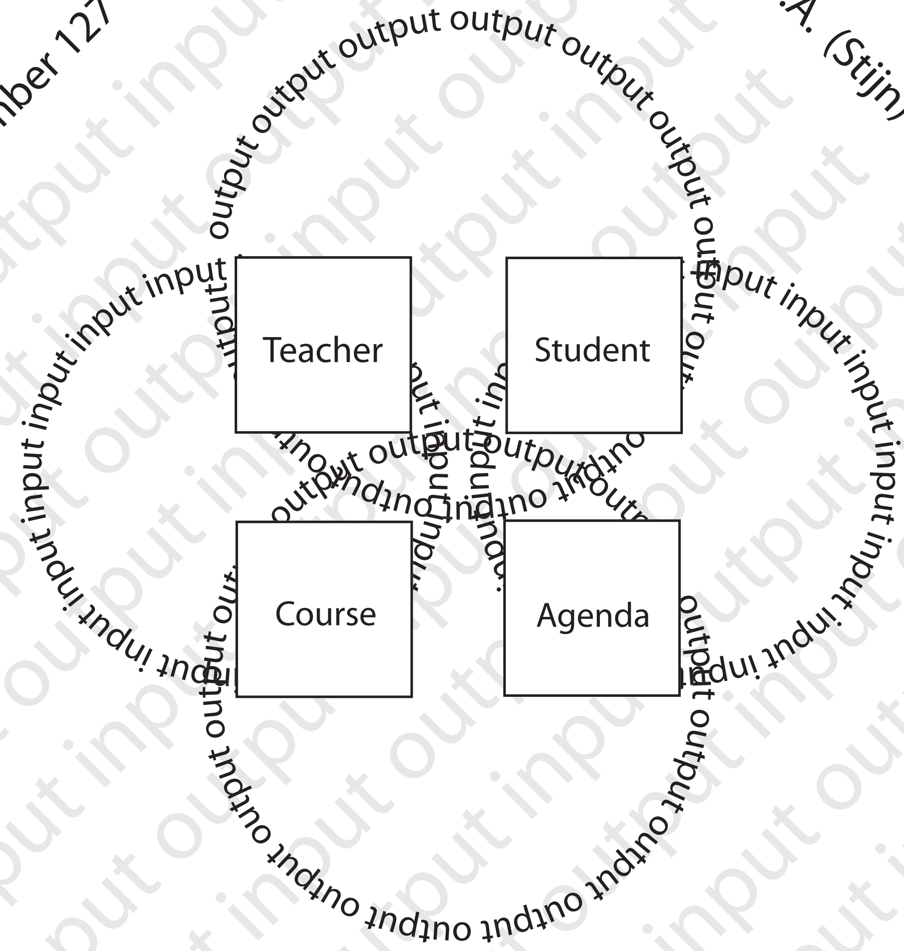
Generating a rule-based modelling agenda
Master thesis Information Science
Author: J.S.M. (Jörg) Mutter

July 2010

Thesis number 127 IK

Supervisor: dr. S.J.B.A. (Stijn) Hoppenbrouwers



Institute for Computing and Information Sciences
Radboud University Nijmegen

Preface

All things come to an end *- Geoffrey Chaucer, English poet*

This thesis will be the final chapter in my life as a student. It has been a long road, with ups and downs, which has taken me through the beautiful world of Artificial Intelligence, before ending in the realm of Information Science. Though sometimes this road seemed endless, I never stopped wandering and now, with the finish in sight, it is time to to say ”Thank you”.

Thank you, to Stijn Hoppenbrouwers, for being my supervisor, for being my sparring partner during my research and for your infinite patience with me. If I'd ever had to write another thesis, I know I'd like always want have you as my supervisor again.

Thank you, to my family and friends, for your unconditional support in all the choices I made, and a special thank you, to my mother, for designing this beautiful front page.

I hope you will enjoy reading this thesis.

July 4th, 2010 Jörg Mutter

Contents

1 Introduction

Models help us deal with the complexity around us every day. In engineering, scale models of buildings, aeroplanes and ships are used to test their ability to cope with the elements they meet. These models can be seen as an actual (partial) representation of what may be built. In the Middle Ages, maps were drawn for seafareres, representing coastal lines and cities on them. Today, we have little devices in our cars that tell us where we have to go in order to reach our destination, replacing the old paper street maps.

Models come in various shapes and forms. In many cases, like scale model building, the rules for making these models are more or less clear. Scale models must represent reality as closely as possible, otherwise they are useless as testing material. Likewise, in cartography, every map must be drawn on a certain scale, which is the same for the whole map. This precision prevents ships from running aground and makes city planning possible.

In computer science, models are used to make graphic representations of complex businesses. Models provide a means of understanding the complex processes and domain knowledge that exist in many companies. They are widely used as a means of verifying the understanding of what the business is all about and to communicate to IT specialists how computers can help optimize the business processes.

When different people talk about a specific domain, they invariably use some form of domain specific language. The words they use have a specific meaning in the context of the domain. Adding to the complexity, they may also use synonyms (different words having the same meaning) and homonyms (one word having different meanings). Understanding these subtleties and translating natural language into a formal model is one of the major challenges for modellers these days.

Not only the use of natural language poses problems to the modeller. To ensure that models are corect, they have to adhere to a certain syntax, leaving little to no room for more than one interpretation. These syntactic rules form the basis for creating, reading and understanding all kinds of models. Without them, models are more or less useless in computer science.

Modelling nowadays is still rather art than science. Interactions with domain experts lead to models, which have to be checked and discussed in length to ensure their correctness. These interactions lead to changes in the model. The model itself has to adhere to the rules of the modelling language, ensuring syntactical soundness and correctness.

In order to transform this modelling art into science, a broader understanding of the modelling process is essential. While much literature can be found on model syntax, semantics and quality, the modelling process itself typically stays underexposed. What drives modellers, what are their options while creating a model and why do they choose one option instead of another?

Some work of interest on this topic has been carried out by Lindeman, who provides us with a functional decomposition of the modelling process, describing the different activities, their role in the modelling process and the order in which they are carried out [Lindeman06]. Her work describes the broader modelling process, from identifying the contracter [[[up to]]] writing the documentation.

The research this thesis describes aims at getting a more detailed insight into the process of creating models. The theoretical part describes a framework linking product and process. The practical assignment aims at getting a more detailed insight into the process of modelling. In order to do so, the *modelling agenda* will be introduced, a computer program that helps modellers create correct models.

The modelling agenda strongly supports the modelling framework as described by Ssebuggwawo et al. [Ssebuggwawo09], which will be discussed further on. In particular, the modelling agenda supports the principle that rules influence models and that models lead to new rules. It also demonstrates the influence of rules on interactions and how these interactions form models.

1.1 Research questions

The questions that will be answered in this thesis will be the following:

1. *How can a modelling agenda be generated based on syntactical rules, procedural rules and a partial model?*

   Subquestions that will be answered are:

   (a) *What is a modelling agenda?*

   (b) *What are procedural rules of modelling?*

2. *Provide a proof of concept.*

1.2 Methodology

This thesis combines a literature study with a proof of concept. This proof of concept will be the modelling agenda, a program that will check a model during creation using declarative metamodelling rules. When not all metamodelling rules are satisfied, the program will create a set of goals for the modeller (the agenda) that have to be fulfilled in order to achieve the main goal: a syntactically sound and correct model (leading to an empty agenda).

The research will be performed top-down and bottom-up, incorporating new insights from the proof of concept along the way.

1.3 Scope

As mentioned above, models come in all sorts of forms. The scope of this research has been limited to Object Role Modelling, ORM for short. This modelling language has been chosen for its quality of being close to natural language. This contributes to an easily understandable dialogue between modeller and program.

1.4 Reading guide

This thesis is devided into two parts: the literature study in chapter 2 and the proof of concept in chapter 3. Chapter 4 answers the above research questions and contains some remarks on future work. The computer code of the modelling agenda can be found in the addendum.

2 Models, Interactions and Rules

2.1 Introduction

In [Ssebuggwawo09], Ssebuggwawo et al. describe an analytical framework and approach for analysing interactions, rules and models. They hypothesize that:

"The interactions that take place in collaborative modeling sessions can be seen as games with players who may either explicitly or implicitly determine and play by rules of a modelling game".

Their hypothesis is at the heart of this research. The modelling agenda program supports the modelling process in the form of a dialogue game. The interactions that take place between the modeller and the program eventually form the model. The interactions are restricted by the meta-model rules. These meta-model rules apply to the model at hand, while building the model leads to more (or less) modelling goals on the agenda.

This modelling framework as an interplay between interactions, rules and models is schematically depicted in figure 1.

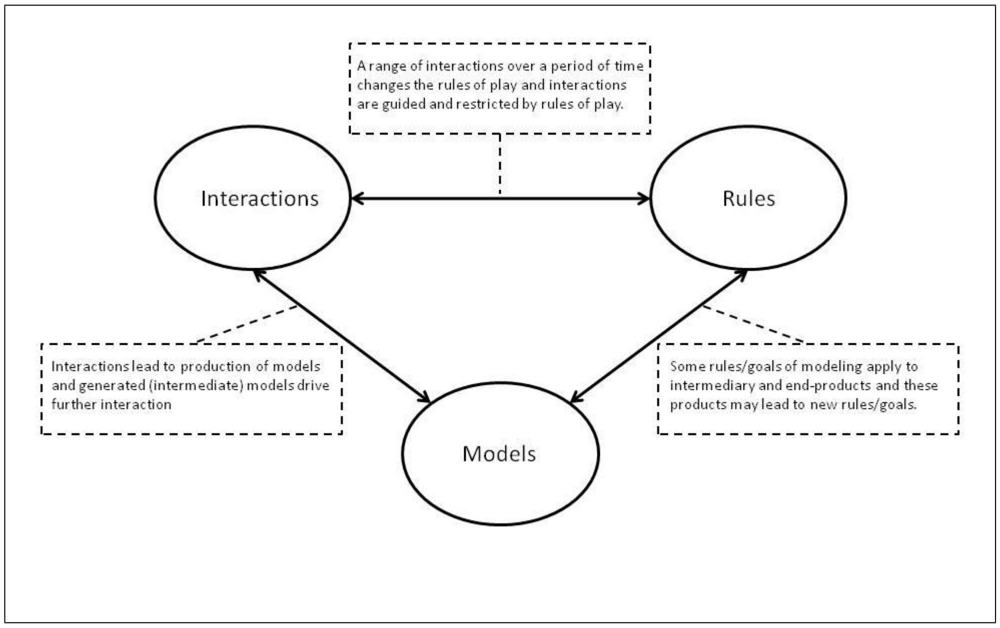

A range of interactions over a period of time changes the rules of play and interactions are guided and restricted by rules of play.
Interactions
Rules
Interactions lead to production of models and generated (intermediate) models drive further interaction
Some rules/goals of modeling apply to intermediary and end-products and these products may lead to new rules/goals.
Models

Figure 1: Interactions, rules and models

2.2 Models

Models help us deal with the complexity of the reality around us. A model may describe (part of) a (software) system, seen from a particular point of view. Models help us explain, understand and/or design a systems behaviour.

Conceptual modelling is a software development activity aiming at constructing a complete and unambiguous model of that part of reality, relevant to the software system in development. During the conceptual modelling phase, the problem domain is unravelled and described. No decisions are taken as to how the problem should be solved technically. The conceptual model forms the foundation for the rest of the software development process.

In object oriented conceptual modelling, reality is modelled as a collection of linked and interacting objects, uniting both the structural aspects and the behavioural aspects of the problem domain. One language well suited for this purpose is Object Role Modelling, or ORM. As ORM plays a significant role in this research, a short introduction will be given below. A full description of ORM can be found in [Halpin98].

2.2.1 Object Role Modelling

Object Role Modelling (ORM) is a fact oriented method, which has initially been developed for modelling information systems at a conceptual level. ORM makes use of natural language statements by examinig them in terms of elementary facts.

In [Halpin98] a *Conceptual Schema Design Procedure* (CSDP) is provided, consisting of seven steps:

1. Transform familiar information examples into elementary facts, and apply quality checks.
2. Draw the fact types, and apply a quality check.
3. Check for entity types that should be combined, and note any arithmetic derivations.
4. Add uniqueness constraints, and check arity of fact types.
5. Add mandatory role constraints, and check for logical derivations.
6. Add value, set comparison and sub-typing constraints.
7. Add other constraints and perform final checks.

This CSDP captures the idea of a modelling process for ORM, but it is only one of many ways to create a syntactically correct ORM-model. One might for example want to initially add only entity types, and connect those by roles later on in the process to form fact types. Others might want to start by adding fact instances to the model en derive the corresponding fact types from there on.

2.2.2 Metamodelling

In order to support the modelling process, a precise description of the modelling language is of great use. This description is best captured in a metamodel of the language. As Clark et al. point out: a metamodel is a model of a modelling language, that captures its essential properties and features [Clark04].

A metamodel thus describes the syntax of a modelling language. As will be demonstrated, a metamodel can then be transformed into declarative rules. These declarative rules constitute a major part of the modelling agenda program this thesis deals with.

2.3 Interactions

The process of modelling typically involves more than one person [Zanten01]. Usually, models are created by one or more domain experts and one or more modelling experts. The modelling experts function as facilitators, translating the domain experts' knowledge into a conceptual model.

The act of modelling can than be seen as a succession of interactions. Interactions may lead to statements on the domain by the domain expert, which can be translated by the modeller into model elements. The resulting model may lead to more interactions, further shaping the model.

As Ssebuggwawo et al. point out, modellers also interact and communicate their ideas and opinions to each other. To reach consensus and agreement, they need to abide to their collective knowledge, conventions and decisisons (rules of their game) [Ssebuggwawo09]. These interactions are beyond the scope of this reasearch, which is merely concerned with the actual model building.

2.3.1 Collaborative modelling

Collaborative modelling is the process, in which domain experts and modellers work together to perform some modelling task. In [Hoppenbrouwers06], Hoppenbrouwers et al. describe modelling processes as in principle involving two related dialogues: one for elicitation and one for formalization. The elicitation dialogue takes place between one or more informers (domain experts) and one or more model mediators (typically information analysts). The formalization dialogue, on the other hand, takes place between the model mediator and the model builder (usually some tool used to capture and verify the actual model).

The role of the mediator is thus to interface between the two dialogues. The elicitation dialogue takes place in normal language; for the formalisation dialogue, some form of controlled language is taken as a key means of bridging the gap between informal and formal.

The formalisation dialogue is of most interest for the modelling agenda program. Interactions between modeller and program will take place in a very simple controlled language. This language has been designed to be understandable for domain experts as well as for a computer. The controlled language rules will be described further on.

2.3.2 Weak workflows

Collaborative modelling is a somewhat ad hoc process, which makes it very hard to support it with a software program. Highly structured processes can often quite easily be captured and supported by workflow systems. Decomposition of these processes often reveals process steps in the form of: ”Task A always leads to Task B”. Tasks in knowledge intensive processes often rely on information gathered in an earlier part of the process: ”Task A delivers information I. Given information I, task B may be skipped and task C may commence.”

Modelling is in fact a weakly-structured, or weak workflow. In [Adsett04], Adsett et al. provide a clear definition of weak workflows: ”A weak workflow within an organization’s information system allows processes to be defined as they are being performed. It requires general knowledge about the organization to be dynamically combined with specific information about a current workflow.”

In [Elst03], Van der Elst et al. describe the characteristics of weak workflows. These characteristics form the basic requirements for the modelling agenda:

1. Lazy and late modelling are supported:
   A modeller may start with a possibly incomplete model and refine it later. He may not have all relevant information at hand during the modelling process, or he may not know to which extent the model should be refined.

2. Modelling and execution of process-models is interleaved:
   In order to be able to already use a partial model it should be possible to work also on the process model while it is being executed instead of having modelling and execution in two distinct phases (like in traditional workflow approaches).

3. Tasks can be dynamically and hierarchically decomposed/refined:
   A typical phenomenon in knowledge work is the abstract notice that a specific subtask is necessary, while the single steps of this subtask are unknown in advance. This is a key characteristic for lazy and late modelling.

4. A rich process logic allows for expressive process representations:
   In knowledge-intensive tasks the execution sequence is often highly dependant from information gathered in earlier processing steps. The traditional modelling of task sequences may not be suffcient. Instead, constraint-like descriptions of pre- and post-conditions of individual tasks and appropriate reasoning mechanisms allow for the dynamic configuration of the work process at runtime.

2.3.3 Dialogue systems

The modelling agenda, in order to support the weakly-structured modelling process, is highly dependent on the information it receives from the modeller.

Any information provided may either lead to new questions, or answer one or more of the existing questions. This interactivity is best captured by a dialogue system.

A dialogue system is a computer system, intended to interact using human language. It gets a predefined command from the modeller, and performs the action (or actions) linked to that command. These commands may eventually be replaced by a more graphical way of modelling.

2.3.4 Dialogue games

Dialogue systems have long been used in computer games, before they were more or less replaced by joystick- and mouse-controlled games. Typical examples include the adventure games from Sierra On-Line, like Space Quest, Police Quest and Kings Quest.

In these games, the player feeds natural language commands to the computer. The computer then parses the command (stripping information like articles and prepositions) and performs some task, or it warns the player that the command was not properly understood.

In artificial intelligence, dialogue games are sometimes used when dealing with incomplete or inconsistent data, like in [Lebbink03] and [Bryant05]. Prakken [Prakken05] mentions: Dialogue systems define the principles of coherent dialogue, referring to Carlson [Carlson83] who defines coherence in terms of the goal of a dialogue. Utterances are coherent when they further the goal of the dialogue in which they are made.

The power of a dialogue game as a support tool for modelling is demonstrated by Ravenscroft et al. Their program, Interloc (collaborative *inter*action through scaffolding *loc*utions) [Ravenscroft06], supports reasoning with incomplete and possibly inconsistent data by multiple persons. One can easily apply this to a modelling session, where modellers may propose adding or changing model elements. Their propositions may be challenged or accepted by other participants.

This method also supports the weak modelling workflow, but leaves the issue of soundness and completeness of the model entirely to the participants. One can imagine the modelling agenda as an extra, automatic participant, adding propositions where necessary in order to keep the model sound and complete.

2.3.5 Modelling as a game

The act of modelling is in essence a dialogue game [Hoppenbrouwers08], in which the actors constantly deal with incomplete and sometimes in-coherent data. The model mediators must gather as much information as possible and transform all this information into a coherent set, to form a model.

When we look at modelling as a game, the main goal of the game is to create a correct and complete model. Subgoals may be defined, not only by the players, but also by the modelling rules (syntactical goals). These syntactical goals form new rules, that have to be followed to come to a correct model. The next section takes a closer look at these rules.

2.4 Rules

In the framework proposed by Ssebuggwawo et al. rules influence models and models influence rules. The rules that come from models are in fact modelling goals. Goals are viewed as a key type of rule (”goal rules”), setting states to strive for [Ssebuggwawo09]. These goal rules also have some influence on the strategy of the modelling process. The syntactical rules of the modelling language (the rules that influence the models) can be captured in declarative rules.

2.4.1 Strategic rules

In [Bommel08], Van Bommel et al. describe a rule-based conceptual framework for capturing strategies of modelling, in the process of obtaining conceptual models. Their paper concerns specifically meta-model driven strategies, aiming to fulfil modelling goals or obligations that are the direct result of meta-model choices. These meta-model oriented goals are imposed by the modelling language, and the correctness of the statements made in it. If the modelling language is strictly defined, these goals are at the core of the strategies that lead to well-formed models.

Modelling languages impose requirements not only on the structure of the model, but also on the modelling process. In ORM for example, theoretically it might be possible for a role to exist on its own, but this would not likely describe a real world concept. Therefore a modeller might want to add the requirement that every entity type belongs to at least one fact type, which links it to another entity type via a relation.

Strategic rules also influence the process of modelling in time, some goals having to be fulfilled before going on to the next. In the modelling agenda, goals (derived from the incomplete model) will be represented as questions on the agenda. As will be demonstrated, prioritizing these questions is one of the programs functions.

2.4.2 Business rules

The syntactical modelling rules can be described as business rules. In his Business Rules Manifesto [Ross03], Ross provides a clear definition of business rules in the form of 10 articles. Especially noteworthy in relation to the modelling agenda are article 2.1 , article 4.1 and article 5.3:

Article 2.1 - Rules are explicit constraints on behavior and/or provide support to behavior

Article 4.1 - Rules should be expressed declaratively in natural language sentences for the business audience

Article 5.3 - Formal logics, such as predicate logic, are fundamental to well-formed expression of rules in business terms, as well as to the technologies that implement business rules

2.5 Conclusion

This chapter has described the relations between models, interactions and rules. Some remarks have been made on (meta)models and a small overview of Object Role Modelling has been presented. The act of modelling has been defined as a succession of interactions between domain expert, modeller and possibly a mediator. Modellers may have their own modelling strategies, which can be captured in business rules. The act of modelling has been identified as a weak workflow, which may be supported by an interactive dialogue system or a (dialogue) game.

3 Design Process

3.1 Introduction: Design Science

This thesis, consisting of a literature study as well as a proof of concept, is an example of design science. Design science has been practiced in computer science for decades, but has been lost during the last 25 years. In his essay, Iivari discusses the ontology, epistemology, methodology and ethics of design science [Iivari07].

In a reaction to Iivari's article, Hevner [Hevner07] poses a design science research cycle, consisting of the Relevance cycle, the Rigor cycle and at the heart the Design cycle.The relevance cycle sets the context for the design science by providing requirements and acceptance criteria for the research results. The rigor cycle adds past knowledge to the research, to ensure its innovative power. The design cycle iterates between construction, evaluation and further design. The design science research cycle is depicted in figure 2.

The knowledge base for this research has been provided in chapter 2. This thesis is a derivative of the framework for interactions, rules and models [Ssebuggwawo09]. The following sections will therefore be devoted to the relevance cycle and the design cycle.

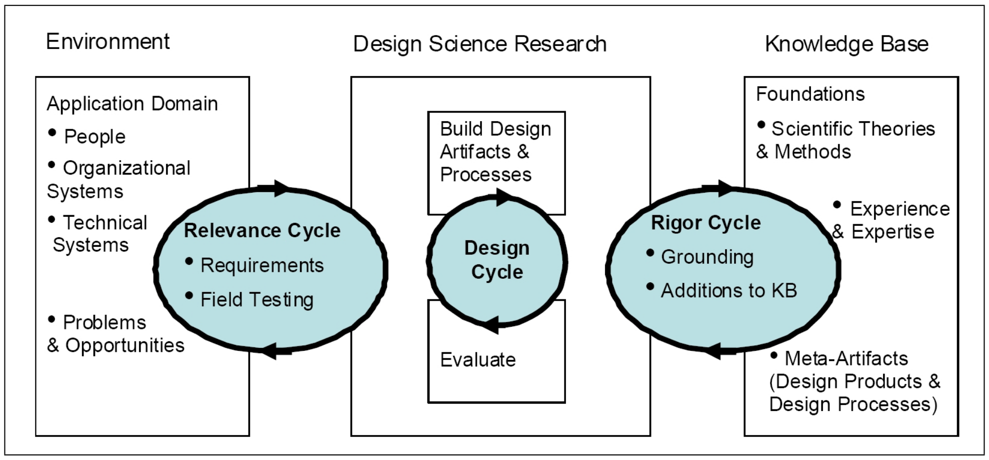


Figure 2: Design science research cycle

3.2 Relevance cycle

In this section, the scope and the requirements for the modelling agenda will be described.

3.2.1 Scope

One modelling language

The goal of the modelling agenda is to help the modeller in designing a syntactically correct model. Any modeller may have his own preferences regarding the modelling language. The modelling agenda should in principle not be limited to one single modelling language. For the proof of concept however, the choice was made to incorporate only one language, Object Role Modelling (ORM).

ORM was chosen because its models are derived from natural language statements. On the one hand, the use of natural language facilitates the elicitation dialogue between domain expert and modeller. The formalization dialogue on the other hand is a form of controlled natural language, as are the ORM statements.

Only basic elements of ORM

ORM is a very rich modelling language. It comprises many types of constraints, allows for subtyping, objectification and many other things. The modelling agenda will not support the full ORM language. Firstly because many model elements are overlapping, i.e. one element can be substituted by one or more other model elements. Secondly, many elements are optional in ORM and therefore models can be syntactically sound and complete without them (syntactical soundness and completeness are the main goal of the modelling agenda).

The modelling agenda will therefore initially support only the following model elements:

- Fact types
- Fact instances
- Object types
- Object instances
- Roles
- Uniqueness constraints

Note that this list may be extended with any model element of choice when needed, this however is beyond the scope of this proof of concept.

Limited temporal ordering

To prove the concept of temporal ordering, the choice was made to make a distinction between only two types of questions:

- Questions that should be answered immediately.
- Questions that may be answered later on in the modelling process.

The first type of questions must appear on top of the list of questions, the rest follows.

Dialogue system

The modelling agenda uses a dialogue system, rather than a graphical user interface (GUI). Building a GUI for the modelling agenda requires an interface with another programming language like Java. This falls beyond the scope of this proof of concept, but in principle the dialogue could even be shaped around ”graphical propositions” using a diagram editor.
A dialogue system is easy to build in Prolog and still provides the interactivity needed to support the weak workflow of modelling. Prefix commands are used to distinguish between several types of input:

1. Add fact type
2. Add fact instance
3. Add object type
4. Add object instance
5. Answer question

When a user types following command: ”Add fact type : Student has Book.”, this tells the modelling agenda to add the fact type ”Student has Book” to the model. In a GUI, this would probably be replaced by a dropdown box with prefix commands and an open text box to input the model element.

Ojects in uppercase, roles in lowercase

Object types and object instances must start with an uppercase letter, roles must start with a lowercase letter. An example of correct input: ”Student has Book”. This choice was made, so that the user does not have the tedious task of telling the modelling agenda for each element if it is an object or a role and speeds up the modelling process significantly.

3.2.2 Requirements

1. The modelling agenda adds given ORM model elements to an ORM model.
2. The modelling agenda detects knowledge gaps in an ORM model, using a set of given business rules.
3. The modelling agenda creates a list of questions, addressing the knowledge gaps mentioned in requirement 2.
4. The modelling agenda adds elements, provided in answers to questions as mentioned in requirement 3, to an ORM model.
5. The modelling agenda adds temporal ordering to the questions mentioned in requirement 3.
6. The modelling agenda prevents double entries of fact types and object types.

3.3 Design cycle

This paragraph contains some remarks on the design work. Design started off with creating the ORM metamodel. From this metamodel, the syntactic rules for ORM were derived. In the last paragraphs, some design choices are described.

3.3.1 The ORM metamodel

In order to gather the declarative rules that determine whether or not an ORM model is syntactically correct, the below metamodel for ORM was drawn up. This initial metamodel accounts for the fact that any element of an ORM model could be added in any stage of the modelling process.

The metamodel for ORM is depicted in figure 3.

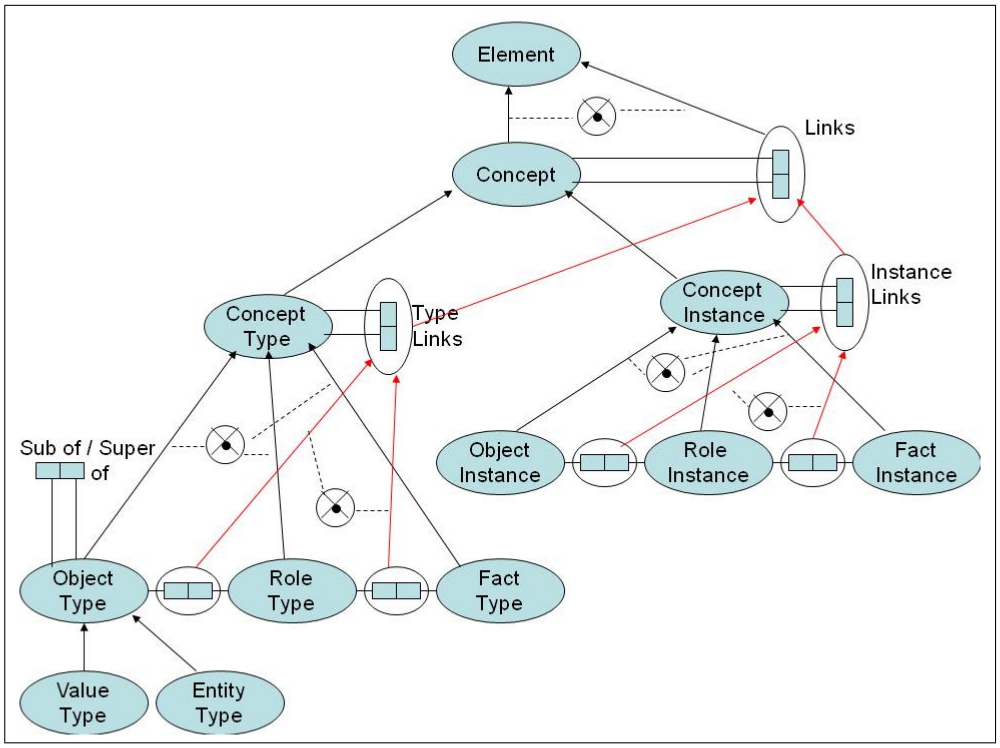


Figure 3: ORM Metamodel

Elements may be concepts, and the links between these concepts. Concepts and links can be devided into types and instances. Concepts are objects, roles, or facts. Object types are either entity types or value types. Object types can also be subtypes or supertypes of other object types. Instances have verbalisations (textual description, like 'Pete owns a car'). Note that constraints on fact types are left implicit in this metamodel.

During the construction phase, the scope was limited to only the basic elements of an ORM model (objects, roles, facts and uniqueness constraints). This reduced the complexity of the metamodel and led to the below, simplified ORM metamodel.

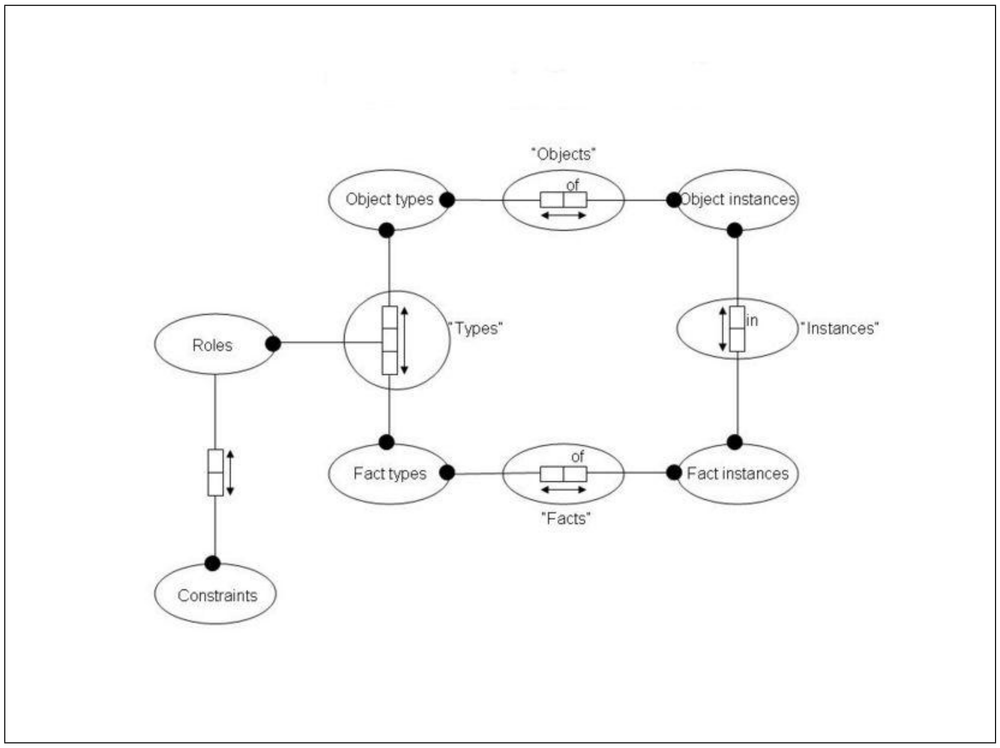


Figure 4: simplified ORM Metamodel

3.3.2 The syntactic metamodelling rules for ORM

Most of the decalarative rules that are needed to check an ORM model for inconsistencies can be easily retrieved from the above refined ORM metamodel. All mandatory role constraints define a single business rule for basic ORM (numers 1 - 12 in the next paragraph). There are three more business rules that cannot be captured from the metamodel; these stem from the fundamental priciples that ORM is based on (numbers 13, 14 and 15).

Business rules for basic ORM

1. Every fact type is populated by at least one fact instance
2. Every fact instance populates at least one fact type
3. Every object type is populated by at least one object instance
4. Every object instance populates at least one object type
5. Every object type participates in at least one fact type
6. Every fact type consists of at least one object type
7. Every role participates in at least one fact type
8. Every fact type consists of at least one role
9. Every object instance participates in at least one fact instance
10. Every fact instance consists of at least one object instance
11. Every role is constrained by at least one uniqueness constraint
12. Every uniqueness constraint belongs to at least one role
13. Every fact type is elementary
14. Every object type is unique (Principle of strong identification)
15. Every fact type is unique (Principle of strong identification)

3.3.3 Design choices

Prolog

After deriving the syntactical rules, a suitable programming language had to be chosen for implementation. Three candidate programming languages were evaluated: a logic programming language (like Prolog), a functional programming language (like Haskell) or an imperative programming language (like Java).

Because the modelling agenda has to be able to evaluate data on the basis of declarative rules, the choice for a Prolog was quickly made. The modelling agenda is basically a theorem prover. Its goal is to reach a state in which the model satisfies all declarative rules, expressed in logic formulas. A logic programming language best suits this purpose.

MySQL database

The initial idea was, to feed the modelling agenda with input files, run the ruleset and then output the agenda onto an output file. After several attempts, this proved to be very difficult. To use the knowledge, gathered from the input, vast arrays of information needed to be stored at runtime.

Therefore, a MySQL database was added to the modelling agenda. Although this requires a lot of interfacing (e.g. every single database result is returned as: ”row(*result*)” and needs to be stripped), it proved to be much easier than using input and output files.

Definite clause grammar

Business rules 1 - 12 (see paragraph 4.5.1) are used to derive the questions for the modelling agenda. These rules need to be parsed and interpreted. For this purpose, a small definite clause grammar (DCG) is used. The keyword ”every” translates as the logic quantifier $\forall$.

After parsing the logical statements, contradictions are sought in the database. $\forall$ is contradicted by an empty slot in the database. When an empty slot is found, a suitable question is added to the agenda.

Answering mechanism

The initial version of the modelling agenda supported only adding statements. This is a problem, because when the agenda contains a question ”Provide a fact type for fact instance ”John takes Mathematics”.”, and the user adds fact type ”Student takes Course”, the agenda can not automatically detect that the added fact type belongs to fact instance ”John takes Mathematics”.

Therefore, an answering module was added to the modelling agenda. A user can input a command, starting with ”Answer question X : ” (X being the number of the question). This combines the answer and the knowledge gap the question addresses.

4 The modelling agenda

This section contains a description of the functionality of the modelling agenda program. In the next paragraph, an example is provided based on a small model. The second paragraph contains a use case diagram of the modelling agenda; the last paragraph contains the use cases.

The use cases are slightly technical, not only providing insight in the interaction between user and program, but also in the interaction between program and database.

4.1 Functional description

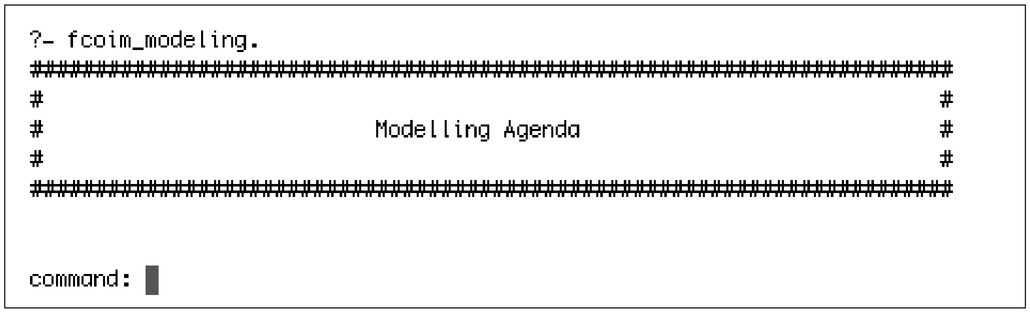
This section describes the modelling of a small domain containing students, teachers and courses.

4.1.1 Starting the modelling agenda

To start the modelling agenda, the user types ”fcoim_modeling”. This triggers a few actions:

1. An introduction text is shown to the user
2. The database is created, if it does not already exist
3. The repeat call is made, to start the command line loop
4. The user is prompted to input a command

Figure 5 shows what the screen looks like after this command.



```
?- fcoim_modeling.
##########################################################################
#                                                                        #
#                          Modelling Agenda                              #
#                                                                        #
##########################################################################


command: 
```

Figure 5: Starting up the modelling agenda

4.1.2 Adding a fact type

At the command prompt, the modeller types ”add fact type : Student takes Course”, and presses enter. A graphical representation of this fact type is shown in figure 6. The modelling agenda stores this input as a fact type and separates the object types from the role (these are stored separately in the database).

Then, the ruleset is applied to the model and some gaps are detected. For these gaps, questions are formulated and the agenda is shown to the user. Beneath the agenda appears the command prompt again. The resulting screen is shown in figure 7.

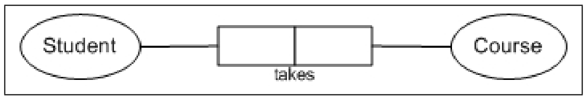
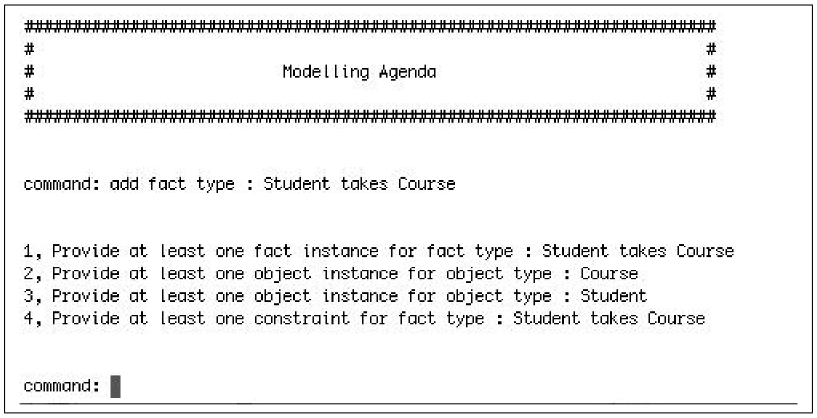


Figure 6: Model after adding a fact type



```
########################################################################
#                                                                      #
#                         Modelling Agenda                             #
#                                                                      #
########################################################################


command: add fact type : Student takes Course


1, Provide at least one fact instance for fact type : Student takes Course
2, Provide at least one object instance for object type : Course
3, Provide at least one object instance for object type : Student
4, Provide at least one constraint for fact type : Student takes Course


command:
```

Figure 7: Agenda after adding a fact type

4.1.3 Adding a fact instance

At the command prompt, the modeller types ”add fact instance : Jones teaches Mathematics”, and presses enter. The modelling agenda stores this input as a fact instance and separates the object instances from the role (these are stored separately in the database).

Then, the ruleset is applied to the model again and this results in more questions in the agenda. The resulting model is shown in figure 8; the resulting screen in figure 9.

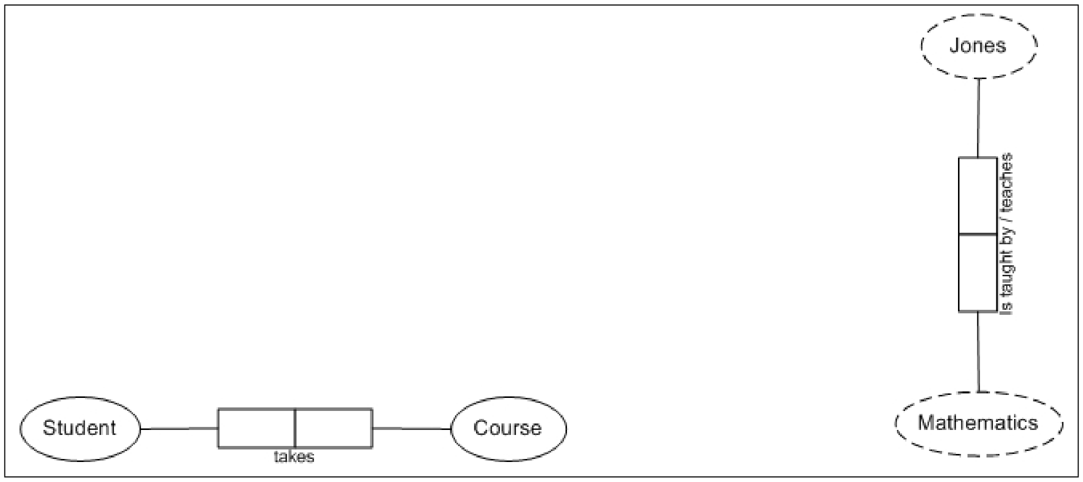
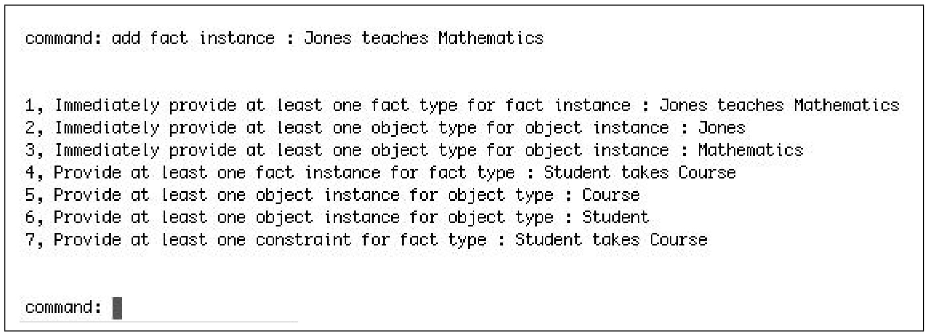


Figure 8: Model after adding a fact instance



```
command: add fact instance : Jones teaches Mathematics


1, Immediately provide at least one fact type for fact instance : Jones teaches Mathematics
2, Immediately provide at least one object type for object instance : Jones
3, Immediately provide at least one object type for object instance : Mathematics
4, Provide at least one fact instance for fact type : Student takes Course
5, Provide at least one object instance for object type : Course
6, Provide at least one object instance for object type : Student
7, Provide at least one constraint for fact type : Student takes Course


command: 
```

Figure 9: Agenda after adding a fact instance

4.1.4 Adding an object type

At the command prompt, the modeller types ”add object type : Teacher”, and presses enter. The modelling agenda stores this input as an object type.

This results in the following model (figure 10) and agenda (figure 11):

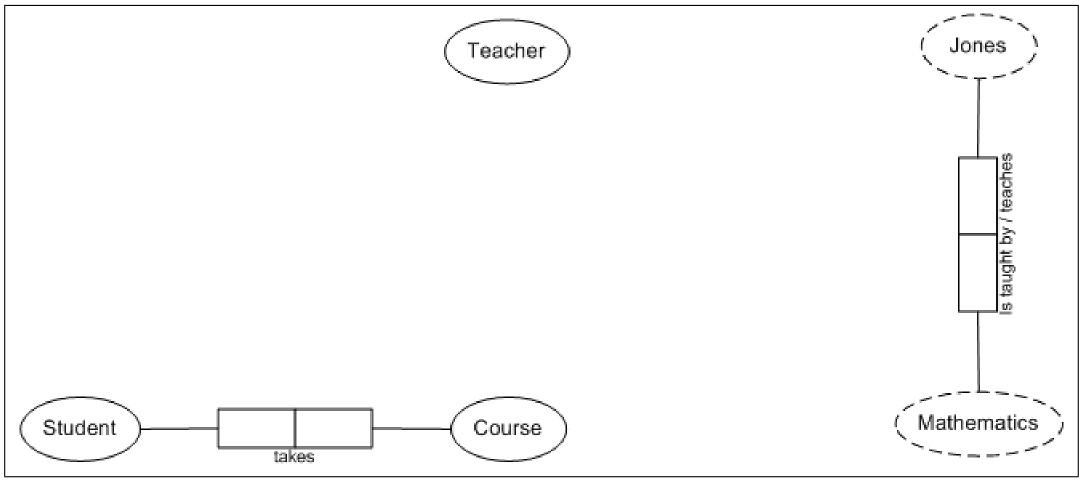
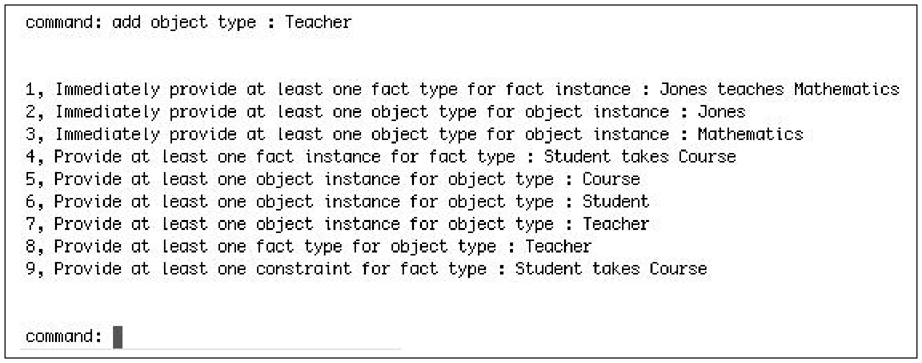


Figure 10: Model after adding an object type



```
command: add object type : Teacher


1, Immediately provide at least one fact type for fact instance : Jones teaches Mathematics
2, Immediately provide at least one object type for object instance : Jones
3, Immediately provide at least one object type for object instance : Mathematics
4, Provide at least one fact instance for fact type : Student takes Course
5, Provide at least one object instance for object type : Course
6, Provide at least one object instance for object type : Student
7, Provide at least one object instance for object type : Teacher
8, Provide at least one fact type for object type : Teacher
9, Provide at least one constraint for fact type : Student takes Course


command:
```

Figure 11: Agenda after adding an object type

4.1.5 Adding an object instance

At the command prompt, the modeller types "add object instance : Peter", and presses enter. The modelling agenda stores this input as an object instance.

The resulting model (figure 12) and agenda (figure 13) are shown below.

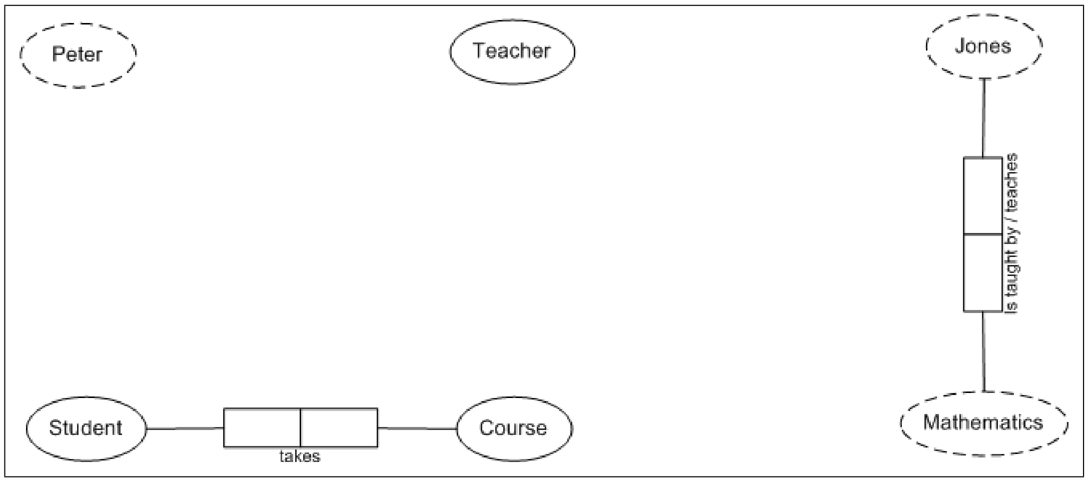
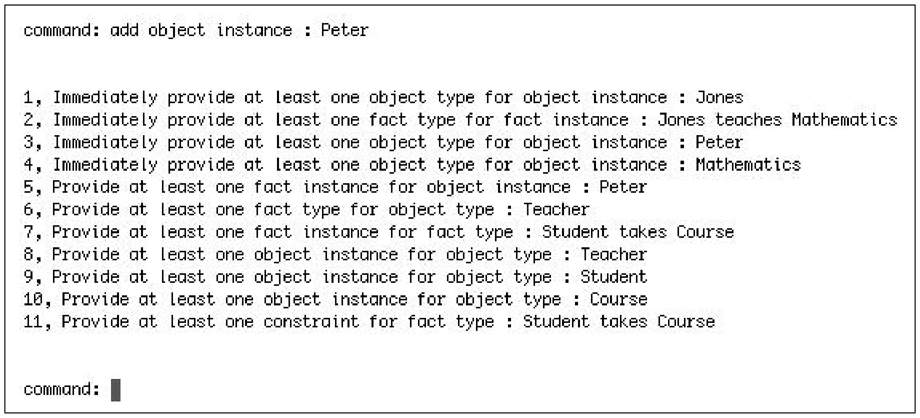


Figure 12: Model after adding an object instance



```
command: add object instance : Peter


1, Immediately provide at least one object type for object instance : Jones
2, Immediately provide at least one fact type for fact instance : Jones teaches Mathematics
3, Immediately provide at least one object type for object instance : Peter
4, Immediately provide at least one object type for object instance : Mathematics
5, Provide at least one fact instance for object instance : Peter
6, Provide at least one fact type for object type : Teacher
7, Provide at least one fact instance for fact type : Student takes Course
8, Provide at least one object instance for object type : Teacher
9, Provide at least one object instance for object type : Student
10, Provide at least one object instance for object type : Course
11, Provide at least one constraint for fact type : Student takes Course


command: 
```

Figure 13: Agenda after adding an object instance

4.1.6 Answering questions

Adding four elements to the model has resulted in eleven questions on the agenda. Four questions start with ″Immediately″, indicating that these have the highest priority. The modeller may however choose to answer any question he likes, he is not obliged to follow the order of the agenda.

The modeller answers the second question on the agenda (Immediately provide at least one fact type for fact instance : Jones teaches Mathematics). At the command prompt, he types ″answer 2 : Teacher teaches Course″, and presses enter. The modelling agenda stores this input as a fact type, belonging to fact instance ″Jones teaches Mathematics″.

The answer contains more information than just the fact type. In answering this question, four other questions are automatically answered:

- question 1 (Immediately provide at least one object type for object instance : Jones) is answered by ″Teacher″
- question 4 (Immediately provide at least one object type for object instance : Mathematics) is answered by ″Course″
- question 6 (Provide at least one fact type for object type : Teacher) is answered by ″Teacher teaches Course″
- question 10 (Provide at least one object instance for object type : Course) is answered by ″Mathematics″

These questions all disappear from the agenda. The resulting model (figure 14) and agenda (figure 15) are shown below.

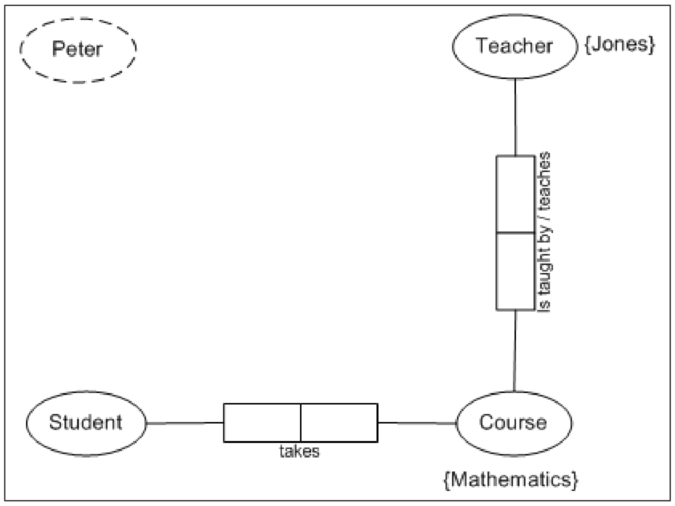
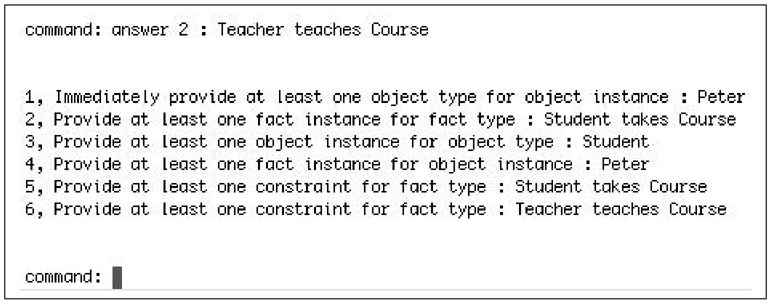


Figure 14: Model after answering a question



```
command: answer 2 : Teacher teaches Course


1, Immediately provide at least one object type for object instance : Peter
2, Provide at least one fact instance for fact type : Student takes Course
3, Provide at least one object instance for object type : Student
4, Provide at least one fact instance for object instance : Peter
5, Provide at least one constraint for fact type : Student takes Course
6, Provide at least one constraint for fact type : Teacher teaches Course


command: █
```

Figure 15: Agenda after answering a question

Next, the modeller answers the second question from the newly created agenda (Provide at least one fact instance for fact type : Student takes Course). He types ”answer 2 : Peter takes Mathematics”. This answers two more questions at the same time:

- question 1 (Immediately provide at least one object type for object instance : Peter) is answered by ”Student”
- question 3 (Provide at least one object instance for object type : Student) is answered by ”Peter”

These questions also disappear from the agenda. The resulting model (figure 16) and agenda (figure 17) are shown below.

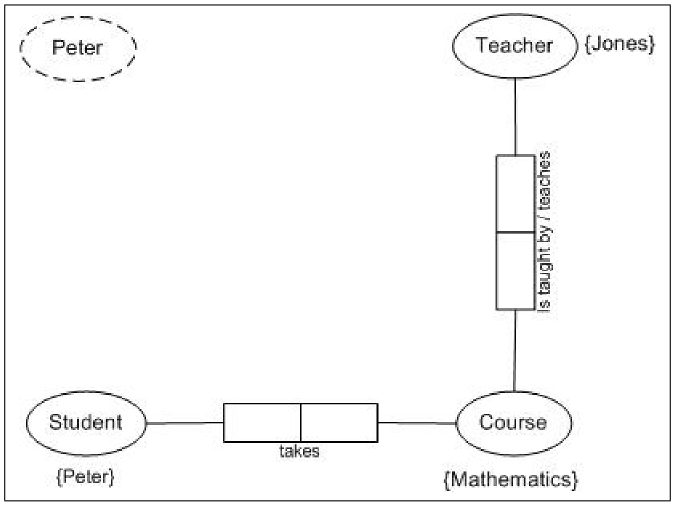
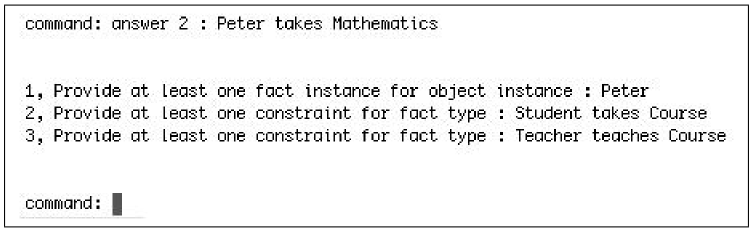


Figure 16: Model after answering the second question



```
command: answer 2 : Peter takes Mathematics


1, Provide at least one fact instance for object instance : Peter
2, Provide at least one constraint for fact type : Student takes Course
3, Provide at least one constraint for fact type : Teacher teaches Course


command: 
```

Figure 17: Agenda after answering the second question

The modeller now answers the question ”Provide at least one fact instance for object instance : Peter”. He types ”answer 1 : Peter takes Mathematics”. This answer one question, but it also adds three more questions to the agenda:

- Immediately provide at least one object type for object instance : Peter
- Immediately provide at least one object type for object instance : Mathematics
- Immediately provide at least one fact type for fact instance : Peter takes Mathematics

This may seem a bit strange, but it is a direct consequence of the fact that instances (object as well as facts) do not have to be unique in an ORM model. The modelling program may not assume that the lastly added fact instance ”Peter takes Mathematics” is the same as the one entered before. Every instance is treated as a new instance, so types have to be provided by the modeller.

The resulting model (figure 18) and agenda (figure 19) are given on the next page.

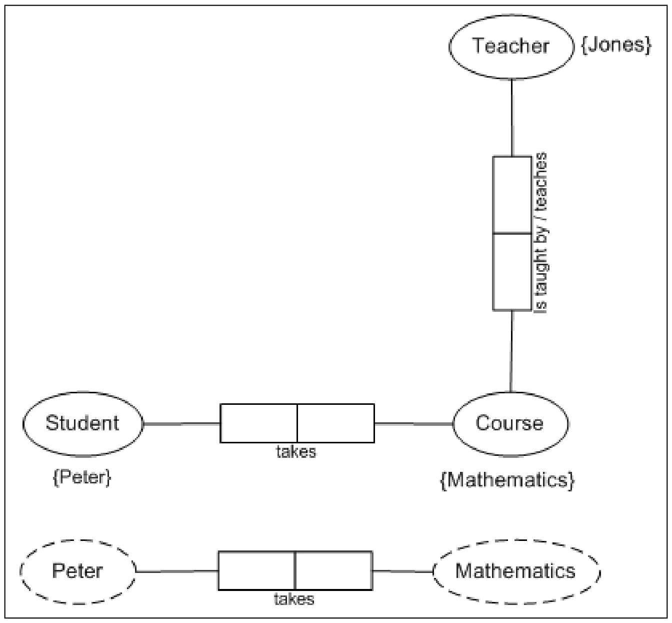
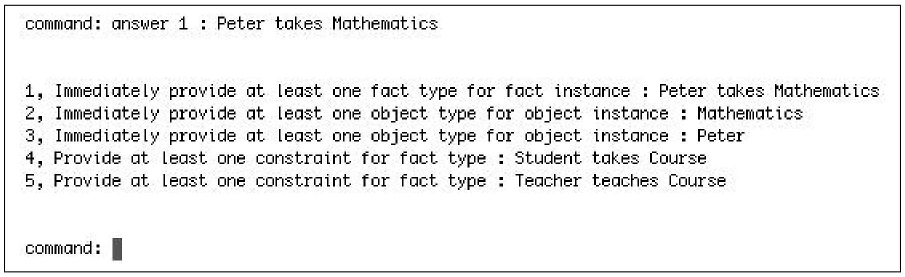


Figure 18: Model after answering the third question



```
command: answer 1 : Peter takes Mathematics


1, Immediately provide at least one fact type for fact instance : Peter takes Mathematics
2, Immediately provide at least one object type for object instance : Mathematics
3, Immediately provide at least one object type for object instance : Peter
4, Provide at least one constraint for fact type : Student takes Course
5, Provide at least one constraint for fact type : Teacher teaches Course


command:
```

Figure 19: Agenda after answering the third question

The modeller than answers the question ”Immediately provide at least one fact type for fact instance : Peter takes Mathematics” by typing ”answer 3 : Student takes Course”. This answers all three newly added questions. The resulting model (figure 20) and agenda (figure 21) are shown below.

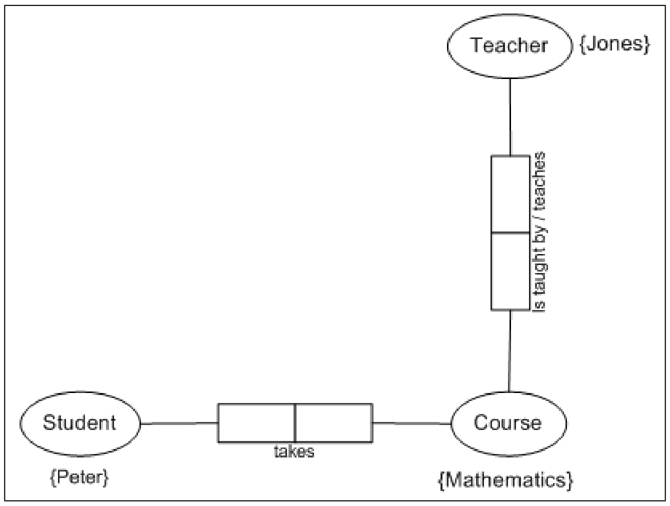
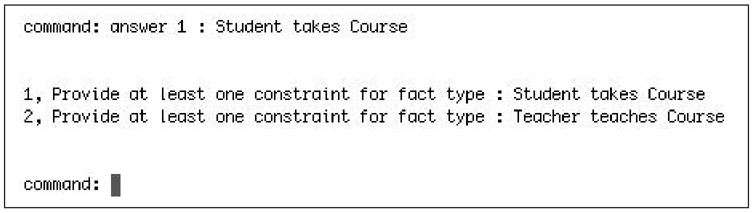


Figure 20: Model after answering the fourth question



```
command: answer 1 : Student takes Course


1, Provide at least one constraint for fact type : Student takes Course
2, Provide at least one constraint for fact type : Teacher teaches Course


command: 
```

Figure 21: Agenda after answering the fourth question

Lastly, the questions for the uniqueness constraints are answered. These uniqueness constraints are represented by a numeric value. In this example, the 4 stands for a uniqueness constraint spanning both roles of *takes*; the 1 stands for a constraint on the first role of *teaches*.

The finished model is shown below (figure 22), the agenda is empty as is shown in figure 23.

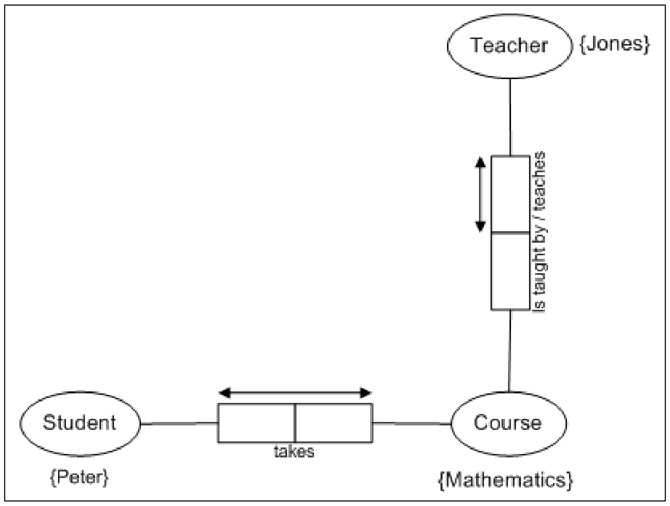
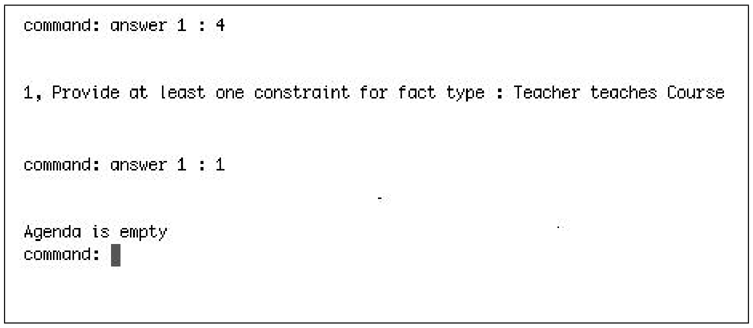


Figure 22: Model after answering the last questions



```
command: answer 1 : 4


1, Provide at least one constraint for fact type : Teacher teaches Course


command: answer 1 : 1


Agenda is empty
command: █
```

Figure 23: Agenda after answering the last question

4.2 Use case diagram

This section contains the use case diagram, followed by the use cases for the modelling agenda in the next section (4.3). The use cases are somewhat technical, not only describing interactions between user and modelling agenda, but also between modelling agenda and database.

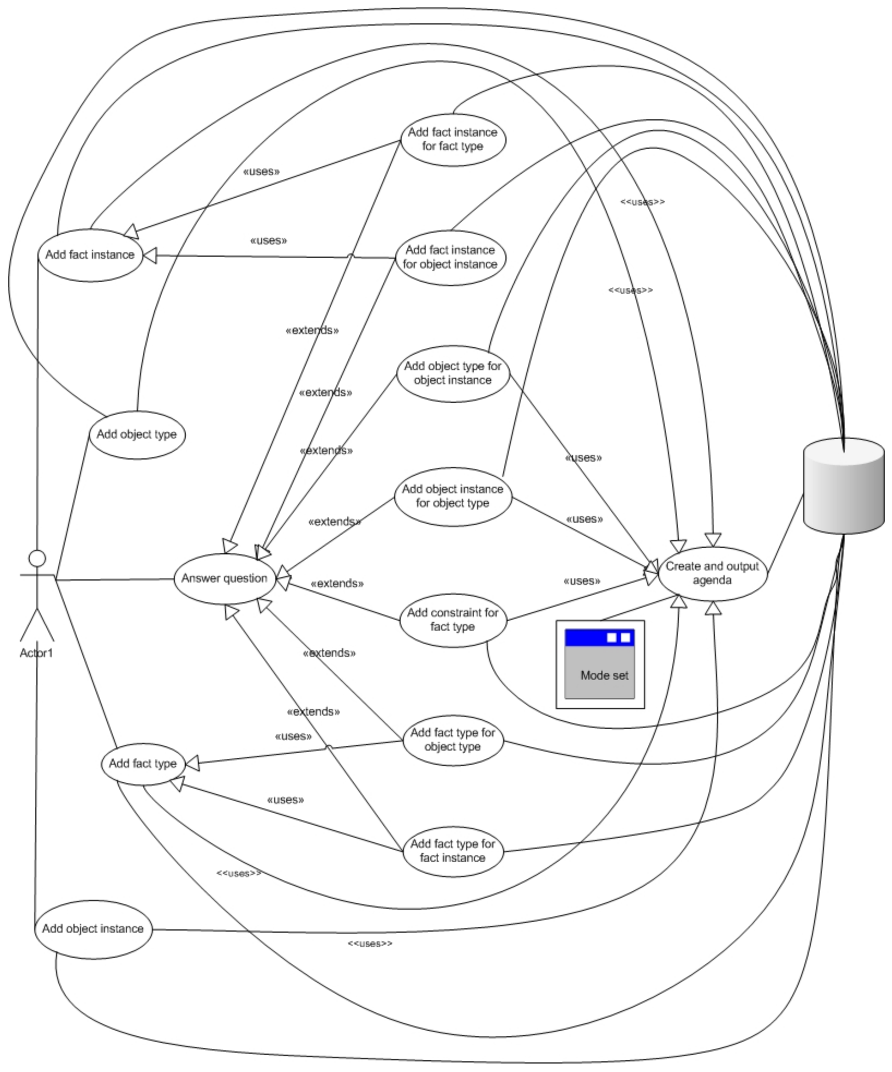

Add fact instance for fact type
Add fact instance
Add fact instance for object instance
Add object type for object instance
Add object type
Add object instance for object type
Answer question
Add constraint for fact type
Create and output agenda
Mode set
Actor1
Add fact type for object type
Add fact type
Add fact type for fact instance
Add object instance
«uses»
«extends»
<<uses>>

Figure 24: Use case diagram

4.3 Use cases

Use Case 1 - Add fact type

Use Case Name:	**Add fact type**
Authors:	Jörg Mutter
Date:	01-02-2008
Iteration:	Focused
Description:	User adds fact type to model.
Actors:	User and database.
Preconditions:	None.
Triggers:	The actor wants to add a fact type to the model.
Basic Course of Events:	1. User enters a command starting with ”add fact type : ” followed by a fact that is not yet in the database, 2. System adds an entry fact - object - role into table types in the database for each object type, 3. System creates separate tables for each object type in the database, 4. System creates a table for the fact type, 5. System enters the fact type and the role into table facts, 6. System proceeds with use case ”Create and output agenda”.

Alternative Path:	1. User enters a command starting with ”answer X”, where X is the number of a question in the agenda starting with ”Immediately provide fact type for object type” or ”Provide fact type for object type”, followed by a fact that is not yet in the database, 2. System adds an entry fact - object - role into table types in the database for each object type, 3. System creates separate tables for each object type in the database, 4. System creates a table for the fact type, 5. System enters the fact type and the role into table facts, 6. System proceeds with use case 13.
Alternative Path:	1. User enters a command starting with ”add fact type : ”, followed by a fact type that is already in the database, 2. System proceeds with use case 13.
Alternative Path:	1. User enters a command starting with ”Answer X”, where X is the number of a question in the agenda starting with ”Immediately provide fact type for object type” or ”Provide fact type for object type”, followed by a fact type that is already in the database, 2. System proceeds with use case 13.
Exception Paths:	None.
Assumptions:	None.
Postconditions:	Tables for objects and fact and entries in several tables.
Related business rules:	All fact types are unique

Use Case 2 - Add fact instance

Use Case Name:	**Add fact instance**
Authors:	Jörg Mutter
Dates:	01-07-2007
Iteration:	Focused
Description:	User adds fact instance to model.
Actors:	User and database.
Preconditions:	None.
Triggers:	The actor wants to add a fact instance to the model.
Basic Course of Events:	1. User enters a command starting with ”add fact instance : ” followed by a fact instance, 2. System adds an entry ”fact - object” into table instances in the database for each object 3. System adds fact instance into table facts, 4. System proceeds with use case ”Create and output agenda”.
Alternative Path:	1. User enters a command starting with ”Answer X”, where X is the number of a question in the agenda starting with ”Immediately provide fact instance for object instance” or ”Provide fact instance for object instance”, followed by a fact instance, 2. System adds an entry fact - object into table instances in the database for each object instance, 3. System enters the fact instance and the role into table facts, 4. System proceeds with use case 13.

Exception Paths:	None.
Assumptions:	None.
Postconditions:	Entries in tables instances and facts.
Related business rules:	

Use Case 3 - Add object type

Use Case Name:	**Add object type**
Authors:	Jörg Mutter
Dates:	01-02-2008
Iteration:	Focused
Description:	User adds object type to model.
Actors:	User and database.
Preconditions:	None.
Triggers:	The actor wants to add an object type to the model.
Basic Course of Events:	1. User enters a command starting with ”add object type : ” followed by an object type that is not yet in the database. 2. System creates a seperate table for the object type, 3. System enters the object type into table objects, 4. System inserts object type into table types, 5. System proceeds with use case ”Create and output agenda”.
Alternative Path:	1. User enters a command starting with ”add object type : ” followed by an object type that is already in the database. 2. System proceeds with use case 13.
Exception Paths:	None.
Assumptions:	None.
Postconditions:	Entries in tables objects and types.
Related business rules:	All object types are unique.

Use Case 4 - Add object instance

Use Case Name:	**Add object instance**
Authors:	Jörg Mutter
Dates:	01-02-2008
Iteration:	Focused
Description:	User adds object instance to model.
Actors:	User and database.
Preconditions:	None.
Triggers:	The actor wants to add an object instance to the model.
Basic Course of Events:	1. User enters a command starting with ”add object instance : ” followed by an object instance, 2. System inserts the object instance into table objects, 3. System inserts the object instance into table instances, 4. System proceeds with use case ”Create and output agenda”.
Alternative Path:	None.
Exception Paths:	None.
Assumptions:	None.
Postconditions:	Entries in tables objects and instances.
Related business rules:	

Use Case 5 - Answer question

Use Case Name:	**Answer question**
Authors:	Jörg Mutter
Dates:	01-02-2008
Iteration:	Focused
Description:	User answers a question in modelling agenda.
Actors:	User.
Preconditions:	There is at least one question in the agenda.
Triggers:	The actor wants to answer a question in the modelling agenda.
Basic Course of Events:	1. User enters an answer to a question in the agenda, asking for an object type for an object instance, 2. System proceeds with use case 6.
Alternative Path:	1. User enters an answer to a question in the agenda, asking for an object instance for an object type, 2. System proceeds with use case 7.
Alternative Path:	1. User enters an answer to a question in the agenda, asking for a fact type for a fact instance, 2. System proceeds with use case 8.
Alternative Path:	1. User enters an answer to a question in the agenda, asking for a fact instance for a fact type, 2. System proceeds with use case 9.
Alternative Path:	1. User enters an answer to a question in the agenda, asking for a fact type for an object type, 2. System proceeds with use case 10.
Alternative Path:	1. User enters an answer to a question in the agenda, asking for a fact instance for an object instance, 2. System proceeds with use case 11.

Alternative Path:	1. User enters an answer to a question in the agenda, asking for a constraint for a fact type, 2. System proceeds with use case 12.
Exception Paths:	None.
Assumptions:	None.
Postconditions:	None.
Related business rules:	

Use Case 6 - Add object type for object instance

Use Case Name:	**Add object type for object instance**
Authors:	Jörg Mutter
Dates:	01-07-2007
Iteration:	Focused
Description:	System adds object type for object instance to model
Actors:	Database.
Preconditions:	At least one untyped object instance exists in the model.
Triggers:	Use case ”Answer question”.
Basic Course of Events:	1. System inserts the object type into table objects, 2. System inserts the object type into table types, 3. System creates a table for the object type, 4. System inserts the object instance into the table created in step 4, 5. System proceeds with use case ”Create and output agenda”.
Alternative Paths:	• In step 2, if entry object type - object instance already exists, proceed with step 3 • In step 3, if entered object type already exists, proceed with step 4 • In step 4, if table already exists, proceed with step 5 • In step 5, if the object instance already occurs in the table, proceed with step 6
Exception Paths:	None.
Assumptions:	None.
Postconditions:	1. Entries in tables objects and types 2. A table for the new object type with an entry for the object instance.
Related business rules:	None.

Use Case 7 - Add object instance for object type

Use Case Name:	**Add object instance for object type**
Authors:	Jörg Mutter
Dates:	01-02-2008
Iteration:	Focused
Description:	System adds object instance for object type.
Actors:	Database.
Preconditions:	At least one uninstantiated object type exists in the model.
Triggers:	Use case ”Answer question”.
Basic Course of Events:	1. System inserts the object instance into table objects on the row where the object type occurs, 2. System inserts the object instance into table instances, 3. System inserts the object instance into the table for the object type, 4. System proceeds with use case ”Create and output agenda”.
Alternative Paths:	• In step 2, if entry object type - object instance already exists, proceed with step 3 • In step 5, if the object instance already occurs in the table, proceed with step 6
Exception Paths:	None.
Assumptions:	None.
Postconditions:	1. Entries in tables objects and instances 2. Entry in the table for the object type.
Related business rules:	None.

Use Case 8 - Add fact type for fact instance

Use Case Name:	**Add fact type for fact instance**
Authors:	Jörg Mutter
Dates:	01-02-2008
Iteration:	Focused
Description:	System adds fact type for fact instance.
Actors:	Database.
Preconditions:	At least one untyped fact instance occurs in the model.
Triggers:	Use case ”Answer question”.
Basic Course of Events:	1. System inserts the fact type Into table facts on the row where the untyped fact instance occurs, 2. System performs steps 2, 3, 4 and 5 from use case ”Add fact type”, 3. System inserts the fact instance into the newly created table for the fact type, 4. System inserts the object instances occurring in the fact instance into the newly created tables for their counterpart object types, 5. System inserts the object types into table objects on the rows where their counterpart object instances occur, 6. System proceeds with use case ”Create and output agenda”.
Alternative Path:	• In step 2, if the combination fact type - fact instance already exists, delete the single fact type and proceed with Use Case 15 • If in step 5 the object type and object instance already occur apart from each other, delete these rows and insert one with the combination object type - object instance
Exception Paths:	None.
Assumptions:	None.

Postconditions:	• Entries in tables facts, objects, types and instances • Entries in the tables for the fact type and its object types
Related business rules:	

Use Case 9 - Add fact instance for fact type

Use Case Name:	**Add fact instance for fact type**
Authors:	Jörg Mutter
Dates:	01-02-2008
Iteration:	Focused
Description:	System adds fact instance for fact type.
Actors:	Database.
Preconditions:	At least one uninstantiated fact type exists in the model.
Triggers:	Use case ”Answer question”.
Basic Course of Events:	1. System inserts the fact instance into table facts on the row where the uninstantiated fact type occurs, 2. System inserts the fact instance into the table for the fact type, 3. System retrieves the object instances occuring in the fact instance 4. System inserts the object instances found in step 3 into table objects on the rows where their counterpart object types occur, 5. System inserts the object instances into the tables for the object types, 6. System proceeds with use case ”Add fact instance” for this fact instance

Alternative Path:	• In step 2, if the combination fact type - fact instance already exists, delete the single fact type and proceed with Use Case 15 • If in step 5 the object type and object instance already occur apart from each other, delete these rows and insert one with the combination object type - object instance
Exception Paths:	None.
Assumptions:	None.
Postconditions:	• Entries in tables facts, objects, types and instances • Entries in the tables for the fact type and its object types
Related business rules:	

Use Case 10 - Add fact type for object type

Use Case Name:	**Add fact type for object type**
Authors:	Jörg Mutter
Dates:	01-02-2008
Iteration:	Focused
Description:	System adds fact type for object type.
Actors:	Database.
Preconditions:	At least one object type occurs in the model without a fact type.
Triggers:	Use case ”Answer question.
Basic Course of Events:	1. System checks if combination fact type - object type already exists in table types in the database, 2. If this combination does not yet exist, the system deletes the row in table types in the database where the object type occurs alone, 3. System proceeds with use case ”Add fact type” for this fact type.
Alternative Path:	In step 2, if the combination fact type - object type already exists, the system proceeds with step 3.
Exception Paths:	None.
Assumptions:	None.
Postconditions:	• Entries in tables facts, objects and types • New tables for the newly introduced object type(s)
Related business rules:	

Use Case 11 - Add fact instance for object instance

Use Case Name:	**Add fact instance for object instance**
Authors:	Jörg Mutter
Dates:	01-02-2008
Iteration:	Focused
Description:	System adds fact instance for object instance.
Actors:	Database.
Preconditions:	At least one object instance occurs in the model without a fact instance
Triggers:	Use case ”Answer question”.
Basic Course of Events:	1. System checks if combination fact instance - object instance already exists in table instances in the database, 2. If this combination does not yet exist, the system deletes the row in table instances in the database where the object instance occurs alone, 3. System proceeds with use case ”Add fact instance” for this fact instance.
Alternative Path:	In step 2, if the combination fact instance - object instance already exists, the system proceeds with step 3.
Exception Paths:	None.
Assumptions:	None.
Postconditions:	• Entries in tables facts, objects and instances in the database • New tables for the newly introduced object instance(s)
Related business rules:	

Use Case 12 - Add constraint for fact type

Use Case Name:	**Add constraint for fact type**
Authors:	Jörg Mutter
Dates:	01-02-2008
Iteration:	Focused
Description:	System adds constraint for fact type.
Actors:	Database.
Preconditions:	At least one unconstrained fact exists in the model.
Triggers:	Use case ”Answer question”.
Basic Course of Events:	1. System inserts the constraint on every row where the fact type occurs in table types, 2. System proceeds with use case ”Create and output agenda”.
Alternative Path:	None.
Exception Paths:	None.
Assumptions:	None.
Postconditions:	Entries in table types.
Related business rules:	

Use Case 13 - Create and output agenda

Use Case Name:	**Create and output agenda**
Authors:	Jörg Mutter
Dates:	01-02-2008
Iteration:	Focused
Description:	System creates and outputs modelling agenda.
Actors:	Database, mode set
Preconditions:	A modelling mode has been set by the user.
Triggers:	Use cases 2, 3, 4, 6, 7, 8 and 12
Basic Course of Events:	1. System scans the database for entries that do not satisfy business rules 1 to 16 in the business rules catalogue, 2. System checks the priority for every violation of a business rule, as described by the user in mode set, 3. System creates a suitable question for every violated business rule (according to the information that has to be added to the model and the priority of the violation), 4. System sorts the questions in the agenda, according to their priority, 5. System shows the agenda on the screen.
Alternative Path:	None.
Exception Paths:	None.
Assumptions:	None.
Postconditions:	A (non-empty) agenda.
Related business rules:	

5 Conclusions and future work

This chapter contains the answers to the research questions in chapter 1. Subsequently, some remarks on future work and research on the modelling agenda are made.

5.1 Questions and answers

How can a modelling agenda be generated based on syntactical rules, procedural rules and a partial model?

What is a modelling agenda?
A modelling agenda is a support tool for a (team of) modeller(s). The modelling agenda in essence is a collection of ”toDo items” for a partial model, based on syntactical and procedural rules for modelling. Adding new elements to the partial model may lead to questions on the modelling agenda. These questions may be answered, leading to new model elements and possibly new items on the agenda.

In a broader perspective, the modelling agenda can be used as (part of) a modelling forum, in which modellers propose adding or changing model elements. These propositions may then be challenged by other modellers, or domain experts, leading to altered partial models. This modelling forum might help close the gap between business and IT.

What are procedural rules of modelling?
Procedural rules of modelling are concrete descriptions of the actions to be performed in a modelling session. These procedural rules tell the modeller(s) which actions have to be taken in order for the model to be sound and complete. These procedural decriptions may be denoted textually (in natural or controlled language or even pseudo code) or graphically (for example a workflow diagram).

Provide a proof of concept for a modelling agenda.
The proof of concept has been demonstrated in chapter 4.1. The source code for the modelling agenda can be found in the appendix.

5.2 Future work and research

Modelling forum
In its current form, the modelling agenda is a pre-emptive model checker, that transforms knowledge gaps into relevant questions. Other tools exist, in which modellers debate the model at hand, posing new questions themselves or challenging earlier decisions or propositions. Integrating the modelling agenda with such a tool might shed some more light on modelling procedures.

Extending the logic
The modelling agenda may be extended with logic for business rules 13, 14, 15 (see paragraph 4.5.1). Also, logic that checks the set of business rules for inconsistencies is needed. In its current form, inconsistent business rules may lead to an infinite loop within the modelling agenda.

Extending the modelling language
The modelling agenda now only supports the key ingredients for a syntactically sound and complete ORM model. This however is only a tiny portion of what ORM offers. Thereby, other languages, like UML, are more widely used and might be supported.

Usability
In order to support natural language better, a spelling checker to correct user input (much like Google does in its search engine) and a graphical user interface (GUI) would be nice additions.

6 Literature

Adsett04 : Adsett, C., Bernardi, A, Liu, S, and Spencer, B. (2004). *Realising Weak Work Workflow with Declarative Flexible XML Routing in SOAP (DeFleX)* (Conference publication). Conference: Proceedings of Business Agents and Semantic Web (BASeWEB'04); a workshop in conjunction with the Seventeenth Canadian Conference on Artificial Intelligence, London, Ontario, Canada, May 16, 2004.

Bommel08 : van Bommel, P., Hoppenbrouwers, S.J.B.A., Proper, H.A., Roelofs, J. (2008). *Concepts and Strategies for Quality of Modeling*, Innovations in information systems modeling: methods and best practices, Advances in database research (ADR) series, Halpin, T., Krogstie, J., Proper, H.A. (editors), pp. 167-189.

Bryant05 : Bryant, D. (2005). *Exploring agent-based argumentation dialogue games*, MSc Dissertation, University of Surrey.

Carlson83 : Carlson, L. (2003). *Dialogue games: an approach to discourse analysis*, Dialogue Games: An Approach to Discourse Analysis, Reidel Publishing Company, Dordrecht, The Netherlands.

Clark04 : Clark, T., Evans, A., Sammut, P., Willans, J. (2004). Applied Metamodelling - A Foundation for Language Driven Development, Xactium, Inc., Sept. 2004.

Elst03 : van Elst, L., Aschoff, F. R., Bernardi, A., Schwarz, S. (2003). *Weakly-structured Workflows for Knowledge-intensive Tasks: An Experimental Evaluation*, WETICE 03: Proceedings of the Twelfth International Workshop on Enabling Technologies, pp.340 345.

Halpin98 : Halpin, T. (1998). *Object-Role Modeling (ORM/NIAM)*, Handbook on Architectures of Information Systems, Chapter 4.

Hevner07 : Hevner, A. (2007). *A Three Cycle View of Design Science Research*, Scandinavian Journal of Information Systems, Vol. 19, No. 2, 2007, pp. 87-92.

Hoppenbrouwers06 : Hoppenbrouwers, S.J.B.A., Proper, H.A., van der Weide, Th.P. (2006). *Exploring Modelling Strategies in a Meta-modelling Context*, OTM Workshops 2006, LNCS 4278, R. Meersman, Z. Tari, P. Herrero et al. (editors), Springer-Verlag Berlin Heidelberg, pp. 11281137.

Hoppenbrouwers08 : Hoppenbrouwers, S,J,B.A., van Bommel, P., Jarvinen, P. (2008). *Method engineering as game design - an emerging HCI perspective on methods and case tools*, Proceedings the Thirteenth International Workshop on Exploring Modeling Methods in Systems Analysis and Design (EMMSAD 2008), held in conjunction with CAISE 2008, volume 337 of CEUR Workshop Proceedings, Halpin, T., Proper, H.A., Krogstie, J. (editors), pp 167-189.

Iivari07 : Iivari, J. (2007). *A Paradigmatic Analysis of Information Systems as a Design Science*, Scandinavian Journal of Information Systems, 19(2), pp. 39-64.

Lebbink03 : Lebbink, H.J., Witteman, C.L.M., Meyer, J.J.C. (2003). *Dialogue games for inconsistent and biased information*, LCMAS 2003, Logic and Communication in Multi-Agent Systems, van der Hoek, W., Lomuscio, A., de Vink, E. (editors), pp. 134-151.

Lindeman06 : Lindeman, L. (2006). Modelleerprocessen. MSc Dissertation, Radboud University Nijmegen; in Dutch.

Prakken05 : Prakken, H. (2005). *Coherence and flexibility in dialogue games for argumentation*, Journal Logic and Computation 15(6), pp. 1009-1040.

Ravenscroft06 : Ravenscroft, A., McAlister, S. (2006). Digital Games and Learning in Cyberspace: A Dialogical Approach, E-Learning Journal, Vol. 3, No 1, pp 38-51

Ross03 : Ross, R., (editor) (2003). Business Rules Manifesto. Business Rules Group, version 2.0., http://www.businessrulesgroup.org/brmanifesto.htm (accessed June 12, 2010)

Ssebuggwawo09 : Ssebuggwawo, D., Hoppenbrouwers, S.J.B.A., Proper, H.A. (2009). *Interactions, Goals and Rules in a Collaborative Modelling Session*, Second IFIP WG 8.1 Working Conference on The Practice of Enterprise Modeling: from Business Strategies to Enterprise Architectures Stockholm, Sweden, November, 2009, Persson, A., Stirna, J. (editors), Springer, Berlin, Germany, pp 54-68

Zanten04 : Veldhuijzen van Zanten, G., Hoppenbrouwers, S.J.B.A., Proper, H.A. (2004). System Development as a Rational Communicative Process. Journal of Systemics, Cybernetics and Informatics, vol. 2(4), pp. 47-51

7 Appendix

7.1 Module overview

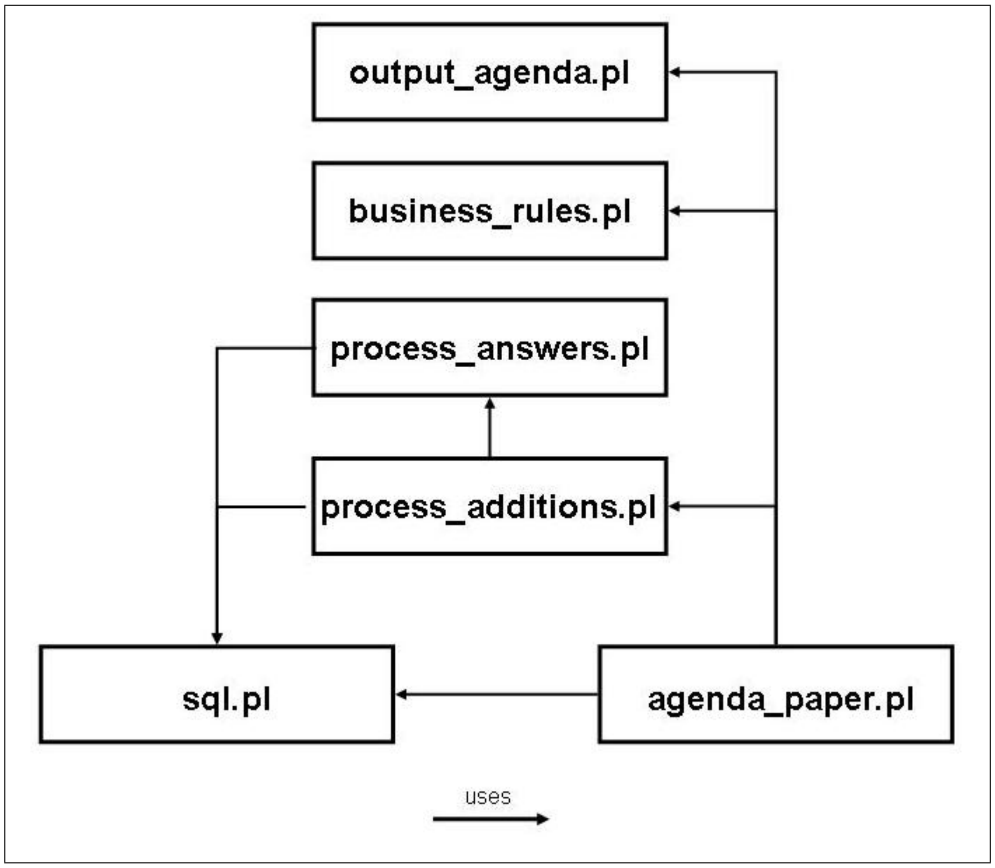


Figure 25: Module overview

The modelling agenda consists of six prolog modules. The main module is agenda_paper.pl. This module contains the main loop which enables the modeller to continuously give new commands. Agenda_paper uses four modules, the first being sql.pl, which contains general functions to connect prolog to a mySQL database.

When the modeller gives a new command, module process_additions is called. When the command contains an answer to an agenda question, this module calls module process_answer. Otherwise the addition is processed in module process_additions itself. Both modules use general functions from sql.pl.

After processing the command, module agenda_paper calls module business_rules, which checks the database for rule violations. After deleting the old agenda questions, every rule violation is transformed into a new agenda question.

Finally, agenda_paper calls module output_agenda, which constructs a neatly numbered new agenda from the questions generated by module business_rules.

7.2 Module agenda_paper.pl

```
:- include('sql.pl').
:- include('process_additions.pl').
:- include('process_answers.pl').
:- include('business_rules.pl').
:- include('output_agenda.pl').

:- dynamic model_name/1.
:- dynamic mode/1.
```

The main loop assures commands can be given continuously Everything after repeat is repeated as long as command 'stop' is not given In case of command 'stop', end_statement ends the program

```
fcoim_modeling :-
    intro,
    connect_db(test, root,''),
    create_tables,
    assert(mode(fcoim)),
    repeat,
    nl,
    write('command: '),
    text_to_list(C),
    execute(C),
    end_statement(C),
    !.
```

Execute tries to execute the command In case of an invalid command, this is communicated In case of command 'stop', end_statement is triggered

```
execute(C) :-
    split_at(C, Lists, ':'),
    process_list(Lists),
    fill_agendas,
    output_agenda,
    !;
    C == [show, agenda],
    output_agenda,
    !;
    C \= [stop],
    write('This is not a valid command!'),
    ! ;
    !.
```

The introduction statement

```
intro :-
    write('#################################################################'),nl,
    write('#'),
    write('                                                              '),
    write('#'),
    nl,
    write('#'),
    write('Modelling Agenda'),
    write('#'),
    nl,
    write('#'),
    write('                                                              '),
    write('#'),
    nl,
    write('#################################################################'),nl,nl.
```

text_to_list reads input from the command line and converts it into list format

```
text_to_list(L) :-
    readline(T),
    wordlist(L,T,[]).

  readline(L) :- get0(Char),buildlist(Char,L).

buildlist(10,[]) :- !.
buildlist(Char,[Char |X]) :- get0(Char2),buildlist(Char2,X).

wordlist([X |Y]) -->word(X),whitespace,wordlist(Y).
wordlist([X]) -->whitespace,wordlist(X).
wordlist([X]) -->word(X).
wordlist([X]) -->word(X),whitespace.

word(W) -->charlist(X),name(W,X).

charlist([X |Y]) -->chr(X),charlist(Y).
charlist([X]) -->chr(X).
chr(X) -->[X], X>=33.

whitespace -->whsp,whitespace.
whitespace -->whsp.
whsp -->[X],X<33.
```

End statement

```
end_statement([stop]) :-
    nl,
    write('Program terminated'),
    retract(mode(_)),
    stop.

  stop.
```

7.3 Module process_additions.pl

Function split_map is used to split multiple answers into separate lists

```
split_map([],_,_).
split_map([X],Y,_) :-
    Y = ([[X]]).
split_map([X |Xs], Y, Z) :-
    split_at([X |Xs], [Voor |Na], Z),
    flatten(Na,Nas),
    split_map(Nas, Ys, Z),
    Y = [Voor |Ys].
```

Function split_at is used to split a list into two separate lists at symbol 'Separator'

```
split_at([], _, _) :- !.
split_at(Input, Output, Separator) :-
    conc(Head, [Separator |Tail], Input),
    Output = [Head, Tail],
    !;
    Output = Input,
    !.
```

Function conc concatenates lists

```
conc([],L,L).
conc([X |L1],L2,[X |L3]) :-
    conc(L1,L2,L3).
```

Function list_to_string converts a list of words into a string

```
list_to_string(List, String) :-
    list_to_string(List, [], String), !.

list_to_string([], L, String) :-
    atom_chars(String, L), !.

list_to_string([H], L, String) :-
    atom_chars(H, Hlist),
    append(L, Hlist, List),
    list_to_string([], List, String), !.

list_to_string([H |T], L, String) :-
    atom_chars(H, Hlist),
    append(Hlist, [' '], HH),
    append(L, HH, List),
    list_to_string(T, List, String), !.
```

Function capitalized checks if a word starts with a capital

```
capitalized(X) :-
    string_to_list(X, Y),
    Y = [C0 |_],
    C0 =<96.
```

Function process_list processes lists of lists in case of file use

```
process_list(X,[]) :-
    process(X,[]).
process_list([X,Y]) :-
    process(X, Y), !.
process_list([ [X,Y] |Rest ]) :-
    process(X, Y),
    process_list(Rest),
    !.
```

Function process receives two lists: one with the function to be called, the other with the fact or object types or instances

```
process(X, Y) :-
    X == [add, fact, type],
    add_fact_type(Y, _, Y);
    X == [add, fact, instance],
    add_fact_instance(Y, Y);
    X == [add, object, type],
    add_object_type(Y);
    X == [add, object, instance],
    add_object_instance(Y);
    X = [answer, Z],
    member(',',Y),
    split_map(Y, [A |As], ','),
    prepare_answer(Z, A),
    process(X,As);
    X = [answer, Z],
    prepare_answer(Z, Y);
    fail.
```

Function add_fact_type receives the fact type
It selects the object types and sends these to function add_object_type, along with the fact type itself.

```
add_fact_type([], Role, Y) :-
    add_role_type(Role),
    list_to_string(Y,S),
    (
    check_existence(S, facts, facttype);
    insert_X(S, facts, facttype)
    ).

add_fact_type([X], Role, Y) :-
    capitalized(X)
    add_object_type(X,Y),
    conc(Role,['...'],Rest),
    add_role_type(Rest),
    list_to_string(Y,S),
    (
    check_existence(S, facts, facttype);
    insert_X(S, facts, facttype)
    );
    conc(Role,[X],Rest),
    add_role_type(Rest),
    list_to_string(Y,S),
    (
    check_existence(S, facts, facttype);
    insert_X(S, facts, facttype)
    ).

add_fact_type([X |Xs], Role, Y) :-
    capitalized(X),
    add_object_type(X,Y),
    conc(Role,['...'],Rest),
    add_fact_type(Xs, Rest, Y);
    conc(Role,[X],Rest),
    add_fact_type(Xs, Rest, Y).
```

Function add_fact_instance receives the fact instance
It selects the object instances and sends these to function
add_object_instance, along with the fact instance itself.

```
add_fact_instance([], Y) :-
    list_to_string(Y,S),
    (
    check_existence(S, facts, factinstance);
    insert_X(S, facts, factinstance)
    ).

    add_fact_instance([X], Y) :-
    capitalized(X),
    add_object_instance(X,Y),
    list_to_string(Y,S),
    (
    check_existence(S, facts, factinstance);
    insert_X(S, facts, factinstance)
    );
    list_to_string(Y,S),
    (
    check_existence(S, facts, factinstance);
    insert_X(S, facts, factinstance)
    ).

add_fact_instance([X |Xs], Y) :-
    capitalized(X),
    add_object_instance(X,Y),
    add_fact_instance(Xs, Y);
    add_fact_instance(Xs, Y).
```

Functon add_object_type(1) receives an object type and places it in tables objects and types

```
add_object_type([X]) :-
    create_table(X),
    check_existence(X, objects, objecttype);
    insert_X(X, objects, objecttype),
    insert_object_into_types(X).
```

Functon add_object_type(2) receives an object type and a fact type.
It places the objects into the objects and types tables.
The fact is placed in tables facts and types.

```
add_object_type(X, Y) :-
    create_table(X),
    list_to_string(Y,S),
    insert_fact_into_types(S,X),
    (
    check_existence(X, objects, objecttype);
    insert_X(X, objects, objecttype)
    ),
    create_table(S).
```

Functon add_object_instance(1) receives an object instance and places it in tables objects and instances

```
add_object_instance([X]) :-
```

```
    insert_X(X, objects, objectinstance),
    insert_object_into_instances(X).
```

Function add_object_instance(2) receives an object instance and a fact instance. It places the objects into the objects and instances tables. The fact is placed in tables facts and instances

```
add_object_instance(X,Y) :-
    list_to_string(Y,S),
    insert_fact_into_instances(S,X),
    insert_X(S, facts, factinstance),
    insert_object_and_instance_into_objects(X, S).
```

Function add_role_type sets the role type in table types to the correct role

```
add_role_type(X) :-
    list_to_string(X,S),
    sformat(SQL,'UPDATE types SET role = "~w" WHERE role = "sngnbknd"',[S]),
    odbc_query(modelling,SQL).
```

7.4 Module process_answers.pl

Function prepare_answer retrieves the answered question from the database and then deletes it.

```
prepare_answer(Z,[]) :-
    sformat(SQL, 'SELECT questions
                  FROM agenda
                  WHERE id = ~w',
    [Z]),
    odbc_query(modelling, SQL, Question),
    isolate_result(Question, U),
    sformat(SQL2, 'DELETE
                  FROM agenda
                  WHERE questions = ("~w")',
    [U]),
    odbc_query(modelling, SQL2).

prepare_answer(Z, X) :-
    sformat(SQL, 'SELECT questions
                  FROM agenda
                  WHERE id = ~w',
    [Z]),
    odbc_query(modelling, SQL, Question),
    isolate_result(Question, U),
    concat_atom(List, ' ', U),
    split_at(List, [Head |Subs], ':'),
    flatten(Subs, Na),
    list_to_string(Na, Q),
    list_to_string(X, Antw),
    query(Query,[Head, [Q], [Antw]],[]),
    simplify_query(Query,C),
    execute(C).
```

Function simplify_query simplifies the query for execution.

```
simplify_query((S,C),C) :-
S,!.
simplify_query(C,C).
```

Function execute executes a function.

```
execute(C) :-
    C,
    !.
```

Function provide_object_type_for_object_instance sets the object type as provided in the answer for an object instance.

```
provide_object_type_for_object_instance(Instance, Antw) :-
   (
   sformat(SQL, 'SELECT objecttype
                 FROM objects
                 WHERE objecttype = "~w"
                 AND objectinstance = "~w"',
   [Antw, Instance]),
   findall(Query, odbc_query(modelling, SQL, row(Query)), X),
   X \= [],
   sformat(SQL2, 'DELETE
                 FROM objects
                 WHERE objectinstance = "~w"
                 AND objecttype IS NULL
                 LIMIT 1',
   [Instance]),
   odbc_query(modelling, SQL2);
   sformat(SQL3, 'UPDATE objects
                 SET objecttype = "~w"
                 WHERE objectinstance = "~w"
                 AND objecttype IS NULL
                 LIMIT 1',
   [Antw, Instance]),
   odbc_query(modelling, SQL3)
   ),
   (
   sformat(SQL4, 'SELECT *
                 FROM types
                 WHERE objecttype = "~w"',
   [Antw]),
   odbc_query(modelling, SQL4, _);
   sformat(SQL5, 'INSERT INTO types (objecttype)
                 VALUES ("~w")
                 ON DUPLICATE KEY UPDATE objecttype=objecttype',
   [Antw]),
   odbc_query(modelling, SQL5)
   ),
   create_table(Antw),
   insert_object(Antw, Instance).
```

Function provide_object_instance_for_object_type sets the object instance as provided in the answer for an object type.

```
provide_object_instance_for_object_type(Object, Antw) :-
    (
    sformat(SQL, 'SELECT objecttype
                   FROM objects
                   WHERE objectinstance = "~w"
                   AND objecttype = "~w"',
    [Antw, Object]),
    findall(Query, odbc_query(modelling, SQL, row(Query)), X),
    X \= [],
    sformat(SQL2, 'DELETE
                   FROM objects
                   WHERE objecttype = "~w"
                   AND objectinstance IS NULL
                   LIMIT 1',
    [Object]),
    odbc_query(modelling, SQL2);
    sformat(SQL2, 'UPDATE objects
                   SET objectinstance = "~w"
                   WHERE objecttype = "~w"
                   AND objectinstance IS NULL
                   LIMIT 1',
    [Antw, Object]),
    odbc_query(modelling, SQL2)
    ),
    (
    sformat(SQL3, 'SELECT *
                   FROM instances
                   WHERE objectinstance = "~w"',
    [Antw]),
    odbc_query(modelling, SQL3);
    sformat(SQL4, 'INSERT INTO instances (objectinstance)
                   VALUES ("~w")
                   ON DUPLICATE KEY UPDATE objectinstance=objectinstance',
                   [Antw]),
    odbc_query(modelling, SQL4)
    ),
    insert_object(Object,Antw).
```

Function provide_fact_type_for_object_type sets the fact type as provided in the answer for an object type.

```
provide_fact_type_for_object_type(Object, Antw) :-
    sformat(SQL, 'SELECT facttype
                   FROM types
                   WHERE facttype = "~w"
                   AND objecttype = "~w"',
                    [Antw, Object]),
    findall(Query, odbc_query(modelling, SQL, row(Query)), X),
    X \= [],
    sformat(SQL2, 'DELETE
                    FROM types
                    WHERE objecttype = "~w"
                    AND facttype IS NULL
                    LIMIT 1',
    [Object]),
    odbc_query(modelling, SQL2);
    sformat(SQL3, 'DELETE
                    FROM types
                    WHERE objecttype = "~w"
                    AND facttype IS NULL
                    LIMIT 1',
    [Object]),
    odbc_query(modelling, SQL3),
    string_to_atom(String,Antw),
    string_to_list(String,Temp),
    string_to_list(Temp,AntwString),
    wordlist(List,AntwString,[]),
    add_fact_type(List,_,List).
```

Function provide_fact_instance_for_object_instance sets the fact instance as provided in the answer for an object instance.

```
provide_fact_instance_for_object_instance(Object, Antw) :-
    sformat(SQL, 'SELECT factinstance
                    FROM instances
                    WHERE factinstance = "~w"
                    AND objectinstance = "~w"',
    [Antw, Object]),
    findall(Query, odbc_query(modelling, SQL, row(Query)), X),
    X \= [],
    sformat(SQL2, 'DELETE
                    FROM instances
                    WHERE objectinstance = "~w"
                    AND factinstance IS NULL
                    LIMIT 1',
    [Object]),
    odbc_query(modelling, SQL2);
    sformat(SQL3, 'DELETE
                    FROM instances
                    WHERE objectinstance = "~w"
                    AND factinstance IS NULL
                    LIMIT 1',
    [Object]),
    odbc_query(modelling, SQL3),
    string_to_atom(String,Antw),
    string_to_list(String,Temp),
    string_to_list(Temp,AntwString),
    wordlist(List,AntwString,[]),
    add_fact_instance(List,List).
```

> Function provide_fact_type_for_fact_instance sets the fact type as provided in the answer for a fact instance.

```
provide_fact_type_for_fact_instance(Instance, Antw) :-
    sformat(SQL, 'SELECT facttype
                    FROM facts
                    WHERE facttype = "~w"
                    AND factinstance = "~w"'
,   [Antw, Instance]),
    findall(Query, odbc_query(modelling, SQL, row(Query)), X),
    X \= [],
    sformat(SQL2, 'DELETE
                    FROM facts
                    WHERE factinstance = "~w"
                    AND facttype IS NULL
                    LIMIT 1',
    [Instance]),
    odbc_query(modelling, SQL2);
    sformat(SQL3, 'UPDATE facts
                    SET facttype = "~w"
                    WHERE factinstance = "~w"
                    AND facttype IS NULL
                    LIMIT 1 ',
    [Antw, Instance]),
    odbc_query(modelling, SQL3),
    string_to_atom(String,Antw),
    string_to_list(String,Temp),
    string_to_list(Temp,AntwString),
    wordlist(AntwList,AntwString,[]),
    add_fact_type(AntwList,_,AntwList),
    sformat(SQL4, 'INSERT INTO `~w` (instances)
                    VALUES ("~w")
                    ON DUPLICATE KEY UPDATE instances=instances',
    [Antw, Instance]),
    odbc_query(modelling, SQL4),
    string_to_atom(StringInst,Instance),
    string_to_list(StringInst,Temp2),
    string_to_list(Temp2,StringInstance),
    wordlist(InstanceList, StringInstance,[]),
    process_object_types(InstanceList, AntwList).
```

Function provide_fact_instance_for_fact_type sets the fact instance as provided in the answer for a fact type.

```
provide_fact_instance_for_fact_type(Fact, Antw) :-
    sformat(SQL, 'SELECT factinstance
                  FROM facts
                  WHERE factinstance = "~w"
                  AND facttype = "~w"',
    [Antw, Fact]),
    findall(Query, odbc_query(modelling, SQL, row(Query)), X),
    X \= [],
    sformat(SQL2, 'DELETE
                  FROM facts
                  WHERE facttype = "~w"
                  AND factinstance IS NULL
                  LIMIT 1',
    [Fact]),
    odbc_query(modelling, SQL2);
    sformat(SQL3, 'DELETE
                  FROM facts
                  WHERE facttype = "~w"
                  AND factinstance IS NULL
                  LIMIT 1',
    [Fact]),
    odbc_query(modelling, SQL3),
    sformat(SQL4, 'INSERT INTO facts (facttype,factinstance)
                  VALUES ("~w","~w")
                  ON DUPLICATE KEY UPDATE factinstance=factinstance',
    [Fact, Antw]),
    odbc_query(modelling, SQL4),
    sformat(SQL5, 'INSERT INTO `~w` (instances)
                  VALUES ("~w")
                  ON DUPLICATE KEY UPDATE instances=instances',
    [Fact, Antw]),
    odbc_query(modelling, SQL5),
    string_to_atom(AntwStr,Antw),
    string_to_list(AntwStr,Temp),
    string_to_list(Temp,AntwString),
    string_to_atom(FactStr,Fact),
    string_to_list(FactStr,Temp2),
    string_to_list(Temp2,FactString),
    wordlist(FactList, FactString,[]),
    wordlist(AntwList, AntwString,[]),
    process_object_instances(FactList, AntwList).
```

Function provide_constraint_for_fact_type sets the constraint as provided in the answer for a fact type.

```
provide_constraint_for_fact_type(Fact, Antw) :-
    sformat(SQL, 'UPDATE types
                    SET uniqueness_constraint = "~w"
                    WHERE facttype = "~w"', [Antw, Fact]),
    odbc_query(modelling, SQL).
```

Function process_object_instances deletes the object types without instances as provided in the answer for that object type, adds an entry for object and instance together and adds an entry in the table for the object type with its instance.

```
process_object_instances([],[]).
process_object_instances([X],[Y]) :-
    capitalized(X),
    sformat(SQL, 'DELETE
                    FROM objects
                    WHERE objecttype="~w"
                    AND objectinstance IS NULL
                    LIMIT 1',
    [X]),
    sformat(SQL2, 'DELETE
                    FROM objects
                    WHERE objectinstance="~w"
                    AND objecttype IS NULL
                    LIMIT 1',
    [Y]),
    sformat(SQL3, 'INSERT INTO objects (objecttype, objectinstance)
                    VALUES ("~w","~w")
                    ON DUPLICATE KEY UPDATE objecttype=objecttype',
    [X, Y]),
    sformat(SQL4, 'INSERT INTO ~w (instances)
                    VALUES ("~w")
                    ON DUPLICATE KEY UPDATE instances=instances',
    [X, Y]),
    odbc_query(modelling, SQL),
    odbc_query(modelling, SQL2),
    odbc_query(modelling, SQL3),
    odbc_query(modelling, SQL4);
    !.
process_object_instances([X |Xs],[Y |Ys]) :-
```

```
    \+ capitalized(X),
    process_object_instances(Xs,Ys);
    capitalized(X),
    sformat(SQL, 'DELETE
                    FROM objects
                    WHERE objecttype="~w"
                    AND objectinstance IS NULL
                    LIMIT 1',
    [X]),
    sformat(SQL2, 'DELETE
                    FROM objects
                    WHERE objectinstance="~w"
                    AND objecttype IS NULL
                    LIMIT 1',
    [Y]),
    sformat(SQL3, 'INSERT INTO objects (objecttype, objectinstance)
                    VALUES ("~w","~w")
                    ON DUPLICATE KEY UPDATE objecttype=objecttype',
    [X, Y]),
    sformat(SQL4, 'INSERT INTO ~w (instances)
                    VALUES ("~w")
                    ON DUPLICATE KEY UPDATE instances=instances',
    [X, Y]),
    odbc_query(modelling, SQL),
    odbc_query(modelling, SQL2),
    odbc_query(modelling, SQL3),
    odbc_query(modelling, SQL4),
    process_object_instances(Xs,Ys).
```

Function process_object_types deletes the object instances without types as provided in the answer for that object instance, adds an entry for object and instance together and adds an entry in the table for the object type with its instance.

```
process_object_types([],[]).
process_object_types([X],[Y]) :-
    capitalized(X),
    sformat(SQL, 'DELETE
                    FROM objects
                    WHERE objectinstance="~w"
                    AND objecttype IS NULL',
    [X]),
    sformat(SQL2, 'DELETE
                    FROM objects
                    WHERE objecttype="~w"
                    AND objectinstance IS NULL',
    [Y]),
    sformat(SQL3, 'INSERT INTO objects (objecttype, objectinstance)
                    VALUES (''~w'',''~w'')
                    ON DUPLICATE KEY UPDATE objecttype=objecttype',
    [Y, X]),
    create_table(Y),
    sformat(SQL4, 'INSERT INTO ~w (instances)
                    VALUES (''~w'')
                    ON DUPLICATE KEY UPDATE instances=instances',
    [Y, X]),
    odbc_query(modelling, SQL),
    odbc_query(modelling, SQL2),
    odbc_query(modelling, SQL3),
    odbc_query(modelling, SQL4);
    !.
process_object_types([X |Xs],[Y |Ys]) :-
    \+ capitalized(X),
    process_object_types(Xs,Ys);
    capitalized(X),
    sformat(SQL, 'DELETE
                    FROM objects
                    WHERE objectinstance=''~w''
                    AND objecttype IS NULL',
    [X]),
    sformat(SQL2, 'DELETE
                    FROM objects
                    WHERE objecttype=''~w''
                    AND objectinstance IS NULL',
    [Y]),
    sformat(SQL3, 'INSERT INTO objects (objecttype, objectinstance)
                    VALUES (''~w'',''~w'')
                    ON DUPLICATE KEY UPDATE objecttype=objecttype',
    [Y, X]),
    create_table(Y),
    sformat(SQL4, 'INSERT INTO ~w (instances)
                    VALUES (''~w'')
                    ON DUPLICATE KEY UPDATE instances=instances', [Y, X]),
    odbc_query(modelling, SQL),
    odbc_query(modelling, SQL2),
    odbc_query(modelling, SQL3),
    odbc_query(modelling, SQL4),
    process_object_types(Xs,Ys).
```

DCG to transform the questions into corresponding functions.

```
query((Q)) -->
```

```
   command(Q).

command((X^Y^provide_object_type_for_object_instance(X,Y)))
   –>[['Provide', at, least, one, object, type, for, object, instance] , [X], [Y]].
command((X^Y^provide_object_type_for_object_instance(X,Y)))
   –>[['Immediately', provide, at, least, one, object, type, for, object, instance], [X], [Y]].

command((X^Y^provide_object_instance_for_object_type(X,Y)))
   –>[['Provide', at, least, one, object, instance, for, object, type], [X], [Y]].
command((X^Y^provide_object_instance_for_object_type(X,Y)))
   –>[['Immediately', provide, at, least, one, object, instance, for, object, type], [X], [Y]].

command((X^Y^provide_fact_type_for_object_type(X,Y)))
   –>[['Provide', at, least, one, fact, type, for, object, type], [X], [Y]].
command((X^Y^provide_fact_type_for_object_type(X,Y)))
   –>[['Immediately', provide, at, least, one, fact, type, for, object, type], [X], [Y]].

command((X^Y^provide_fact_instance_for_object_instance(X,Y)))
   –>[['Provide', at, least, one, fact, instance, for, object, instance], [X], [Y]].
command((X^Y^provide_fact_instance_for_object_instance(X,Y)))
   –>[['Immediately', provide, at, least, one, fact, instance, for, object, instance], [X], [Y]].

command((X^Y^provide_fact_type_for_fact_instance(X,Y)))
   –>[['Provide', at, least, one, fact, type, for, fact, instance], [X], [Y]].
command((X^Y^provide_fact_type_for_fact_instance(X,Y)))
   –>[['Immediately', provide, at, least, one, fact, type, for, fact, instance], [X], [Y]].

command((X^Y^provide_fact_instance_for_fact_type(X,Y)))
   –>[['Provide', at, least, one, fact, instance, for, fact, type], [X], [Y]].
command((X^Y^provide_fact_instance_for_fact_type(X,Y)))
   –>[['Immediately', provide, at, least, one, fact, instance, for, fact, type], [X], [Y]].

command((X^Y^provide_constraint_for_fact_type(X,Y)))
   –>[['Provide', at, least, one, constraint, for, fact, type], [X], [Y]].
command((X^Y^provide_constraint_for_fact_type(X,Y)))
   –>[['Immediately', provide, at, least, one, constraint, for, fact, type], [X], [Y]].
```

7.5 Module output_agenda.pl

Function output_agenda is used to reorder the agenda, reorder the numbering of the questions and output the agenda

```
output_agenda :-
      reorder_agenda,
      get_questions(X),
      X \= [],
      nl, nl,
      process_results(X),
      nl,
      !;
      nl,nl,
      write('Agenda is empty'),
      !.
```

Function reorder_agenda is used to reorder the agenda and the numbering of the questions

```
reorder_agenda :-
      odbc_query(modelling, 'CREATE TABLE newagenda
                              (
                              type ENUM("IMM","NORM") NOT NULL,
                              id MEDIUMINT NOT NULL AUTO_INCREMENT,
                              questions VARCHAR(256) UNIQUE,
                              PRIMARY KEY (type,id)
                              )
                              ENGINE = MYISAM'),
      odbc_query(modelling, 'INSERT INTO newagenda
                              SELECT * FROM agenda ORDER BY type'),
      odbc_query(modelling, 'DROP TABLE agenda'),
      odbc_query(modelling, 'RENAME TABLE newagenda TO agenda'),

      /* First, double the max id from agenda, to avoid double key entries */
      odbc_query(modelling, 'SELECT COUNT(*)
                              FROM agenda',
                  W,
                  [types([integer])]),
      isolate_result(W,F),
      string_to_list(F,G),
      name(Total,G),
      NewTotal is Total * 2,
      sformat(SQL, 'SET @var_name =  w',[NewTotal]),
      odbc_query(modelling, SQL),
      odbc_query(modelling, 'UPDATE agenda
                                        SET id = (@var_name := @var_name +1)'),

      /* Then reset count from 1 */
      odbc_query(modelling, 'SET @var_name2 = 0'),
      odbc_query(modelling, 'UPDATE agenda
                                        SET id = (@var_name2 := @var_name2 +1)'),
      odbc_query(modelling, 'SELECT COUNT(*)
                                        FROM agenda',
                       W,
                       [types([integer])]),
      isolate_result(W,E),
      string_to_list(E,M),
      name(Int,M),
      NewInt is Int + 1,
      sformat(SQL2, 'ALTER TABLE agenda AUTO_INCREMENT =  w', [NewInt]),
      odbc_query(modelling, SQL2).
```

Function process_results is used to isolate the questions from the database results

```
process_results([]).
process_results([X]) :-
    isolate_result(X,U),
    write(U), nl.

process_results([X|Xs]) :-
    isolate_result(X,U),
    write(U), nl,
    process_results(Xs).
```

Function get_questions is used to retrieve the questions from the database

```
get_questions(Y) :-
    findall(X, odbc_query(modelling, 'SELECT id, questions
                                                    FROM agenda',
             X),
    Y).
```

7.6 Module business_rules.pl

Function fill_agendas first deletes the old agenda, and then searches for all entries in the database that conflict with the ruleset.

```
fill_agendas :-
    odbc_query(modelling, 'DELETE FROM agenda'),
    ruleset(R),
    zoek_tegenmodel(R, _).
```

The set of business rules for ORM.

```
ruleset(R) :-
    R =
    (
    [
    ["Ieder feittype wordt gepopuleerd door minimaal een feitinstantie", facts, 0],
    ["Iedere feitinstantie hoort bij minimaal een feittype", facts, 1],
    ["Ieder objecttype wordt gepopuleerd door minimaal een objectinstantie", objects, 0],
    ["Iedere objectinstantie hoort bij minimaal een objecttype", objects, 1],
    ["Ieder objecttype participeert in minimaal een feittype", types, 0],
    ["Ieder feittype bestaat uit minimaal een objecttype", types, 0],
    ["Iedere rol participeert in minimaal een feittype", types, 0],
    ["Ieder feittype bestaat uit minimaal een rol", types, 0],
    ["Iedere objectinstantie participeert in minimaal een feitinstantie", instances, 0],
    ["Iedere feitinstantie bestaat uit minimaal een objectinstantie", instances, 0],
    ["Ieder feittype wordt beperkt door minimaal een uniqueness constraint", types, 0],
    ["Iedere uniqueness constraint beperkt minimaal een feittype", types, 0]
    ]
    ).
```

The set of business rules that are incorporated in code (database restrictions):
Every object type is unique (Principle of strong identification)
Every fact type is unique (Principle of strong identification)

And one for the modeller to check:
Every fact type is elementary

Function zoek_tegenmodel reads every rule in the ruleset, transforms it into a logic formula and parses its elements to find the corresponding database columns. It then checks if the formula is true for all entries in these columns and if not, stores these entries in a resultset.

```
zoek_tegenmodel([],[]).
zoek_tegenmodel([[Rule, Table, Prio] |Rules], Resultset) :-
    lees_tekst_naar_lijst_binair(List, Rule),
    formulate(List,Formula),
    parse(Formula, Col1, Col2),
    retrieveContradictions(Table, Col1, Col2, Resultset),
    create_questions(Col1, Col2, Prio, Resultset),
    zoek_tegenmodel(Rules, _).
```

Function lees_tekst_naar_lijst_binair converts text to the Prolog list format.

```
lees_tekst_naar_lijst_binair(L,T) :-
    wordlist(L,T,[]).
```

Function formulate converts the list to a logic formula.

```
formulate(List,Formula) :-
    logic_formula(Formula,List,[]),!.
```

Function parse selects the database columns to check.
Momentarily only works for formulas quantified as ALL.x EX.y.

```
parse((Q1,Q2,E1,E2,_), E1, E2) :-
    Q1 = all(_),
    Q2 = ex(_);
    fail.
```

Function retrieveContradictions selects all NULL-values.
Momentarily only works for formulas quantified as ALL.x EX.y.

```
retrieveContradictions(Table, Col1, Col2, Resultset) :-
    sformat(SQL1,'SELECT ~w
                  FROM ~w
                  WHERE ~w IS NULL',
    [Col1, Table, Col2]),
    findall(Y, odbc_query(modelling, SQL1, Y),Resultset).
```

Function createQuestions creates the questions for the agenda.
It tidies the query results and transforms them into questions.

```
create_questions(_,_,_,[]).
create_questions(Col1, Col2, Prio, [Result |Results]) :-
    isolate_result(Result,X),
    tidy_column(Col2, Tidy2),
    tidy_column(Col1, Tidy1),
    write_prio(Prio, PrioRes, PrioDB),
    concat(PrioRes, Tidy2, C1),
    concat(C1, ' for ', C2),
    concat(C2, Tidy1, C3),
    concat(C3, ' : ', C4),
    concat(C4, X, C5),
    sformat(SQL2,'INSERT INTO agenda (type, questions)
                    VALUES ("~w", "~w")
                    ON DUPLICATE KEY UPDATE questions=questions',
    [PrioDB, C5]),
    odbc_query(modelling, SQL2),
    Results [],
    create_questions(Col1, Col2, Prio, [Results]);
    !.
```

Function isolate_result strips the query results.

```
isolate_result(X,U) :-
    sformat(F,'~w',X),
    atom_codes(F,G),
    starts_with(G,"row(",Z),
    ends_with(Z,")",Y),
    atom_chars(U,Y).

starts_with(Whole, Part, X) :- append(Part, X, Whole).
ends_with(Whole, Part, X) :- append(X, Part, Whole).
```

Function tidy_column returns the full name of a database column.

```
tidy_column(X,Y) :-
    X = facttype,
    Y = 'fact type';
    X = factinstance,
    Y = 'fact instance';
    X = objecttype,
    Y = 'object type';
    X = objectinstance,
    Y = 'object instance';
    X = uniqueness_constraint,
    Y = 'constraint'.
```

Function write_prio returns the start of a question, depending on the priority of the corresponding business rule.

```
write_prio(Prio, PrioRes, PrioDB) :-
    Prio == 0,
    PrioRes = 'Provide at least one ',
    PrioDB = 'NORM';
    Prio == 1,
    PrioRes = 'Immediately provide at least one ',
    PrioDB = 'IMM'.
```

```
DCG of the business logic.

logic_formula returns the logic formula:
quantifier(X), quantifier(Y), element(X), element(Y), function(X,Y)

quantifiedVariable returns: quantifier(X), element(X), X
```

```
logic_formula((Q1,Q2,E1,E2,F)) -->
    quantifiedVariable((Q1,_,X,E1)),
    functie((X^Y^F)),
    quantifiedVariable((Q2,_,Y,E2)).

quantifiedVariable((Z,Y,X,U)) -->
    quantifier((X^Z)),
    element((X^Y^U)).

    quantifier((X^ex(X))) -->[minimaal, een].
quantifier((X^all(X))) -->[ieder].
quantifier((X^all(X))) -->['Ieder'].
quantifier((X^all(X))) -->[iedere].
quantifier((X^all(X))) -->['Iedere'].
quantifier((X^all(X))) -->[alle].
quantifier((X^all(X))) -->['Alle'].

element((X^facttype(X)^facttype)) -->[feittype].
element((X^factinstance(X)^factinstance)) -->[feitinstantie].
element((X^objecttype(X)^objecttype)) -->[objecttype].
element((X^objectinstance(X)^objectinstance)) -->[objectinstantie].
element((X^role(X)^role)) -->[rol].
element((X^uniqueness_constraint(X)^uniqueness_constraint)) -->[uniqueness,
constraint].

element(facttype) -->[feittype].
element(factinstance) -->[feitinstantie].
element(objecttype) -->[objecttype].
element(objectinstance) -->[objectinstantie].
element(role) -->[rol].
element(uniqueness_constraint) -->[uniqueness, constraint].
```

```
functie((X^Y^populeert(X,Y))) -->[wordt, gepopuleerd, door].
functie((X^Y^populeert(Y,X))) -->[hoort, bij].
functie((X^Y^participeert_in(X,Y))) -->[participeert, in].
functie((X^Y^participeert_in(Y,X))) -->[bestaat, uit].
functie((X^Y^beperkt(X,Y))) -->[beperkt].
functie((X^Y^beperkt(Y,X))) -->[wordt, beperkt, door].
```

7.7 Module sql.pl

Function connect_db for connecting to the database.

```
connect_db(DBName, User, Pass) :-
    odbc_connect( DBName, _,
                  [ user(User),
                  password(Pass),
                  alias(modelling),
                  open(once)
                  ]).
```

Function create_tables creates the necessary tables on startup.

```
create_tables :-
    odbc_query(modelling,
                'CREATE TABLE IF NOT EXISTS agenda
                (type ENUM("IMM","NORM") NOT NULL,
                id SMALLINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
                questions VARCHAR(256) UNIQUE)
                ENGINE = MYISAM'),
    odbc_query(modelling,
                'CREATE TABLE IF NOT EXISTS types (
                id SMALLINT(10) PRIMARY KEY AUTO_INCREMENT,
                facttype VARCHAR(254),
                objecttype VARCHAR(254),
                role VARCHAR(254),
                uniqueness_constraint CHAR(3) )'),
    odbc_query(modelling,
                'ALTER TABLE types
                ADD UNIQUE (facttype, objecttype)'),
    odbc_query(modelling,
                'CREATE TABLE IF NOT EXISTS instances (
                id SMALLINT(10) PRIMARY KEY AUTO_INCREMENT,
                factinstance VARCHAR(254),
                objectinstance VARCHAR(254))'),
    odbc_query(modelling,
                'CREATE TABLE IF NOT EXISTS facts (
                id SMALLINT(10) PRIMARY KEY AUTO_INCREMENT,
                facttype VARCHAR(254),
                factinstance VARCHAR(254) UNIQUE)'),
    odbc_query(modelling,
                'ALTER TABLE facts
                ADD UNIQUE (facttype,factinstance)'),
    odbc_query(modelling,
                'CREATE TABLE IF NOT EXISTS objects (
                id SMALLINT(10) PRIMARY KEY AUTO_INCREMENT,
                objecttype VARCHAR(254),
                objectinstance VARCHAR(254),
                factinstance VARCHAR(254))'),
    odbc_query(modelling,
                'ALTER TABLE objects
                ADD UNIQUE (objecttype,objectinstance)');
    write('Please select an empty database'),nl,
    stop.
```

Function create_table creates a table for an object.

```
create_table(TableName) :-
   sformat(SQL,'CREATE TABLE IF NOT EXISTS '~w'
                  (instances VARCHAR(256) PRIMARY KEY)',
                  [TableName]),
   odbc_query(modelling, SQL).
```

Function insert inserts a value into a table.

```
create_table(TableName) :-
insert_X(X,Table,Column) :-
   sformat(SQL,'INSERT INTO ~w (~w)
                  VALUES ("~w")
                  ON DUPLICATE KEY UPDATE ~w=~w',
                  [Table,Column,X, Column, Column]),
   odbc_query(modelling, SQL).
```

Function check_existence checks if a value already exists in a table.

```
check_existence(X,Table, Column) :-
   sformat(SQL2, 'SELECT *
                  FROM ~w
                  WHERE ~w = ("~w")',
                  [Table, Column, X]),
   odbc_query(modelling, SQL2, _).
```

Function create_table creates a table.

```
insert_fact_into_types(Fact, X) :-
   sformat(SQL2, 'DELETE
                  FROM types
                  WHERE objecttype = "~w"
```

```
                    AND facttype IS NULL
                    LIMIT 1',
    [X]),
                    odbc_query(modelling, SQL2),
    sformat(SQL,'INSERT INTO types (facttype, objecttype, role)
                    VALUES ("~w", "~w", "sngnbknd")
                    ON DUPLICATE KEY UPDATE facttype=facttype',
                    [Fact, X]),
    odbc_query(modelling, SQL).
```

Function insert_object_into_types inserts an object in table types.

```
insert_object_into_types(Object) :-
    sformat(SQL,'SELECT *
                    FROM types
                    WHERE objecttype = ("~w")',
                    [Object])
    odbc_query(modelling, SQL, X),
    X \= row('null');
    sformat(SQL2,'INSERT INTO types (objecttype)
                    VALUES ("~w")',
                    [Object]),
    odbc_query(modelling, SQL2).
```

Function insert_fact_into_instances inserts a fact in table instances.

```
insert_fact_into_instances(Fact, X) :-
    sformat(SQL,'INSERT INTO instances (factinstance, objectinstance)
                    VALUES ("~w", "~w")
                    ON DUPLICATE KEY UPDATE factinstance=factinstance',
                    [Fact, X]),
    odbc_query(modelling, SQL).
```

Function insert_object_into_instances inserts an object in table instances.

```
insert_object_into_instances(Object) :-
    sformat(SQL2,'INSERT INTO instances (objectinstance)
                    VALUES ("~w")',
                    [Object]),
    odbc_query(modelling, SQL2).
```

Function insert_object_and_fact_into_objects inserts an object and a fact in table objects.

```
insert_object_and_fact_into_objects(Object, Fact) :-
    sformat(SQL2,'INSERT INTO objects (objecttype, factinstance)
                    VALUES ("~w","~w")',
                    [Object, Fact]),
    odbc_query(modelling, SQL2).
```

Function insert_object_and_instance_into_objects inserts an object and a fact instance in table objects.

```
insert_object_and_instance_into_objects(Object, Fact) :-
    sformat(SQL2,'INSERT INTO objects (objectinstances, factinstance)
                    VALUES ("~w","~w")',
                    [Object, Fact]),
    odbc_query(modelling, SQL2).
```

Function insert_object inserts an object.

```
insert_object(TableName, Object) :-
    sformat(SQL,'INSERT INTO ~w (instances)
                    VALUES ("~w")
                    ON DUPLICATE KEY UPDATE instances=instances',
                    [TableName, Object]),
    odbc_query(modelling, SQL).
```