relations are composed by the trigger, which merges the ACL data from the inheriting table with the one from superordinate table. Figure 6.12 illustrates this composition. After every delete, insert, or update statement on the ACL tables the triggers fire and execute their function.

This trigger function is described in the Activity Diagram in Figure 6.13 and is processed as follows:

1. It deletes all tuple of the ACL table with the effective rights (e.g., ArchitectureEffectiveACL).
2. The function queries all data from the superordinate ACL (e.g., InterfaceACL). This set may contain multiple tuple specifying rights for different roles, but for one interface.
3. The function subtracts all tuple for an entity (e.g., Architecture) for which an own ACL (e.g., ArchitectureACL) tuple exists. This ensures that in moment an access right is explicitly specified the inheritance no longer affects the access of the corresponding tuple (e.g., the architecture).
4. The function adds all entries from the ACL table (e.g., ArchitectureACL), which belongs to the inheriting entity (e.g., Architecture).
5. The function inserts the resulting set in the table with the effective access rights (e.g., ArchitectureEffectiveACL).

The so defined trigger function has to be implemented twice, once for the composition of ArchitectureEffectiveACL and once for the composition of DesignEntityEffectiveACL. The use of input parameters (which would allow to implement just one function) is possible, but the necessary implementation details could not be cleared during this thesis.

The triggers itself are defined at 4 relations, which are:
6 Design

trigger fired

delete all ACL tuple of the EffectiveACL of S

for each tuple of C receive all SACL tuple of the corresponding S

subtract all tuple from this set which have an own entry in the CACL

add all tuple of CACL

write this set in the EffectiveACL of C

C = current entity
S = superordinate entity

Figure 6.13: Activity Diagram of the trigger function

- InterfaceACL – so that a change on this table causes a recomposition of the ArchitectureEffectiveACL,
- ArchitectureACL – so that a change on this table causes a recomposition of the ArchitectureEffectiveACL,
- ArchitectureEffectiveACL – so that a change on this table causes a recomposition of the DesignEntityEffectiveACL, and
- DesignEntityACL – so that a change on this table causes a recomposition of the DesignEntityEffectiveACL.

This combination of effective rights tables and triggers enables the inheritance of rights as required and, at the same time ensures the integrity of the access control data. The inheritance of rights eases the management, because rights must not be specified on every entity, even though this is possible. It is expected that in most cases it will be enough to specify rights for an interface and use the same rights for architectures and design entities.

6.2 Design of the Business Tier

To speak as an architect, a solid house needs a solid foundation. The presentation tier and the business tier are this house, build on the data tier. In the case of ModelLib, the design of the data tier needed to be done well thought, especially in meeting the requirements regarding the protection of IP. The development of the tuple-wise access control and its realisation contributed the major part to this thesis. The realisation of other requirements, e.g., the browsing for models, the review process for new models, or the connection interface to EDA tools have a lower priority.
Design of the Business Tier

than the security aspects. Additionally, new requirements for the information stored in the database and their protection emerged during the realisation of the data tier. The resulting problems from the experiences with the ModelLib prototype and the discussions with the project partners had to be solved before the design of the other tiers makes sense. Of course, the realisation of security mechanisms in the data tier leads to less effort for these issues in the business tier. A business logic is now able to simply pass queries and let the ACL mechanics do the work of verification. However, the limited amount of time for this thesis left did not allow a detailed design of the business and the presentation tier. That is why only a first tentative draft is presented.

The technology chosen for the realisation of the business tier are EJB (see Section 5.2.2), in fact Entity Beans and Session Beans. The use of Entity Beans mostly benefits from the a persistence manager, which is responsible for the consistence and persistence of data. It manages concurrent access to shared objects. Thus Entity Beans can represent the database objects. The persistence manager has to be mapped to the objects in the database. This so called object-relational mapping [KS06, pp. 5] is not trivial, especially in combination with views. However, once the persistence manager is established, it prevents the developer from the use of SQL statements to keep the data, handled in the application, synchronous with those in the underlying data source. [KS06]

The persistence manager also eases the maintenance of the system, because changes in the database must not be adapted to the SQL statements in the application. As already mentioned, the data tier absorbed the most work, thus the object-relational mapping for the persistence manager is not applied here.

An approach for design of the business tier without a persistence manager has two possible solutions to use Entity Beans:

- Database objects are handled by SQL statements through methods within the Entity Beans,

- Database objects are handled by SQL statements through separate Session Beans.

The author of this thesis favours the second approach, because the Session Beans can be easily removed in case that a persistence manager is established. The Class Diagram in Figure 6.14 shows the idea of the design for the business tier. The database objects are represented by Entity Bean classes. Their parameter would be mainly the attributes of the table or, in case of a composition or an aggregation (e.g., Document and Author) other entities. Their methods are the setter and getter methods for the parameters (not specified in the Class Diagram due to clearness reasons). The Entity Beans classes are associated with their respective handler Session Beans, which instantiate the Entity Beans. Handler Beans inherit the methods of the abstract class EntityBeanHandler and may implement additional parameter and methods. The EntityBeanHandler is abstract (with empty methods), because each inheriting handler Bean must implement the methods differently depending on SQL statements. These statements implement the interaction with the database to query the values for the entities. The EntityBeanHandlerRemote interface is the connection point for other components of the system. The GlobalSessionBean is stateful and instantiates a session. This session is bound to the client and carries the user information and a database connection. The database connection is kept during an active session, which makes it reusable for the involved EntityHandler Beans. The interface GlobalSessionRemote makes the Bean available for other components, e.g., Servlets or JSP.
Figure 6.14: Class Diagram of a possible business tier of ModelLib
6.3 Design of the Presentation Tier

The presentation tier should also be designed modularly. For example, one main page includes other pages. Information, which is used more than one time should be separated in an own page, so that it is reusable in multiple situations. Figure 6.15 gives an idea of possible components of the presentation tier. A main page, e.g., index.jsp includes components such as header.jsp, footer.jsp, and menu.jsp. They all provide elements of the presentation, which are part in every page. The content.jsp represents an element with different content depending on the selected menu option, e.g., a page for managing of data about the user or a page for the design tool information.

A typical layout of a page is presented in Figure 6.16. The header contains information about navigation, e.g., which element the user selected or what his last action was. The login section is where the user enters his account and password for the authentication. If he is already authenticated, information about his account are presented instead, e.g., membership of roles, last model browsed, etc. The footer shows external links and information about the ModelLib system, e.g., application version, responsible developer, etc. The model class navigation part enables a user to browse the classes in their hierarchy, like in file browsers (e.g., Microsoft’s Windows Explorer, KDE’s Konqueror). The main menu section provides access to certain presentation parts of ModelLib, e.g., design tools, references, help section, etc. The content section presents the information the user has requested.
Since the business tier and the presentation tier have not been implemented yet, some screenshots of the prototype are used to give an idea how the meta information from the database can be presented.

Figure 6.17 shows the header information on top, giving the user the information that he is actually in the Design Tools section. Below this, one can see the main menu to enter the different sections of ModelLib. What follows is the name of the design tool and its vendor (ADVance MS from Mentor Graphics Corp.). Also a table of design languages and their versions supported by the design tool and references to documents can be seen. The bottom of the screenshot shows the footer, which contains user information and additional links.

Footer, main menu, and header are also included in the second screenshot (Figure 6.18). It shows meta information about a model, in this case an active low pass filter. The model class path is on top followed by the interface meta data. Two tables present the parameters and the ports, which belong to this interface. The corresponding architecture information comes after the interface section. Following the description of the architecture, the information about the design entities implementing this architecture in particular design languages. A table of attached files and the references for the model follow the design entities. The name of a file is the link to the model source code in the Subversion repository, so that the code can easily be found. One can see that several add and edit buttons (e.g., Add Architecture, Interface Edit) are displayed throughout the page. The implementation of a new version, based on this thesis, should only make such buttons available, if the user possesses the required rights to delete, update, or insert information.

These two examples, though they are borrowed from the prototype, are very suitable to give an idea of how meta information about AMS design models can be presented to the user.
6.3 Design of the Presentation Tier

Figure 6.18: Presentation of the meta information about a model [MV06]
This chapter described the design of the Web-based application regarding the chosen N-tier architecture. The focus was clearly on the data tier. This is why the relation to the two other tiers is unrepresented. The reason is the high relevance of the protection of some information in the database regarding the IP.
7 Implementation

According to the software life cycle the implementation follows the design. The previous chapter explained the design of the ModelLib application, especially the data tier. The high priority of security aspects caused the focus of this work on the design and implementation of the data tier. That is why only a tentative draft for the other tiers could be developed. Without a design, there is no implementation. Thus, the topic of this chapter is the implementation of the data tier.

From the 57 tables, 27 sequences, 2 functions, 2 trigger functions, 4 triggers, and the 46 views with 129 rules, implemented in the database, only a few aspects are described in this chapter. They are related to the presented issues of the design of the data tier. Further it is not intended to provide the complete source code or describe every function or unit test. Moreover, examples are used to illustrate the realisation and the test concepts.

7.1 Computation of the Model Class Access Rights (Implementation)

Section 6.1.4 described two functions necessary to compute the effective rights of model classes depending on their parent class. The PL/pgSQL – SQL Procedural Language, provided with PostgreSQL, is used to write such a recursive function. Though the use of such an additional language decreases the portability and platform independence, in this case it could not be avoided. PL/pgSQL has a manageable set of keywords, which allow loops, if-then conditions, etc. to add control structures to the SQL language. It is easy to read and to learn.

The Listing 7.1 shows the source code of the function for the ModelClassEffectiveACLView from the Activity Diagram (Figure 6.8) as an example of PL/pgSQL. Line 1 and 2 specify the name and input parameter of the function. The lines 5 to 7 declare the variables of the function. Line 13 fills the variable with the ModelClass tuples via a SELECT statement and loops through them. Lines 15 to 19 set the initial values for the parameter, which are handed over to the recursive function. Line 20 fills the return variable with the result from the recursive function (which is much more complex, and thus attached in the Appendix A.1). The Lines 29 and 30 set global security rights for this function.

Listing 7.1: Source code of modelclassacl_compute_function()

```sql
CREATE OR REPLACE FUNCTION modelclassacl_compute_function()
RETURNS SETOF modelclassacl AS
$BODY$
--This function initialises the values and hands them to the recursive function as parameter. A view or a query are not able to call a function with such a parameter
-- and can not set the rolename in this parameter. The rolename would otherwise been overwritten in the recursive function
Declare
var_modelclassacl_row modelclassacl%Rowtype;--type ‘Rowtype’ is one or more rows of the table named before ‘%’ and its column types
```

83
7 Implementation

```sql
var_modelclass_row modelclass%RowType;

Begin
  --var_modelclassacl_row will be filled with an acl row for every modelclass
  --merged from all rows of modelclassacl which contain roles the current user is member of
  --and the rights of parent nodes of the modelclass
  for var_modelclass_row in Select modelclassid FROM modelclass
    Loop--for all modelclasses

    var_modelclassacl_row.rolename := current_user;--initialises with the current user and guarantees that this is the one checked for in recursion
    var_modelclassacl_row.modelclassid := var_modelclass_row.modelclassid;--initialises with the modelclassid from the loop
    var_modelclassacl_row.selectright:= null;--initialises with the weakest right
    var_modelclassacl_row.updateright := null;
    var_modelclassacl_row.deleteright := null;
    Select Into var_modelclassacl_row * From modelclassacl_recursive_compute_function(var_modelclassacl_row);--call recursive function with the initialised variable

  return next var_modelclassacl_row;
  End Loop;
  return;
  End;
$BODY$
LANGUAGE 'plpgsql' VOLATILE;
ALTER FUNCTION modelclassacl_compute_function() OWNER TO mladmin;
GRANT EXECUTE ON FUNCTION modelclassacl_compute_function() TO mladmin;
GRANT EXECUTE ON FUNCTION modelclassacl_compute_function() TO public;

This function is called from the ModelClassEffectiveACLView. The view is defined in SQL as shown in Listing 7.2

Listing 7.2: SQL statement for the ModelClassEffectiveACLView

CREATE OR REPLACE VIEW modelclasseffectiveaclview AS
  SELECT modelclassacl_compute_function.rolename,
         modelclassacl_compute_function.modelclassid,
         modelclassacl_compute_function.selectright,
         modelclassacl_compute_function.updateright,
         modelclassacl_compute_function.deleteright
  FROM modelclassacl_compute_function()
  ORDER BY modelclassacl_compute_function.modelclassid;
ALTER TABLE modelclasseffectiveaclview OWNER TO mladmin;
GRANT ALL ON TABLE modelclasseffectiveaclview TO mladmin;
GRANT SELECT ON TABLE modelclasseffectiveaclview TO mluser;
```
7.1 Computation of the Model Class Access Rights (Implementation)

![Figure 7.1: An example of a model class hierarchy](image)

Table 7.1: Possible ACL values and their theoretic result in a computation (n=null, 1=true, 0=false)

<table>
<thead>
<tr>
<th>ParentID</th>
<th>ModelClassId 1</th>
<th>ModelClassId 2</th>
<th>ModelClassId 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent right value</td>
<td>n</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ChildID</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>Child right value</td>
<td>n</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Child right value in ModelClassEffectiveACLView</td>
<td>n</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

This view contains the rights the current user has for every model class. Due to the fact that only three values occur in each right column of an ACL, the result of the recursive function is relatively easy to foretell. Basically nine possible cases exists, three values in a parent node and three for each in the child nodes. All other cases, no matter if higher or lower in the model class hierarchy, are one of these nine cases. Additionally, the computation of rights for select, updates, and delete have all the set of possible values (null, true, false), so that one right is representative for all.

The hierarchy shown in Figure 7.1, has three root nodes each having three child nodes. All cases can be simulated with this hierarchy. A specification of access rights as described in Table 7.1 should result in effective rights as to read in the last row of this table. This specification of rights was used to test the ModelClassEffectiveACLView and thereby the underlying functions. The model class hierarchy was build as shown, the values had been inserted in the ModelClassAcl table, and than the ModelClassEffectiveACLView was selected. The view presented the same values as expected. Thus it is reasonable to say that the functions are working correctly.

The view ModelClassview joins the ModelClassEffectiveACLView with the ModelClass table. The view for model classes is defined in SQL as shown in Listing 7.3. Therein the lines 4 to 8 define the conditions, under which a tuple from ModelClass is selected. The ModelClassId of ModelClass must be an ID where the current user equals the value in Owner or it must be part of a set, which contains all ModelClassIds from ModelClassEffectiveACLView where the SelectRight for the current user equals true. The line 12, 13, and 14 represent the global rights for the three basic roles as described in Section 6.1.4.

Listing 7.3: Definition of the view for model classes in SQL

```sql
CREATE OR REPLACE VIEW modelclassview AS
SELECT mc.modelclassid, mc.name, mc."level", mc.creationdate, J
mc."owner"
FROM modelclass mc
WHERE (mc.modelclassid IN (SELECT modelclasseffectiveaclview.modelclassid
FROM modelclasseffectiveaclview
WHERE (modelclasseffectiveaclview.selectright = true)))
```
7 Implementation

Listing 7.4 shows the rule, which is defined to realise an update operation on the view. In its WHERE clause, starting at line 4, are two of the three checked conditions similar:

- The first (line 4) is responsible to ensure that the changed tuples in ModelClass really are the ones intended to be changed. This condition is important for every rule, because without it the rule would update every tuple in the table.
- The second (line 5 to 12) checks if the old ModelClassId is granted to be changed.
- The third (line 13 to 20) checks if the new ModelClassId is granted to be updated.

The last two are also very important, because they avoid a scenario, in which a user abuses the rights he has to update a tuple to change the values and the Id. The changes would be passed to the table and applied there to the new Id instead of the old one as unauthorised manipulation. In the update rule for the ModelClassView, the new Id is not passed with the rule, so that it would not be a problem in this case, but it is used to show intention of conditions. Every update rule has this similar check for old and new *Ids. Though in most cases it is not explicitly necessary, it is implemented to take precaution against unauthorised manipulation.

Listing 7.4: The update rule for the ModelClassView

Listing 7.5 shows the delete rule. Since a delete operation does not have any new values, there is no need for a condition checking for them. The structure of the conditions in the rule is similar to the one for the update rule. This can be said about every rule, because they are all based on the same
7.2 Implementation of the RolesMembershipView

Listing 7.6 shows a typical insert right, which passes the new data to the rights table. In this case it is ModelClass.

Listing 7.5: The delete rule for the ModelClassView

```sql
CREATE OR REPLACE RULE delete_on_modelclassview_rule AS ON DELETE TO modelclassview
DO INSTEAD
DELETE FROM modelclass
WHERE modelclass.modelclassid = old.modelclassid
AND ((old.modelclassid IN (SELECT mceav.modelclassid
FROM modelclasseffectiveaclview mceav
WHERE mceav.deleteright = true))
OR (old.modelclassid IN (SELECT mc.modelclassid
FROM modelclass mc
WHERE mc."owner"::name = "current_user"())));
```

Listing 7.6: The insert rule for the ModelClassView

```sql
CREATE OR REPLACE RULE insert_on_modelclassview_rule AS ON INSERT TO modelclassview
DO INSTEAD
INSERT INTO modelclass (modelclassid, name, "level", "owner", creationdate)
SELECT new.modelclassid, new.name, new."level", new."owner", new.creationdate;
```

The same approach to test the correctness of the rules was used as for the functions to compute the effective rights for the model classes.

7.2 Implementation of the RolesMembershipView

Listing 7.7 shows the implemented RolesMembershipView as described in Section 6.1.4. It is developed to determine the membership of a role. For this information from the internal schema of the database is used (lines 5 and 8), in fact a relation of all roles (line 5) and a relation of their memberships (line 7). Both are united to one relation.

Listing 7.7: The RolesMembershipView in SQL

```sql
CREATE OR REPLACE VIEW rolesmembershipview AS
SELECT rolesmembership.role_name, rolesmembership.member_in
FROM (SELECT enabled_roles.role_name, enabled_roles.role_name AS member_in
FROM information_schema.enabled_roles
UNION
SELECT applicable_roles.grantee, applicable_roles.role_name
FROM information_schema.applicable_roles) rolesmembership;

ALTER TABLE rolesmembershipview OWNER TO mladmin;
GRANT ALL ON TABLE rolesmembershipview TO mladmin;
```
7 Implementation

As already mentioned this view is involved in almost every query addressing an ACL table, e.g., in the rule for a delete operation on the *ParameterInterfaceView*, which is described in Section 6.1.4. Since the parameter entity has no own ACL and is linked to the interface entity, the rule uses the *InterfaceACL*. The rule checks if a user, who is about to delete the parameter tuple, is allowed to update the corresponding interface tuple. Its source code is shown in Listing 7.8 and defines how the manipulation of the view is passed to the original table. The query to the view is replaced (line 2) by the query to the *Parameter* table (line 3), but only if the condition in line 4 to 24 is true. First the deleted *ParameterId* is compared to the one in the parameter table, so that not all tuples are removed. Lines 5 to 24 specify a set, of which the linked *InterfaceId* has to be an element to permit deletion of the parameter. This is the main part of the security mechanism. It checks if the *InterfaceId* about to be deleted is an element of a set of tuples where:

- the current user or a role he is member of (line 9 to 11) has a granted *UpdateRight* in the ACL for the corresponding interface (line 6 to 12),

- subtracting all tuples the current user or a role he is member of (line 16 to 19) has a denied *UpdateRight* in the ACL for the corresponding interface (line 14 to 20).

In case that the current user is the owner of the interface, the delete operation of the parameter tuple is also allowed (line 21 to 24).

Listing 7.8: SQL source code of the rule for an delete on the *ParameterInterfaceView*

```
1 CREATE OR REPLACE RULE delete_on_parameterinterfaceview_rule AS ON DELETE TO parameterinterfaceview
2 DO INSTEAD
3 DELETE FROM parameter
4 WHERE parameter.parameterid = old.parameterid
5 AND (old.interfaceid IN (
6   SELECT ia.interfaceid
7   FROM interfaceacl ia
8   WHERE (ia.rolename IN (9
10     SELECT a.member_in
11     FROM rolesmembershipview a
12     WHERE "current_user"() = a.role_name::name))
13     AND ia.updateright = true
14   ) EXCEPT
15   SELECT ia.interfaceid
16   FROM interfaceacl ia
17   WHERE (ia.rolename IN (18     SELECT a.member_in
19     FROM rolesmembershipview a
20     WHERE "current_user"() = a.role_name::name))
21     AND ia.updateright = false))
22 OR (old.interfaceid IN (23     SELECT i.interfaceid
24     FROM interface i
25     WHERE i."owner":name = "current_user"()));
```

The two listings of this section represent the typical use of the *RolesMembershipView* in a common rule for manipulating a view. Further source code will not be presented here, because the structure of rules is repetitive and thus would not provide much benefit. Instead, a scenario to test the correctness of the functionality of the *RolesMembershipView* in the use of a rule is described.
Table 7.2: Possible roles with and without membership and their rights

<table>
<thead>
<tr>
<th>Rolename</th>
<th>DocumentId</th>
<th>ACL rights for operation on the view for</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>select</td>
<td>update</td>
</tr>
<tr>
<td>a_role</td>
<td>31</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>member_of_a_role</td>
<td>31</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>not_member_of_a_role</td>
<td>31</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>owner_of_document</td>
<td>31</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>a_role</td>
<td>32</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>member_of_a_role</td>
<td>32</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>not_member_of_a_role</td>
<td>32</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>owner_of_document</td>
<td>32</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>a_role</td>
<td>33</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>member_of_a_role</td>
<td>33</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>not_member_of_a_role</td>
<td>33</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>owner_of_document</td>
<td>33</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

The test addresses the DocumentView (containing the documents depending on the rights of the current user). It covers the full spectrum of relevant cases that can occur (owner, roles with and without a membership combined with the different and equal rights). Instead of combining the three values with three other values for the select right and than repeating this for the update and for the delete right, the combinations are distributed over the three rights. This is possible, because the rights are computed equally. Therefore Table 7.2 lists the typical ACL table values in the first five columns (self explaining role names have been chosen) and the results, expected if the user is executing an operation on the view, in the last three columns. The values in column 1 to 5 are inserted in the DocumentACL. If one of the four role is connected to the database and executes an operation on the DocumentView the columns 6 to 8 describe what happens. For example, the role member_of_a_role selects the view. His own ACL values would allow him to see the documents with DocumentId 31, 32, and 33. Since he is member of a_role, which is denied to access document 33, he is also denied to access this document. Hence the can not see this document as the columns 6 symbolises with a ‘denied’ entry. In contrast, the user not_member_of_a_role has no membership and thus his rights are directly applied as specified. The role owner_of_document is the owner and thus always allowed to have all rights on the document, regardless of any specified rights in the ACL or any membership. The theoretical results are equal the results of the actual test, so that the correctness of the rules can be ensured. The test scenario is also applied on many other views, because of its general structure and completeness.

7.3 Implementation of the Trigger functions for Rights Inheritance

Section 6.1.4 described functions for triggers on the tables InterfaceACL, ArchitectureACL, DesignEntityACL, and ArchitectureEffectiveACL. The functions are used to compute the inheritance of rights for their corresponding tables. The rights for the Architecture tuples are merged from the InterfaceACL and the ArchitectureACL and than written in the ArchitectureEffectiveACL. The rights from ArchitectureEffectiveACL and the DesignEntityACL are merged and than written in the DesignEntityEffectiveACL. The trigger function is first deleting all tuple of the table containing the
Table 7.3: Results of the performance test of the triggers

<table>
<thead>
<tr>
<th>Number of tuples in the InterfaceACL</th>
<th>Operation on the tuples (avg. time per statement in ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>update</td>
</tr>
<tr>
<td>50</td>
<td>43</td>
</tr>
<tr>
<td>100</td>
<td>62</td>
</tr>
<tr>
<td>200</td>
<td>142</td>
</tr>
<tr>
<td>300</td>
<td>1942</td>
</tr>
<tr>
<td>500</td>
<td>5142</td>
</tr>
<tr>
<td>750</td>
<td>8461</td>
</tr>
</tbody>
</table>

effective rights and than merging them new. The source code of the function and the trigger is not put in this section to not overload it with long code snippets. It can be found in the Appendix A.2 Instead it is intended to show results of the performance tests of the functions. The test ran on a PC with 1 GB RAM and a 3 GHz Pentium 4 processor with 1 MB cache.

These tests are based on a kind of worst case scenario. This means that all manipulation operations (update, insert, delete), which fire the trigger, are executed on the InterfaceACL. This affects the recomposition of the rights in the ArchitectureEffectiveACL, which is than automatically triggering the recomposition of rights in the DesignEntityEffectiveACL. This leads to a maximum of workload. The Table 7.3 lists the results of the tests. Therein the average time in milliseconds (out of 20 iterations) of a statement is listed for every operation. The operations are executed on 50, 100, 200, 300, 500, and 750 tuples. The jump of time between 200 and 300 tuples is quite high compared to the one afterwards. It can be explained through the use of indexes and the optimisation, which is part of PostgreSQL’s internal query computation. It is applied once a certain number of tuples is involved in a query. The conclusion from the test is that the system performs acceptable up to 500 tuples in the InterfaceACL relation. The decreasing performance is a result of the simplicity of the trigger function algorithm, because first all tuples of the effective rights table are deleted and then rebuild. The development of a better algorithm, which is only re-composing tuples affected from a change instead of all, is intended as part of the future work.

Usually, the implementation is followed by the testing of components, the integration testing, and the deployment of the complete application. Due to the given reasons, this has to be done once the application is completed.
8 Conclusions and Outlook

The thesis at hand is entitled “Development of a Web-Based Application for Collecting Models and Supporting the Design of AMS Hardware Systems”. The title indicates that it addresses topics of software engineering as well as topics of electrical engineering, though the focus is clearly on the software engineering side. The intended goal to develop a complete running system with the functionality as described in Chapter 4 could not be reached. The reasons therefore are the complexity in the realisation of certain requirements, which occupied the major part of this work. After the decision for a N-tier architecture and for Java EE technology, which is described in Chapter 5, the project management decided to concentrate on the security issues first to reach a stable state in the database. Especially the realisation of a tuple-wise access control mechanism in the meta information database, which has the highest priority of the requirements, are part of the current version. The Chapters 6 and 7 described the design and the implementation of the access control in detail, though it was not possible cover every aspect in detail. The applied solutions in the database completely provide the required fine grained access control mechanism, so that the design and implementation of the business and presentation tier are able to build on that. Altogether this thesis contribute an important part to the ModelLib system.

The next step in the development of the Web-based application are the realisation of the business and presentation tiers. This addresses the other requirements, which are not implemented yet. Thereby, the implementation of the application logic with Session Beans is one task. Another one is the object-relational mapping with a persistence manager and Entity Beans. This allows a direct coupling between the database and the Entity Beans, so that the use of SQL statements would become unnecessary. Furthermore, the use of JSF for the GUI could ease the implementation of a review process for new model information. They allow a validation of input data in combination with special error messages or pages. For example, JSF pages process a syntactical validation and a Bean processes the semantical validation. Both use the same error pages to show the results of the validation. The development of a keyword search is an open issue as well as the connection to EDA tools. For the latter the use of remote interfaces of Session Beans are a promising approach to start with. An external tool is able to communicate with the Beans via the remote interface as if it is part of the system. This would enable the tool to use the application logic in the business tier directly. Though, it might be necessary to develop additional Beans to interact with the EDA tools, depending on how a tool interaction differs from a human user interaction.

Another task is the unification of the user account management. At the moment the database, the Subversion repository, and the Wiki platform use independent account management. In the future, the security of the GlassFish server as well as the Java version of the Wiki should use the PostgreSQL account information via coupling to JavaSecurity.

The current version of the ModelLib meta information database stores structural informations about models (how a model is build). It is intended to store also semantical meta information (How a model can be used). Once a database schema is developed for this, the view/rule system has to be extended to the new tables to benefit from the fine grained access control mechanism.
8 Conclusions and Outlook
A Additional Source Code of the Implementation

A.1 modelclassacl_recursive_compute_function(modelclassacl)

Listing A.1: The modelclassacl_recursive_compute_function(modelclassacl)

CREATE OR REPLACE FUNCTION ¶
  modelclassacl_recursive_compute_function(modelclassacl)
RETURNS SETOF modelclassacl AS $BODY$
Declare
  var_modelclassacl_row Alias For $1;--type 'modelclass' is one row of a table modelclassacl or query
  var_modelclassid modelclassacl.modelclassid%Type;
  var_parentid modelclassshierarchy.parentid%Type;
Begin
  --check if all rights are denied Yes-> stop recursion No-> go on
  If var_modelclassacl_row.selectright = false and
      var_modelclassacl_row.updateright = false and
      var_modelclassacl_row.deleteright = false Then
    Return next var_modelclassacl_row;
  Else
    --Test if allready an entry for role and modelclassid in modelclassacl exists Yes-> merge rights No->go higher in the modelclasshierarch
    Perform * From modelclassacl mca where
    var_modelclassacl_row.modelclassid = mca.modelclassid and
    mca.rolename in (Select a.member_in
    FROM rolesmembershipview a
    WHERE var_modelclassacl_row.rolename = a.role_name::name);
    If Found Then
      --Test if there is a denied selectrights on the modelclass Yes-> do nothing No-> go on
      If var_modelclassacl_row.selectright = false Then NULL;
      Else
        --Test if there is any denied selectright for the modelclass in the acl Yes-> set new selectright false No-> go on
        Perform * From modelclassacl mca where mca.selectright = false and
        var_modelclassacl_row.modelclassid = mca.modelclassid and
        mca.rolename in (Select a.member_in
        FROM rolesmembershipview a
        WHERE var_modelclassacl_row.rolename = a.role_name::name);
WHERE var_modelclassacl_row.rolename = $a.role_name::name$;

If Found Then
var_modelclassacl_row.selectright = false;
Else
-- Test if there are any selectrights granted on the modelclass
Yes -> set new selectright true No -> do nothing
Perform * From modelclassacl mca where mca.selectright = true $
 and var_modelclassacl_row.modelclassid = mca.modelclassid$
 and mca.rolename in (Select a.member_in
 FROM rolesmembershipview a
 WHERE var_modelclassacl_row.rolename = $a.role_name::name$);

If Found Then
var_modelclassacl_row.selectright = true;
End if; -- new selectright granted
End if; -- new selectright denied
End if; -- old selectright denied

-- Test if there is a denied updaterights on the modelclass
Yes -> do nothing
No -> go on
If var_modelclassacl_row.updateright = false Then NULL;
Else
-- Test if there is any denied updateright for the modelclass in the acl
Yes -> set new updateright false No -> go on
Perform * From modelclassacl mca where mca.updateright = false $
 and var_modelclassacl_row.modelclassid = mca.modelclassid$
 and mca.rolename in (Select a.member_in
 FROM rolesmembershipview a
 WHERE var_modelclassacl_row.rolename = $a.role_name::name$);

If Found Then
var_modelclassacl_row.updateright = false;
Else
-- Test if there are any updaterights granted on the modelclass
Yes -> set new updateright true No -> do nothing
Perform * From modelclassacl mca where mca.updateright = true $
 and var_modelclassacl_row.modelclassid = mca.modelclassid$
 and mca.rolename in (Select a.member_in
 FROM rolesmembershipview a
 WHERE var_modelclassacl_row.rolename = $a.role_name::name$);

If Found Then
var_modelclassacl_row.updateright = true;
End if; -- new updateright granted
End if; -- new updateright denied
End if; -- old updateright denied

-- Test if there is a denied deleterights on the modelclass
Yes -> do nothing
No -> go on
If var_modelclassacl_row.deleteright = false Then NULL;
Else
A.1 modelclassacl_recursive_compute_function(modelclassacl)

```sql
--Test if there is any denied deleteright for the modelclass in acl
Yes-> set new deleteright false
No-> go on
Perform * From modelclassacl mca
    where mca.deleteright = false
    and var_modelclassacl_row.modelclassid = mca.modelclassid
    and mca.rolename in (Select a.member_in
                            FROM rolesmembershipview a
                            WHERE var_modelclassacl_row.rolename = a.role_name::name);
If Found Then
    var_modelclassacl_row.deleteright = false;
Else
    --Test if there are any deleterights granted on the modelclass
    Yes -> set new deleteright true
    No -> do nothing
Perform * From modelclassacl mca
    where mca.deleteright = true
    and var_modelclassacl_row.modelclassid = mca.modelclassid
    and mca.rolename in (Select a.member_in
                            FROM rolesmembershipview a
                            WHERE var_modelclassacl_row.rolename = a.role_name::name);
If Found Then
    var_modelclassacl_row.deleteright = true;
End if; --new deleteright granted
End if; --new deleteright denied
End if; --old deleteright denied
--go up in modelclasshierarchy
--Test if the modelclass has a parent note
Yes -> go up and use current value of a modelclassacl row for next iteration
Perform * From modelclasshierarchy mch
    where mch.childid = var_modelclassacl_row.modelclassid;
If Found Then
    var_modelclassid = var_modelclassacl_row.modelclassid; --stores current modelclassid in a var so that it get not lost
    --get parent modelclassid and access function with it
    Select Into var_modelclassacl_row.modelclassid mch.parentid
        From modelclasshierarchy mch
    Select Into var_modelclassacl_row * From
        modelclassacl_recursive_compute_function(var_modelclassacl_row);
    var_modelclassacl_row.modelclassid := var_modelclassid; --write the original modelclassid back in the result of effective rights computation
Else
    return next var_modelclassacl_row;
End if; --parent note test
Else
    --go up in modelclasshierarchy
    --Test if the modelclass has a parent note
    Yes -> go up and use current value of a modelclassacl row for next iteration
    Perform * From modelclasshierarchy mch
        where mch.childid = var_modelclassacl_row.modelclassid;
    If Found Then
```
A.2 Trigger Function and Trigger

Listing A.2: The trigger function to compose the ArchitectureEffectiveACL

```sql
CREATE OR REPLACE FUNCTION architecture_effective_rights_computation()
RETURNS "trigger" AS
$BODY$
Declare
Begin
DELETE FROM architectureeffectiveacl;
INSERT INTO architectureeffectiveacl SELECT ia.rolename, a.architectureid, ia.selectright, ia.updateright, ia.deleteright
FROM interfaceacl ia, interface i, designentity de, architecture a
WHERE ia.interfaceid = i.interfaceid
AND i.interfaceid = de.interfaceid
AND de.architectureid = a.architectureid
AND NOT (a.architectureid IN (SELECT aa.architectureid
FROM architectureacl aa))
UNION
SELECT aa.rolename, aa.architectureid, aa.selectright, aa.updateright, aa.deleteright
FROM architectureacl aa;
Return NULL;
$BODY$
END;
```

LANGUAGE 'plpgsql' VOLATILE SECURITY DEFINER;

ALTER FUNCTION modelclassacl_recursive_compute_function(modelclassacl) OWNER TO mladmin;

GRANT EXECUTE ON FUNCTION modelclassacl_recursive_compute_function(modelclassacl) TO mladmin;

GRANT EXECUTE ON FUNCTION modelclassacl_recursive_compute_function(modelclassacl) TO public;

A.2 Trigger Function and Trigger
Listing A.3: The trigger on the InterfaceACL

```sql
CREATE TRIGGER architectureeffectiverightstrigger
AFTER INSERT OR UPDATE OR DELETE
ON interfaceacl
FOR EACH STATEMENT
EXECUTE PROCEDURE architecture_effective_rights_computation();
```

### A.3 ModelLib Database Tables

Table A.1: List of all tables in the meta information database

<table>
<thead>
<tr>
<th>Table name</th>
<th>Rights for basic roles</th>
<th>Related views and tables (not in the ERD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>mladmin</td>
<td>all mluser mlguest</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ArchitectureView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ArchitectureEffectiveACL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DesignentityEffectiveACL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ArchitectureACLView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ArchitectureAssertionView (via rule)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ArchitectureDocumentView (via rule)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AssertionView (via rule)</td>
</tr>
<tr>
<td>ArchitectureACL</td>
<td>mladmin</td>
<td>all mluser mlguest</td>
</tr>
<tr>
<td></td>
<td></td>
<td>architectureaclview</td>
</tr>
<tr>
<td></td>
<td></td>
<td>architectureview</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ArchitectureEffectiveACL</td>
</tr>
<tr>
<td>ArchitectureAssertion</td>
<td>mladmin</td>
<td>all mluser mlguest</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ArchitectureAssertionview</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AssertionView</td>
</tr>
<tr>
<td>ArchitectureDocument</td>
<td>mladmin</td>
<td>all mluser mlguest</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ArchitectureDocumentView</td>
</tr>
<tr>
<td>ArchitectureEffectiveACL</td>
<td>mladmin</td>
<td>all mluser mlguest</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ArchitectureView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DesignentityEffectiveACL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ArchitectureAssertionView (via rule)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ArchitectureDocumentView (via rule)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AssertionView (via rule)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ParameterArchitectureView</td>
</tr>
<tr>
<td>ArchitectureParameter</td>
<td>mladmin</td>
<td>all mluser mlguest</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ParameterArchitectureView</td>
</tr>
<tr>
<td>Assertion</td>
<td>mladmin</td>
<td>all mluser mlguest</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AssertionView</td>
</tr>
</tbody>
</table>

Rights: (S)elect, (U)pdate, (I)nsert, (D)elete
### Table name | Rights for basic roles | Related views and tables (not in the ERD)
---|---|---
Author | mldadmin, mluser, mlguest | all | ArchitectureDocumentView
| | | – | AuthorView
DesignEntity | mldadmin, mluser, mlguest | all | ArchitectureEffectiveACL
| | | – | DesignEntityEffectiveACL
| | | – | ArchitectureView
| | | – | DesignEntityView
| | | – | DesignEntityACLView (via rule)
| | | – | DesignEntityDependencyView (via rule)
| | | – | DesignEntityDocumentView (via rule)
| | | – | DesignEntityLicenseView (via rule)
| | | – | TestbenchDesignEntityView (via rule)
DesignEntityACL | mldadmin, mluser, mlguest | all | DesignEntityEffectiveACL
| | | – | DesignEntityACLView
| | | – | TestbenchDesignEntityView
DesignEntityDependency | mldadmin, mluser, mlguest | all | DesignEntityDependencyView
DesignEntityDocument | mldadmin, mluser, mlguest | all | DesignEntityDocumentView
DesignEntityEffectiveACL | mldadmin, mluser, mlguest | all | DesignEntityView
| | | – | DesignEntityDependencyView (via rule)
| | | – | DesignEntityDocumentView (via rule)
| | | – | DesignEntityLicenseView (via rule)
| | | – | TestbenchDesignEntityView (via rule)
DesignEntityFile | mldadmin, mluser, mlguest | all | DesignEntityFileView
| | | – | FileView
DesignEntityLicense | mldadmin, mluser, mlguest | all | DesignEntityLicenseView
DesignEntityMessage | mldadmin, mluser, mlguest | all | DesignEntityMessageView
DesignLanguage | mldadmin, mluser, mlguest | all | DesignLanguageVersionView
| | | S | DesignLanguageVersionView
DesignLanguageVersion | mldadmin, mluser | all | DesignLanguageVersionView
| | | S | DesignEntityView

Rights: (S)elect, (U)pdate, (I)nsert, (D)elete
<table>
<thead>
<tr>
<th>Table name</th>
<th>Rights for basic roles</th>
<th>Related views and tables (not in the ERD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mlguest</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>mluser</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>mlguest</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>DesignLanguageVersionMessage</td>
<td>mladmin</td>
<td>DesignLanguageVersionMessageView</td>
</tr>
<tr>
<td>mluser</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>mlguest</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>DesignToolVersion</td>
<td>mladmin</td>
<td>DesignToolVersionView</td>
</tr>
<tr>
<td>mluser</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>mlguest</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>DesignToolVersion-</td>
<td>mladmin</td>
<td>DesignToolVersionView</td>
</tr>
<tr>
<td>DesignLanguageVersion</td>
<td>mluser</td>
<td></td>
</tr>
<tr>
<td>mlguest</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>mluser</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>mlguest</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>DesignToolVersionDocument</td>
<td>mladmin</td>
<td>DesignToolVersionDocumentView</td>
</tr>
<tr>
<td>mluser</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>mlguest</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>DesignToolVersionMessage</td>
<td>mladmin</td>
<td>DesignToolVersionMessageView</td>
</tr>
<tr>
<td>mluser</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td>mlguest</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td>Document</td>
<td>mladmin</td>
<td>DocumentView</td>
</tr>
<tr>
<td>mluser</td>
<td>–</td>
<td>ArchitectureDocumentView (via rule)</td>
</tr>
<tr>
<td>mlguest</td>
<td>–</td>
<td>DesignEntityDocumentView (via rule)</td>
</tr>
<tr>
<td>DocumentACL</td>
<td>mladmin</td>
<td>DocumentACLView</td>
</tr>
<tr>
<td>mluser</td>
<td>–</td>
<td>ArchitectureDocumentView (via rule)</td>
</tr>
<tr>
<td>mlguest</td>
<td>–</td>
<td>DesignEntityDocumentView (via rule)</td>
</tr>
<tr>
<td>DocumentACLView</td>
<td>mladmin</td>
<td>DocumentACLView</td>
</tr>
<tr>
<td>mluser</td>
<td>–</td>
<td>ArchitectureDocumentView (via rule)</td>
</tr>
<tr>
<td>mlguest</td>
<td>–</td>
<td>DesignEntityDocumentView (via rule)</td>
</tr>
<tr>
<td>DocumentAuthorView (via rule)</td>
<td>mladmin</td>
<td>DocumentAuthorView</td>
</tr>
<tr>
<td>mluser</td>
<td>–</td>
<td>ArchitectureDocumentView (via rule)</td>
</tr>
<tr>
<td>mlguest</td>
<td>–</td>
<td>DesignEntityDocumentView (via rule)</td>
</tr>
<tr>
<td>DocumentKeywordView (via rule)</td>
<td>mladmin</td>
<td>DocumentKeywordView</td>
</tr>
<tr>
<td>mluser</td>
<td>–</td>
<td>ArchitectureDocumentView (via rule)</td>
</tr>
<tr>
<td>mlguest</td>
<td>–</td>
<td>DesignEntityDocumentView (via rule)</td>
</tr>
<tr>
<td>InterfaceDocumentView (via rule)</td>
<td>mladmin</td>
<td>InterfaceDocumentView</td>
</tr>
<tr>
<td>mluser</td>
<td>–</td>
<td>ArchitectureDocumentView (via rule)</td>
</tr>
<tr>
<td>mlguest</td>
<td>–</td>
<td>DesignEntityDocumentView (via rule)</td>
</tr>
<tr>
<td>DocumentDocumentTypeView (via rule)</td>
<td>mladmin</td>
<td>DocumentDocumentTypeView</td>
</tr>
<tr>
<td>mluser</td>
<td>–</td>
<td>ArchitectureDocumentView (via rule)</td>
</tr>
<tr>
<td>mlguest</td>
<td>–</td>
<td>DesignEntityDocumentView (via rule)</td>
</tr>
<tr>
<td>DocumentAuthorView (via rule)</td>
<td>mladmin</td>
<td>DocumentAuthorView</td>
</tr>
<tr>
<td>mluser</td>
<td>–</td>
<td>ArchitectureDocumentView (via rule)</td>
</tr>
<tr>
<td>mlguest</td>
<td>–</td>
<td>DesignEntityDocumentView (via rule)</td>
</tr>
<tr>
<td>DocumentKeywordView (via rule)</td>
<td>mladmin</td>
<td>DocumentKeywordView</td>
</tr>
<tr>
<td>mluser</td>
<td>–</td>
<td>ArchitectureDocumentView (via rule)</td>
</tr>
<tr>
<td>mlguest</td>
<td>–</td>
<td>DesignEntityDocumentView (via rule)</td>
</tr>
<tr>
<td>InterfaceDocumentView (via rule)</td>
<td>mladmin</td>
<td>InterfaceDocumentView</td>
</tr>
<tr>
<td>Rights: (S)elect, (U)update, (I)insert, (D)elete</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rights: (S)elect, (U)update, (I)insert, (D)elete
<table>
<thead>
<tr>
<th>Table name</th>
<th>Rights for basic roles</th>
<th>Related views and tables (not in the ERD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DocumentAuthor</td>
<td>mladmin</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td>mluser</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DocumentAuthorView</td>
</tr>
<tr>
<td>DocumentDocumentType</td>
<td>mladmin</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td>mluser</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DocumentDocumentTypeView</td>
</tr>
<tr>
<td>DocumentKeyword</td>
<td>mladmin</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td>mluser</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DocumentKeywordView</td>
</tr>
<tr>
<td>DocumentMessage</td>
<td>mladmin</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td>mluser</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DocumentMessageView</td>
</tr>
<tr>
<td>DocumentType</td>
<td>mladmin</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td>mluser</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FileView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FileACLView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DesignEntityFileView (via rule)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FileDesignToolVersionView (via rule)</td>
</tr>
<tr>
<td>FileACL</td>
<td>mladmin</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td>mluser</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FileView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FileACLView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DesignEntityFileView (via rule)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DesignEntityFileView (via rule)</td>
</tr>
<tr>
<td>FileDesignToolVersion</td>
<td>mladmin</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td>mluser</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FileDesignToolVersionView</td>
</tr>
<tr>
<td>Interface</td>
<td>mladmin</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td>mluser</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ArchitectureEffectiveACL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InterfaceView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InterfaceACLView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModelClassInterfaceView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AssertionView (via rule)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InterfaceAssertionView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InterfaceKeywordView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InterfaceMessageView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ParameterInterfaceView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Portview</td>
</tr>
<tr>
<td>InterfaceACL</td>
<td>mladmin</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td>mluser</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ArchitectureEffectiveACL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InterfaceView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InterfaceACLView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AssertionView (via rule)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModelClassInterfaceView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InterfaceAssertionView</td>
</tr>
</tbody>
</table>

Rights: (S)elect, (U)pdate, (I)nsert, (D)elete
### A.3 ModelLib Database Tables

<table>
<thead>
<tr>
<th>Table name</th>
<th>Rights for basic roles</th>
<th>Related views and tables (not in the ERD)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>InterfaceDocumentView (via rule)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InterfaceKeywordView (via rule)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ParameterInterfaceView (via rule)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Portview (via rule)</td>
</tr>
<tr>
<td>InterfaceAssertion</td>
<td>mladmin mluser mlguest</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td></td>
<td>–</td>
</tr>
<tr>
<td>InterfaceDocument</td>
<td>mladmin mluser mlguest</td>
<td>all</td>
</tr>
<tr>
<td>InterfaceKeyword</td>
<td>mladmin mluser mlguest</td>
<td>all</td>
</tr>
<tr>
<td>InterfaceParameter</td>
<td>mladmin mluser mlguest</td>
<td>all</td>
</tr>
<tr>
<td>Keyword</td>
<td>mladmin mluser mlguest</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td></td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>–</td>
</tr>
<tr>
<td>ModelClass</td>
<td>mladmin mluser mlguest</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td></td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>–</td>
</tr>
<tr>
<td>ModelClassACL</td>
<td>mladmin mluser mlguest</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td></td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>–</td>
</tr>
<tr>
<td>ModelClassHierarchy</td>
<td>mladmin mluser mlguest</td>
<td>all</td>
</tr>
<tr>
<td>ModelClassInterface</td>
<td>mladmin mluser mlguest</td>
<td>all</td>
</tr>
<tr>
<td>ModelClassKeyword</td>
<td>mladmin mluser mlguest</td>
<td>all</td>
</tr>
</tbody>
</table>

Rights: (S)elect, (U)update, (I)nsert, (D)elete
### A.4 ModelLib Database Views

Table A.2: List of all views in the meta information database

<table>
<thead>
<tr>
<th>View name</th>
<th>Rights for basic roles</th>
<th>Related views and tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArchitectureACLView</td>
<td>mladmin</td>
<td>all S U I D</td>
</tr>
<tr>
<td></td>
<td>mluser</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>–</td>
</tr>
<tr>
<td>ArchitectureAssertionView</td>
<td>mladmin</td>
<td>all S U I D</td>
</tr>
<tr>
<td></td>
<td>mluser</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>–</td>
</tr>
<tr>
<td>ArchitectureDocumentView</td>
<td>mladmin</td>
<td>all S U I D</td>
</tr>
<tr>
<td></td>
<td>mluser</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>S</td>
</tr>
<tr>
<td>ArchitectureView</td>
<td>mladmin</td>
<td>all S U I D</td>
</tr>
<tr>
<td></td>
<td>mluser</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>S</td>
</tr>
<tr>
<td>DesignToolVersionView</td>
<td>mladmin</td>
<td>all S U I D</td>
</tr>
<tr>
<td></td>
<td>mluser</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>S</td>
</tr>
</tbody>
</table>

Rights: (S)elect, (U)date, (I)nsert, (D)elete
<table>
<thead>
<tr>
<th>View name</th>
<th>Rights for basic roles</th>
<th>Related views and tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>mlguest</td>
<td>S</td>
<td>AssertionView DesignEntityView ParameterArchitectureView</td>
</tr>
<tr>
<td>mladmin mluser mlguest</td>
<td>all</td>
<td></td>
</tr>
<tr>
<td>AuthorView</td>
<td>mladmin mluser mlguest</td>
<td>all S U I D S</td>
</tr>
<tr>
<td>DesignEntityACLView</td>
<td>mladmin mluser mlguest</td>
<td>all S U I D</td>
</tr>
<tr>
<td>mladmin mluser mlguest</td>
<td>all</td>
<td></td>
</tr>
<tr>
<td>DesignEntityDependencyView</td>
<td>mladmin mluser mlguest</td>
<td>all S U I D</td>
</tr>
<tr>
<td>mladmin mluser mlguest</td>
<td>all</td>
<td></td>
</tr>
<tr>
<td>DesignEntityDocumentView</td>
<td>mladmin mluser mlguest</td>
<td>all S U I D</td>
</tr>
<tr>
<td>mladmin mluser mlguest</td>
<td>all</td>
<td></td>
</tr>
<tr>
<td>DesignEntityFileView</td>
<td>mladmin mluser mlguest</td>
<td>all S U I D</td>
</tr>
<tr>
<td>mladmin mluser mlguest</td>
<td>all</td>
<td></td>
</tr>
<tr>
<td>DesignEntityLicenseView</td>
<td>mladmin mluser mlguest</td>
<td>all S U I D</td>
</tr>
<tr>
<td>mladmin mluser mlguest</td>
<td>all</td>
<td></td>
</tr>
<tr>
<td>DesignEntityMessageView</td>
<td>mladmin mluser mlguest</td>
<td>all S U I D</td>
</tr>
<tr>
<td>mladmin mluser mlguest</td>
<td>all</td>
<td></td>
</tr>
<tr>
<td>DesignEntityView</td>
<td>mladmin mluser mlguest</td>
<td>all S U I D</td>
</tr>
<tr>
<td>DesignLanguage-</td>
<td>mladmin mluser mlguest</td>
<td>all S U I D</td>
</tr>
<tr>
<td>VersionMessageView</td>
<td>mladmin mluser mlguest</td>
<td></td>
</tr>
<tr>
<td>DesignLanguageVersionView</td>
<td>mladmin mluser mlguest</td>
<td>all S U I D</td>
</tr>
<tr>
<td>DesignToolVersion-</td>
<td>mladmin mluser mlguest</td>
<td>all S U I D</td>
</tr>
<tr>
<td>DesignLanguageVersion-</td>
<td>mladmin mluser mlguest</td>
<td>all S U I D</td>
</tr>
<tr>
<td>Rights: (S)elect, (U)date, (I)nsert, (D)elete</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Additional Source Code of the Implementation

<table>
<thead>
<tr>
<th>View name</th>
<th>Rights for basic roles</th>
<th>Related views and tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>MessageView</td>
<td>mlguest</td>
<td>S</td>
</tr>
<tr>
<td>DesignToolVersion-</td>
<td>mladmin</td>
<td>all</td>
</tr>
<tr>
<td>MessageView</td>
<td>mluser</td>
<td>S U I D</td>
</tr>
<tr>
<td>mlguest</td>
<td>all</td>
<td>S</td>
</tr>
<tr>
<td>DesignToolVersionView</td>
<td>mladmin</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td>mluser</td>
<td>S U I D</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>–</td>
</tr>
<tr>
<td>DocumentACLView</td>
<td>mladmin</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td>mluser</td>
<td>S U I D</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>–</td>
</tr>
<tr>
<td>DocumentAuthorView</td>
<td>mladmin</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td>mluser</td>
<td>S U I D</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>S</td>
</tr>
<tr>
<td>Document-DocumentTypeView</td>
<td>mladmin</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td>mluser</td>
<td>S U I D</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>–</td>
</tr>
<tr>
<td>DocumentKeywordView</td>
<td>mladmin</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td>mluser</td>
<td>S U I D</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>S</td>
</tr>
<tr>
<td>DocumentMessageView</td>
<td>mladmin</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td>mluser</td>
<td>S U I D</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>S</td>
</tr>
<tr>
<td>DocumentView</td>
<td>mladmin</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td>mluser</td>
<td>S U I D</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ArchitectureDocumentView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DesignEntitsDocumentView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DesignEntity-LicenseView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Document-DocumentTypeView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DocumentKeywordView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DocumentMessageView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InterfaceDocumentView</td>
</tr>
<tr>
<td>FileACLView</td>
<td>mladmin</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td>mluser</td>
<td>S U I D</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>–</td>
</tr>
<tr>
<td>FileDesignTool-</td>
<td>mladmin</td>
<td>all</td>
</tr>
<tr>
<td>VersionView</td>
<td>mluser</td>
<td>S U I D</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>–</td>
</tr>
<tr>
<td>FileView</td>
<td>mladmin</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td>mluser</td>
<td>S U I D</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DesignEntityFileView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FileDesignTool-VersionView</td>
</tr>
<tr>
<td>InterfaceACLView</td>
<td>mladmin</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td>mluser</td>
<td>S U I D</td>
</tr>
<tr>
<td></td>
<td>mlguest</td>
<td>–</td>
</tr>
</tbody>
</table>

Rights: (S)elect, (U)pdate, (I)nsert, (D)elete
## A.4 ModelLib Database Views

<table>
<thead>
<tr>
<th>View name</th>
<th>Rights for basic roles</th>
<th>Related views and tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>InterfaceAssertionView</td>
<td>mladmin mluser mlguest</td>
<td>all S U I D S</td>
</tr>
<tr>
<td>InterfaceDocumentView</td>
<td>mladmin mluser mlguest</td>
<td>all S U I D S</td>
</tr>
<tr>
<td>InterfaceKeywordView</td>
<td>mladmin mluser mlguest</td>
<td>all S U I D S</td>
</tr>
<tr>
<td>InterfaceView</td>
<td>mladmin mluser mlguest</td>
<td>all S U I D S</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ArchitectureView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AssertionView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DesignEntityView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InterfaceAssertionView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InterfaceDocumentView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InterfaceKeywordView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModelClassInterfaceView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ParameterInterfaceView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PortView</td>
</tr>
<tr>
<td>MessageView</td>
<td>mladmin mluser mlguest</td>
<td>all S U I D S</td>
</tr>
<tr>
<td>ModelClassACLView</td>
<td>mladmin mluser mlguest</td>
<td>all S U I D S</td>
</tr>
<tr>
<td>ModelClassEffectiveACLView</td>
<td>mladmin mluser mlguest</td>
<td>all S –</td>
</tr>
<tr>
<td>ModelClassHierarchyView</td>
<td>mladmin mluser mlguest</td>
<td>all S U I D S</td>
</tr>
<tr>
<td>ModelClassInterfaceView</td>
<td>mladmin mluser mlguest</td>
<td>all S U I D S</td>
</tr>
<tr>
<td>ModelClassKeywordView</td>
<td>mladmin mluser mlguest</td>
<td>all S U I D S</td>
</tr>
<tr>
<td>ModelClassView</td>
<td>mladmin mluser mlguest</td>
<td>all S U I D S</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModelClassHierarchyView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModelClassInterfaceView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModelClassKeywordView</td>
</tr>
<tr>
<td>Rights: (S)elect, (U)update, (I)nsert, (D)elete</td>
<td></td>
<td></td>
</tr>
<tr>
<td>View name</td>
<td>Rights for basic roles</td>
<td>Related views and tables</td>
</tr>
<tr>
<td>------------------------------</td>
<td>------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>mlguest</td>
<td>S</td>
<td>ModelClassKeywordView</td>
</tr>
<tr>
<td>ParameterArchitectureView</td>
<td>mladmin all S U I D</td>
<td></td>
</tr>
<tr>
<td>ParameterInterfaceView</td>
<td>mluser all S U I D S</td>
<td></td>
</tr>
<tr>
<td>PortView</td>
<td>mladmin all S U I D</td>
<td></td>
</tr>
<tr>
<td>RolesMembershipView</td>
<td>mladmin all – –</td>
<td>see Table A.3</td>
</tr>
<tr>
<td>Testbench-DesignEntityView</td>
<td>mladmin all S U I D S</td>
<td></td>
</tr>
<tr>
<td>UserDirectoryView</td>
<td>mladmin all S U I D S</td>
<td></td>
</tr>
<tr>
<td>UserInformationView</td>
<td>mladmin all –</td>
<td></td>
</tr>
</tbody>
</table>

Rights: (S)elect, (U)pdate, (I)nsert, (D)elete
Table A.3: Table of views depending on the RolesMembershipView

<table>
<thead>
<tr>
<th>Views depending on the RolesMembershipView</th>
</tr>
</thead>
<tbody>
<tr>
<td>InterfaceView</td>
</tr>
<tr>
<td>ArchitectureView</td>
</tr>
<tr>
<td>DocumentView</td>
</tr>
<tr>
<td>DesignEntityView</td>
</tr>
<tr>
<td>FileView</td>
</tr>
<tr>
<td>ArchitectureAssertionView</td>
</tr>
<tr>
<td>ArchitectureDocumentView</td>
</tr>
<tr>
<td>AssertionView</td>
</tr>
<tr>
<td>DesignEntityDependencyView</td>
</tr>
<tr>
<td>DesignEntityDocumentView</td>
</tr>
<tr>
<td>DesignEntityFileView</td>
</tr>
<tr>
<td>DesignEntityLicenseView</td>
</tr>
<tr>
<td>DesignEntityDocumentView</td>
</tr>
<tr>
<td>DesignEntityDocumentView</td>
</tr>
<tr>
<td>DesignEntityFileView</td>
</tr>
<tr>
<td>DesignEntityLicenseView</td>
</tr>
<tr>
<td>DesignEntityDocumentView</td>
</tr>
<tr>
<td>DocumentAuthorView</td>
</tr>
<tr>
<td>DocumentDocumentTypeView</td>
</tr>
<tr>
<td>DocumentKeywordView</td>
</tr>
<tr>
<td>FileDesignToolVersionView</td>
</tr>
<tr>
<td>InterfaceAssertionView</td>
</tr>
<tr>
<td>InterfaceDocumentView</td>
</tr>
<tr>
<td>InterfaceKeywordView</td>
</tr>
<tr>
<td>ModelClassInterfaceView</td>
</tr>
<tr>
<td>ParameterArchitectureView</td>
</tr>
<tr>
<td>ParameterInterfaceView</td>
</tr>
<tr>
<td>PortView</td>
</tr>
<tr>
<td>TestbenchDesignEntityView</td>
</tr>
</tbody>
</table>
Bibliography


Bibliography

[HS97] Heuer, Andreas und Gunter Saake: *Datenbanken – Konzepte und Sprachen*. Interna-


[JBC+06] Jendrock, Eric, Jennifer Ball, Debbie Carson, Ian Evans, Scott Fordin, and Kim Haase: *

visited on 27/03/2006.

[Kec05] Kecher, Christoph: *UML-2.0 – Das umfassende Handbuch*. Galileo Press GmbH,
Bonn, 2005.

[KMP+04] Kappel, Gerti, Elke Michlmayr, Birgit Pröll, Siegfried Reich, and Werner Retschitzeg-
ger: *Web engineering – old wine in new bottles?* In Koch, N., P. Fraternale, and
M. Wirsing (editors): *Web Engineering 4th International Conference, ICWE 2004,
Munich, Germany, July 26-30, 2004, Proceedings*, volume 3140 of Lecture Notes in

[KPRR06] Kappel, Gerti, Birgit Pröll, Siegfried Reich, and Werner Retschitzegger: *Web Engi-
neering – The Discipline of Systematic Development of Web Application*. John Wiley

[KS06] Keith, Mike and Merrick Schincariol: *Pro EJB 3 – Java Persistence API*. Apress,

[MV06] Mähne, Torsten and Alain Vachoux: *ModelLib: A web-based platform for col-
lecting behavioural models and supporting the design of ams systems*. In *Fo-


visited on 13/07/2006.

visited on 24/08/2006.

[OTDRG05] O’Connor, I., F. Tissafi-Drissi, G. Révy, and F. Gaffiot: *UML/XML-based app-
roach to hierarchical AMS synthesis*. In *Forum on Specification and Design Lan-
on 10/03/2006.
**Bibliography**

<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
</table>
Statement

I hereby declare that I created this work alone and by using only legal means.

Lausanne, 30th January 2007

Thomas Böhm