Square Support

Towards a prototype for automated support of object-oriented modeling
based on information grammars

Master’s Thesis No. 421, July 31, 1997

Student:
Michael van de Ven

Thesis supervisor
Dr. P. van Bommel
Contents

Preface v

Chapter 1 Introduction 1

1.1 Problem area 1
   1.1.1 Information grammars 2
   1.1.2 Object-oriented modeling 3
1.2 Problem statement 3
1.3 Thesis outline 5

Chapter 2 Problem analysis 7

2.1 Modeling process 7
   2.1.1 Domain expert 7
   2.1.2 System analyst 8
   2.1.3 Way of working 9
   2.1.4 Way of modeling 10
   2.1.5 Way of supporting 11
2.2 Automated support 11
   2.2.1 Abstracting the information grammar 12
   2.2.2 Communicating the information grammar 13
2.3 Delphi programming 14
2.4 Structured Query Language (SQL) 15
   2.4.1 Embedded SQL 15
2.5 Summary and outlook 16

Chapter 3 Design of automated support 17

3.1 System design 17
3.2 Abstracting the information grammar 20
   3.2.1 Object action involvement model 20
   3.2.2 Object property model 29
   3.2.3 Object life model 34
   3.2.4 Information grammar 38
3.3 Communicating the information grammar 39
   3.3.1 Identification of object instances 39
   3.3.2 Population structure of objects 40
   3.3.3 Sentence generation and validation 46

3.4 Summary and outlook 47

Chapter 4 Prototype: Square Support 49

   4.1 From process models to graphical user-interface 49
   4.2 Creating tables and queries 57
   4.3 Data access through database components 58
   4.4 Summary and outlook 60

Chapter 5 Conclusions and further research 61

   5.1 Conclusions 61
   5.2 Further research 62

Appendix A Graphical Notation 63

Appendix B Paradox Tables 65

   B.1 Input of analysis models 65
   B.2 Populating the analysis models 67

Appendix C Program files 69

Bibliography 71

Index 73
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>Overview of the analysis phase</td>
<td>2</td>
</tr>
<tr>
<td>Figure 1.2</td>
<td>Problem area of the tool in the analysis phase</td>
<td>4</td>
</tr>
<tr>
<td>Figure 1.3</td>
<td>Reading this thesis</td>
<td>5</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>Overview of the analysis phase</td>
<td>10</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>System design of the CASE-tool</td>
<td>17</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Process model of the application</td>
<td>18</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Process model of models</td>
<td>20</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>Syntax Object Action Involvement Model in PSM with specialization</td>
<td>22</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>Naming and verbalization in PSM</td>
<td>23</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>Syntax Object Action Involvement Model in APSM</td>
<td>24</td>
</tr>
<tr>
<td>Figure 3.7</td>
<td>Naming and verbalization in APSM</td>
<td>26</td>
</tr>
<tr>
<td>Figure 3.8</td>
<td>Process model of OAIM</td>
<td>28</td>
</tr>
<tr>
<td>Figure 3.9</td>
<td>Syntax Object Property Model in PSM with specialization</td>
<td>30</td>
</tr>
<tr>
<td>Figure 3.10</td>
<td>Naming and verbalization in PSM</td>
<td>30</td>
</tr>
<tr>
<td>Figure 3.11</td>
<td>Syntax Object Property Model in APSM</td>
<td>31</td>
</tr>
<tr>
<td>Figure 3.12</td>
<td>Task structure of OPM</td>
<td>33</td>
</tr>
<tr>
<td>Figure 3.13</td>
<td>Syntax Object Life Model in PSM with specialization</td>
<td>35</td>
</tr>
<tr>
<td>Figure 3.14</td>
<td>Syntax Object Life Model in APSM</td>
<td>36</td>
</tr>
<tr>
<td>Figure 3.15</td>
<td>Task structure of OLM</td>
<td>37</td>
</tr>
<tr>
<td>Figure 3.16</td>
<td>Populating label types</td>
<td>40</td>
</tr>
<tr>
<td>Figure 3.17</td>
<td>Populating elementary object types</td>
<td>40</td>
</tr>
<tr>
<td>Figure 3.18</td>
<td>Populating group types</td>
<td>41</td>
</tr>
<tr>
<td>Figure 3.19</td>
<td>Populating sequence types</td>
<td>42</td>
</tr>
<tr>
<td>Figure 3.20</td>
<td>Populating module types</td>
<td>43</td>
</tr>
<tr>
<td>Figure 3.21</td>
<td>Populating action types</td>
<td>44</td>
</tr>
<tr>
<td>Figure 3.22</td>
<td>Extension data access of OAIM</td>
<td>45</td>
</tr>
<tr>
<td>Figure 3.23</td>
<td>Extension data access of OPM</td>
<td>46</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Forms used with the CASE-tool</td>
<td>49</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Graphical form Square Support</td>
<td>50</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Graphical form New</td>
<td>51</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Graphical form Open</td>
<td>52</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>Graphical form Models – pagecontrol OAIM</td>
<td>53</td>
</tr>
<tr>
<td>Figure 4.6</td>
<td>Graphical form Models – pagecontrol OPM</td>
<td>54</td>
</tr>
<tr>
<td>Figure 4.7</td>
<td>Graphical form Models – pagecontrol OLM</td>
<td>55</td>
</tr>
<tr>
<td>Figure 4.8</td>
<td>Graphical form About</td>
<td>56</td>
</tr>
<tr>
<td>Figure 4.9</td>
<td>Data modules used with the CASE-tool</td>
<td>57</td>
</tr>
<tr>
<td>Figure 4.10</td>
<td>Database components architecture</td>
<td>59</td>
</tr>
<tr>
<td>Figure 4.11</td>
<td>Connection of forms with data modules</td>
<td>60</td>
</tr>
</tbody>
</table>
Preface

With this Master's thesis I finish my study Computing Science at the University of Nijmegen, The Netherlands, which I started in September 1992. With this, I also finish my terrific years of study. In addition I intended to make a positive scientific contribution to Object-Oriented Modeling based on Information Grammars with this thesis subject.

Within the Student Research Lab\(^1\) I was placed in the Research Team *Hypertext, information retrieval and natural language*. This Research Team is involved with *hypertext and information retrieval* and with automatic processing of *natural language*. Natural language processing can be applied in information modeling in several ways, which can be divided in syntactic and semantic approaches.

My thesis supervisor was Dr. P. van Bommel, of the research-line Information Systems\(^2\) within the Faculty of Mathematics and Informatics of the University of Nijmegen. The research line Information Systems concentrates its research around the topics *data modeling* and *information retrieval*. Special attention is paid to the integration of both. In both topics the use of informal language is an important research theme.

Nijmegen, July 1997

Michael van de Ven

---

\(^1\) [http://www.cs.kun.nl/is/edu/afstud/lab/](http://www.cs.kun.nl/is/edu/afstud/lab/)

\(^2\) [http://www.cs.kun.nl/is/](http://www.cs.kun.nl/is/)
Chapter 1

Introduction

In this chapter the problem area and the problem statement of this thesis are explained. References are made to related work, which fits within the subject of this thesis. First section 1.1 gives a description of the problem area. Also the main subjects within the problem area, information grammars and object-oriented modeling, are elaborated in this section. From there, in section 1.2 the problem statement of this thesis is explained. Finally, section 1.3 gives a guideline how to read this thesis.

1.1 Problem area

In many articles and experiences in the field of information system development, the analysis phase has come up as the bottleneck of the information system development process. It is also known that the later in this process, an error is detected, the more time and money has to be invested in the correction of this error.

A successful information system development depends on the two following factors, which have to be focussed during the analysis phase:

- flexibility in the analysis phase;
- communication with the information system.

The flexibility of the information system can be promoted by use of object-orientation. Besides the communication of the user with the information system is promoted by conceptual modeling based on natural language (see also [FKW96] and [BFW96]).

Figure 1.1 provides an overview of the analysis phase, which is based on use of natural language. The stages are passed through until the textual description of the conceptual model matches the informal specification within the natural language specification.

In [Fre97] a formal framework is explained for the derivation, verification and validation of object-oriented analysis models based on information grammars, intended for the specification of flexible and communication-oriented information systems. With this, Frederiks initiates implementation of a tool for as well as the system analyst as the domain expert, but finally to support the validation of the informal specification by the domain expert.
The development of such a tool requires further research to support paraphrasing constraints and populations from the object-oriented analysis models to an information grammar, and finally of paraphrasing this grammar to a textual description, to validate the informal specification (see [DFW96]). Furthermore, all this has to be designed properly before it can be implemented.

The different facets, like natural language and object-oriented modeling, within this problem area, provide an interesting combination of a theoretical and a practical problem, which is explained in the section problem statement.

### 1.1.1 Information grammars

Most of the latest modeling techniques, start with a description of the domain which has to be modeled in natural language, also called the Universe of Discourse. That description is called the initial specification, from which the required information system can be derived and is provided by domain experts.

Also several conceptual data modeling techniques, including PSM (see [Hof93]), are based on a natural language approach. These techniques aim on deriving the grammar that governs the communication within the Universe of Discourse. This grammar is called the information grammar and it can be depicted as an information structure diagram.
One of the advantages of the natural language approach is readily validating the results by the domain expert, as each result corresponding to a (partial) grammar for sentences in the language spoken in the Universe of Discourse (see [FKW95]).

1.1.2 Object-oriented modeling

In object-oriented analysis several specific views on the nature of the application domain can be abstracted from the normal form specification. Each view has its corresponding object-oriented analysis model, and represents a (partial) linguistic view on the information grammar. For feedback purposes, these models have an associated verbalization mechanism, which enables the system analyst to describe the model and its instantiations in a terminology understandable for informed users. The next three object-oriented analysis models are distinguished:

- **Object action involvement model**: This model describes the types of objects and actions that have been determined from the normal form specification. It provides the relationships from object types with action types, and also the responsibilities of object types for the action types.

- **Object property model**: This model describes the properties of object types, the static aspects of the application domain.

- **Object life model**: This model considers the application domain from a historical perspective, and describes for each object type the course of life of its instances.

A complete description of the dynamic and static aspects of the intended system is provided by the information architecture, which describes the conceptual object class hierarchy. This conceptual model of the Universe of Discourse is a composition of the three earlier mentioned analysis models (see [FKW96]).

1.2 Problem statement

The problem statement of this thesis can be derived from the previous introduction and elaboration of the problem area, and is formulated by the following statement:

*Developing towards a prototype of a CASE-tool to support paraphrasing the conceptual model in the analysis phase of object-oriented modeling based on information grammars, using an information grammar derived from object-oriented analysis models.*

Automated tool support for software engineers should lead to corresponding improvements in productivity. A wide range of tools to support software development have been developed and the term computer-aided software engineering (CASE) has come into use as a generic term for the automated support of software engineering. By developing a prototype of a CASE-tool, a working, albeit limited, system is available quickly to demonstrate the feasibility and usefulness of the application. Also difficult to use or confusing
user services may be identified and refined and the prototype serves as a basis for writing the further specification of the CASE-tool.

Developing a CASE-tool requires a practical application, based on theories and techniques derived from the information systems and software engineering courses, and most of all on the work of Frederiks. With this, the following must be taken into account:

*How should a CASE-tool support object-oriented modeling based on information grammars in the most optimal way, and in which way should it be designed?*

Figure 1.2 is like the overview depicted in figure 1.1, the overview of the analysis phase. The definition of the problem area of the tool, as it comes up in the problem statement, is represented by the solid lines. The dashed lines are not covered by the CASE-tool.

---

**Figure 1.2: problem area of the tool in the analysis phase**

Interpretation of the *object-oriented analysis models*, which are modeled from the normal form specification, provides an *information grammar*. Interpretation of an according *population*, which is also modeled from the normal form specification, provides a *lexicon*. From the grammar and lexicon a *textual description* is produced by the system analyst, which is used by the domain expert, to validate the *informal specification*.

In other words, this thesis is intended to design and implement the automated generation of the textual description for validation of their application domain by the domain experts, and for validation of the consistency of the system, by the system analysts.
Beside the subject of this thesis it is also important to mention briefly, that theories developed at the University of Nijmegen remain not only theories but can be put into practice, used and even farther developed by students and researchers. With this thesis as an example I also point at the practical part of the courses “Information Systems”, which should put the taught theories into practice in the near future, using CASE-tools or even developing or improving them.

1.3 Thesis outline

This thesis can be read in different ways, as indicated in figure 1.3. To get a quick view on this thesis, one reads the introduction and the conclusions with aspects regarding further research.

The chapters “Problem analysis” and “Design of automated support” must be read with some know-ledge of information systems, modeling and using them. These are based on the work of Frederiks [Fre97] and elaborate the fundamentals of the CASE-tool. The chapter “Problem analysis” defines the borders and requirements of the problem area of this thesis and how to start the design of the CASE-tool. Then in chapter “Design of automated support” this CASE-tool is designed according section “Problem analysis”.

Figure 1.3: reading this thesis

The chapter “Prototype: Square Support” considers usage and possibilities of the prototype “Square Support”, which is based on the results from chapter “Design of automated support”. The final chapter considers the conclusions and further research of this thesis.
Chapter 2

Problem analysis

This chapter is intended to define the borders and the requirements of the problem area of this thesis. The underlying theoretical knowledge is significant for the following stages in this thesis. In this chapter, the way of supporting, divided into abstracting and communicating the information grammar, is explained from the modeling process. In section 2.1 the whole modeling process is explained, towards automated support in section 2.2. Finally section 2.3 considers the Delphi programming environment, section 2.4 considers the Structured Query Language, both used for development of the CASE-tool.

2.1 Modeling process

In [Fre97] a formal framework is explained for the derivation, verification and validation of object-oriented analysis models based on information grammars, intended for the specification of flexible and communication-oriented information systems.

One of the advantages of this natural language approach is readily validating the results by the domain expert, as each result corresponding to a (partial) grammar for sentences in the language spoken in the Universe of Discourse (see [FKW95]). Mainly to exploit this advantage, a part of the accompanying modeling process is supported by building a CASE-tool.

The framework provided by Frederiks, has as point of departure a description in natural language of the Universe of Discourse. Using natural language requires a number of skills for the people involved: the domain experts and the system analysts (base axioms from [Nij89]). First these base axioms are mentioned to explain the roles of the people involved in this modeling process. Then the way of working, way of modeling and finally the way of supporting are explained.

2.1.1 Domain expert

The domain expert is someone who knows the Universe of Discourse, but does not know the conceptual aspects of modeling that Universe of Discourse. In other words, a domain expert is someone with superior detail-knowledge and minor capabilities for abstraction.
The domain expert creates sample sentences of the system, which express possible events of the Universe of Discourse. Because he has a superior detail-knowledge of the Universe of Discourse, he is also able to manage these sample sentences.

Using natural language leads to the following base axioms describing the cognitive identity of domain experts:

[D-1] Domain experts can provide any number of significant sample sentences

[D-2] Domain experts can split sample sentences into elementary sentences

[D-3] Domain experts can reformulate sample sentences into an unifying format

[D-4] Domain experts can order the sample sentences according to the dynamics of the application domain

[D-5] Domain experts can validate a description of their application domain

[D-6] Domain experts can judge the significance of a sample sentence

Communicate the problem domain to the system analyst is the first step in the modeling process. Because a complete set of sample sentences is a too strong requirement in practice, it is weakened to any number of sample sentences, but with some specific requirements on those sentences. Elementary sentences are not splitable without loss of information, which is a prerequisite for conceptual modeling, and can only be judged with the superior detail-knowledge of the domain expert. Reformulating these sample sentences leads to detection of syntactical categories when a system analyst performs a grammatical analysis. Finally, by ordering the sample sentences, the dynamics of the Universe of Discourse are captured.

During the modeling process, the domain expert must be able to validate a description of his application domain and the significance of generated sample sentences, both provided by the system analyst.

2.1.2 System analyst

The system analyst is someone who does not know the Universe of Discourse, but knows the conceptual aspects of modeling that Universe of Discourse. In other words, a system analyst is someone with minor detail-knowledge and superior capabilities for abstraction.

Because he has superior capabilities for abstraction of the Universe of Discourse, the system analyst is able to structure sample sentences, provided by the domain expert, and manage the abstracted structures.

Using natural language leads to the following base axioms describing the cognitive identity of system analysts:
2.1 Modeling process

[A-1] System analysts can validate a set of sample sentences for consistency

[A-2] System analysts can perform a grammatical analysis on a set of sample sentences

[A-3] System analysts can abstract sentence structures from a set of related syntactical categories

[A-4] System analysts can generate new sample sentences

[A-5] System analysts can match sentence structures with concepts of a modeling technique

First the set of sample sentences provided by the domain expert is validated for consistency. Then grammatical analysis of sample sentences leads to detection of syntactical categories, from which the underlying sentence structure is derived. With these results the system analyst is able to generate new sample sentences. Finally, he must be able to match those results with concepts of a particular modeling technique.

2.1.3 Way of working

The way of working indicates the way in which the information system is developed, i.e. the overall strategy for the development of the information system. This strategy deals with the identification of tasks and subtasks in the development process and their feasible order. In figure 2.1 an overview of the way of working in the analysis phase is provided, which is based on use of natural language.

The initial specification, from which the required information system can be derived, is provided by domain experts. Reformulating this initial specification in an uniform way results in an informal specification, the input of the grammatical analysis performed by the system analysts. During the grammatical analysis significant syntactical categories and their relationships are discovered and described in the normal form specification.

Then the normal form specification has to be modeled and is matched to concepts of a modeling technique, which results in the abstraction of the object-oriented analysis models and a population. Interpretation of the object-oriented analysis models provides an information grammar, where interpretation of the population provides a lexicon.
Finally by paraphrasing, a textual description is produced from this grammar by the system analyst, which is used by the domain expert to validate the informal specification.

### 2.1.4 Way of modeling

The way of modeling describes the modeling concepts used in the method, the way the concepts are related, and the properties of these concepts. It structures the models which can be used in the information system development.

A system analyst must understand the problem domain precisely, in order to model a Universe of Discourse adequately. The conceptual modeling technique used by the system analyst has to provide the system analyst with the proper concepts. Frederiks introduces a new object-oriented modeling technique, also called PSM², which can be seen as an extension, with concepts for modeling dynamic aspects, of the Predicator Set Model (PSM), a conceptual data modeling technique (see Hof93).

Within this new object-oriented modeling technique, the two dimensions of a Universe of Discourse (static and dynamic) can both be represented with the analysis models described in section 1.3: the object action involvement model, the object property model and the object life model.

For a more detailed description of these three object-oriented analysis models I will refer to the work of Frederiks himself ([Fre97]).
2.2 Automated support

2.1.5 Way of supporting

The *way of supporting* of a method deals with all possible support for performing and facilitating systems development tasks, and can thus be defined as the collection of all possible tools. Tools are either automated (database management systems, code generators) or not automated (white-boards, paper and pencil). A generally accepted name for automated tools is CASE-tool.

To achieve the stage of validation of the description of their application domain by the domain experts (base axiom D-5), system analysts have to match sentence structures with concepts of a modeling technique (base axiom A-5) and generate new sample sentences (base axiom A-4).

For this, the required information grammar is provided by interpretation of the object-oriented analysis models (abstracting the information grammar). These can be used to produce of a textual description with sample sentences of these analysis models, or to obtain a parser from the information grammar which allows the ability to check whether sentences of the expert language are captured by the information grammar (communicating the information grammar).

The purpose of this thesis is to achieve paraphrasing the conceptual model, with abstracting and communicating the information grammar by automated support, implementing a CASE-tool. In the next section this CASE-tool is examined.

2.2 Automated support

Implementing a CASE-tool can support paraphrasing the conceptual model, using an information grammar derived from object-oriented analysis models. This CASE-tool is useful for the domain expert as well as the system analyst. It is also clear that the information grammar is the most important factor of this part of the modeling process. Now all this can be captured within two main tasks of such a CASE-tool.

- abstracting the information grammar
- communicating the information grammar

With these purposes, the CASE-tool as a database system has to be designed entirely, deriving its elements and their relations results in a system design [Van92]. The next step is to attribute functionality to the database system. As well as the main tasks, the CASE-tool consists of some common tasks, like opening or saving an analysis project. These tasks can be explained according their process models [Hof93]. Finally these models are used to implement a user interface and handle its events.

Designing and developing the two main activities of this thesis, automated abstraction and communication by the CASE-tool, will be explained in the sequel of this section.
2.2.1 Abstracting the information grammar

When the normal form specification has been modeled and matched to concepts of our object-oriented modeling technique, it results in the abstraction of the object-oriented analysis models. Before the information grammar can be abstracted with the tool for use in the communication, the structure of the analysis models, way of modeling, first has to be elaborated.

Input of the object-oriented analysis models from the way of modeling, requires a proper design according the following aspects:

- model the syntax of each object-oriented analysis model in PSM
- model the naming and verbalization of the object action involvement model and object property model in PSM
- transform PSM-models with complex components to APSM-models
- derive the internal representation from the APSM-models
- design process models for each model

The analysis models are described by Frederiks, according some formal aspects including the syntax, naming and verbalization. At this point only these three aspects are of any interest. The syntax of each analysis model is a formal description of the model in sets, functions and axioms and is modeled with the conceptual modeling technique PSM. The derived PSM-schemas are also called meta-models, which do not capture the formal semantics of an analysis model, only its syntactical aspects. Note, that the correctness of the axioms and constraints of the models, when they are put into the system, will not be considered in the scope of this thesis.

After modeling the syntax, the syntactical naming and verbalization aspects of the object action involvement model and object property model are modeled with PSM (the object life model does not have any naming or verbalization rules). These aspects make these two models, communicable by naming their elements and providing rules for verbalization of its structure.

Now the kind of data for the input of the object-oriented analysis models is conceptually modeled, the data has to be organized in a certain way, which is also called the internal representation. Because the obtained PSM-schemas are rather complex to derive the internal representation, they have to be transformed to APSM-schemas. Atomic Predicator Set Model (APSM) is a subset of PSM with only the atomic components. When only atomic components are used rather than complex components like generalization or objectification, the internal representation can be derived easily. In [Kal97] APSM is defined and a transformation from PSM to APSM is given, which is in fact flattening of complex components to atomic components.

The design of the input of the object-oriented analysis models is completed, by designing process models for each analysis model. These process models capture the dynamic aspects, such as actions and their execution order (see [Hof93]).
2.2 Automated support

With this design the input of the analysis models can be implemented with the Delphi programming environment (using Delphi is explained in section 2.5), according the following aspects:

- design a user-interface based on the process models
- create paradox tables using the internal representation with Delphi’s Database Desktop
- connect the user-interface to the paradox tables and handle the events

Now the algorithms for interpreting the analysis models into a corresponding information grammar [Fre97] have to be implemented, so the information grammar can be abstracted.

2.2.2 Communicating the information grammar

Beside the abstraction of an information grammar, modeling and matching of the normal form specification to concepts of our object-oriented modeling technique, it results in the abstraction of a population of the object-oriented analysis models. Before the lexicon can be abstracted with the tool for use in the communication, this population of the models first has to be stored, preparatory to the way of supporting.

Input of that population requires a proper design according the following aspects:

- model the population of the object action involvement model and object property model in PSM
- transform PSM-models with complex components to APSM-models
- derive the internal representation from the APSM-models
- expand process models for the object action involvement model and object property model

For object type from the analysis models, all aspects concerning the population of that object has to be elaborated. Through identification of object instances and the population structure of objects is worked towards an implementation of populating the analysis models. This identification deals with the question whether the instances of each object type can be denoted. The purposes of the transformation of PSM-schemas, derivation of the internal representation and the process models are already described in the previous subsection.

With this design the population of the analysis models can be implemented with the Delphi programming environment (using Delphi is explained in section 2.5), according the following aspects:

- expand the design of the user-interface based on the process models
- create paradox tables using the internal representation with Delphi’s Database Desktop
- connect the user-interface to the paradox tables and handle the events

Now the algorithms for interpreting the population into a lexicon [Fre97] have to be implemented. Then communicating the information grammar can be accomplished, by
designing and implementing generation (produce a textual description with sample sentences) and validation of sentences (check sentences of the expert language) according the information grammar.

2.3 Delphi programming

This section gives an overview of Delphi and the ideas behind Delphi programming for Windows 95 and why Delphi is used to implement the CASE-tool to support object-oriented modeling based on information grammars.

The first step in developing a Delphi application is to design the screens. These can be constructed easily with dragging and dropping of components. After designing the interface, Delphi can do anything like traditional programming languages. Components in Delphi will recognize events like mouse clicks, how the objects respond to them depends on the written code. This makes Delphi programming fundamentally different from conventional procedural-oriented programming.

Programs in conventional programming languages run from the top down. A Delphi program works completely different, it is a set of independent pieces of code that are activated by, and so respond to, only the events they have been told to recognize.

The programming code in Delphi that tells a program how to respond to events like mouse clicks begins inside what Delphi calls event procedures. An event procedure is a body of code that is only executed in response to an external event. In almost all cases, everything executable in a Delphi program is either in an event procedure or is used by an event procedure, to help the procedure carry out its job. Components are customized by changing their properties, and their behavior by using methods.

With the use of database components, the designed user-interface can be connected to any possible underlying database, e.g. the database used by the CASE-tool. With lookup features it is even possible to construct data with already existing data, so data errors to be handled, can be reduced to a minimum.

Other advantages provided by Delphi are sophisticated error handling of preventing end users from bombing an application, a fast compiler with clear error-messages, report creation and an extensive help system and manuals for quick reference.

Delphi is an easy to use and a time saving environment with many features to develop complex database applications, which are all arguments to use it developing this CASE-tool. After designing the user-interface from the process models it is connected to the designed table structure, then the event handling is added to complete the application.
2.4 Structured Query Language (SQL)

Structured Query Language (SQL), is the set of commands that is used to access data within the database. Application programs and tools often allow users to access the database without directly using SQL, but these applications in turn must use SQL when executing the user's request. The strengths of SQL benefit all ranges of users including application programmers, database administrators, management, and end users. This section provides background information on SQL used by the database system that will be designed and developed within this thesis.

Technically speaking, SQL is a data sublanguage. The purpose of SQL is to interface to a relational database such as Paradox, and all SQL statements are instructions to the database. In this it differs from general purpose programming languages like C and Basic. Among the features of SQL are the following:

- it processes sets of data as groups rather than as individual units;
- it provides automatic navigation to the data;
- it uses statements that are complex and powerful individually, and that therefore stand alone.

Essentially, SQL lets you work with data at the logical level, only being concerned with the implementation details when you want to manipulate them. For example, to retrieve a set of rows from a table, define a condition has to be defined, used to filter the rows. All rows satisfying the condition are retrieved in a single step and can be passed as a unit to the user, to another SQL statement, or to an application. SQL provides commands for a variety of tasks including:

- querying data;
- inserting, updating, and deleting rows in a table;
- creating, replacing, altering, and dropping objects;
- controlling access to the database and its objects;
- guaranteeing database consistency and integrity.

SQL unifies all of the above tasks in one consistent language. Because all major relational database management systems support SQL, you can transfer all skills you have gained with SQL from one database to another. In addition, since all programs written in SQL are portable, they can often be moved from one database to another with very little modification.

2.4.1 Embedded SQL

Embedded SQL refers to the use of standard SQL commands embedded within Delphi's programming environment. Embedded SQL is a collection of these commands:

- all SQL commands, such as SELECT and INSERT, available with SQL with interactive tools;
Problem analysis

- dynamic SQL execution commands, such as PREPARE and OPEN, which integrate the standard SQL commands with Delphi's programming environment.

Embedded SQL also includes extensions to some standard SQL commands. Embedded SQL is supported by a precompiler. The precompiler interprets embedded SQL statements and translates them into statements that can be understood by the procedural language compiler.

2.5 Summary and outlook

In this chapter the modeling process has been considered, which led to a description, how the CASE-tool should support object-oriented modeling based on information grammars in the most optimal way, and in which way it should be designed. Within this description, beside the main functions of such a tool, a distinction has been made between two main tasks of the CASE-tool: abstracting the information grammar and communicating the information grammar. Finally, an overview has been given of the Delphi programming environment and the ability of using SQL, which is used to develop the CASE-tool.

In chapter 3 the above mentioned aspects, as they are described in this chapter, are used as a guideline to design the CASE-tool. Implementation of the CASE-tool will not be considered until chapter 4.
Chapter 3
Design of automated support

Before implementing a CASE-tool, it requires a proper design, which is elaborated in this chapter. In section 3.1 the system as a whole is designed, deriving its elements and relations. Also common tasks of the system are attributed to the design using process models. Then the two main tasks of the CASE-tool are elaborated. In section 3.2, abstracting the information grammar is designed, and in section 3.3 communicating the information grammar. All this is based on chapter 2, which has been used as a guideline to design the CASE-tool.

3.1 System design
This CASE-tool is a system which uses as well a user database as a Data Dictionary / Directory (DD/D) database. Because Delphi does not allow tables within tables and to avoid very long tables with slow retrieval and update actions, a DD/D database is used to store definitions of tables.

![Figure 3.1: system design of the CASE-tool](image-url)
Using the application can be seen as making analyses. For each new analysis, the definitions of empty tables are used to create a user database for that analysis. So each analysis has its own user database stored on the physical drive. Now the user is able to manipulate the data stored in the user database of a selected analysis.

The tables derived from the design in the next two sections of abstracting and communicating the information grammar, are put into the DD/D database. When within these sections references are made to data access, e.g. the process models, always the user database of a certain analysis is meant.

After explaining the databases, functionality must be attributed to the application. This application is called *Square Support* for ease. In the sequel of this section the functionality of this application is considered according the constructed process models, with as well common tasks as the tasks depicted from the problem analysis in the previous chapter.

**Process models**

The application *Square Support* is decomposed into its parts in figure 3.2. A decision has to be made to execute one of the tasks *Analysis*, *View* or *Help*. Their subtasks are also included in the figure. Termination of the application is achieved when the subtask *Exit* of the task *Analysis* is executed.

![Figure 3.2: process model of the application](image)
3.1 System design

This process model is in fact the menu of the application with three submenus. The sub-tasks of the three submenus will be described now:

**Analysis menu**

- **New**
  A new analysis can be started, from which name, directory and comment will be stored. In the directory a sub-directory as user database is created with the name of the analysis, and also the required tables from the DD/D are created.

- **Open**
  A stored analysis can be opened.

- **Save as**
  An analysis can be saved with another name from which name, directory and comment will be stored. In the directory a sub-directory as user database is created with the name of the analysis, and also the required tables from the DD/D are created.

- **Close**
  The current analysis is closed

- **Exit**
  The application is terminated

**View menu**

- **Models**
  Input of the analysis models with population.

- **Information Grammar**
  Abstracting the information grammar based on Models.

- **Communication**
  Communicating the information grammar based on Models.

**Help menu**

- **About**
  Information about Square Support is showed.

The menu items **Analysis** and **Help** are divided into some common subtasks. The subtasks of menu item **View** are the most prominent tasks. As described in the previous chapter the most important tasks are abstracting and communicating the information grammar. Before the information grammar can be abstracted the object-oriented analysis models have to be put into the system. Also before the information grammar can be communicated these models have to be populated. This input and populating of these analysis models are now combined within the task **Models**, on behalf of the user-interface (and thus the user) later in this thesis.
The task Models depicted from figure 3.2, is decomposed into its parts in figure 3.3, where a decision has to be made, to choose the object action involvement model (OAIM), the object property model (OPM) or the object life model (OLM), or to return to the main menu.

3.2 Abstracting the information grammar

In this section the information grammar is abstracted, after input of the object action involvement model, object property model and the object life model, which are described and designed according some aspects Frederiks used, to give a formal description of each model.

3.2.1 Object action involvement model

In this subsection, the syntax, the naming and verbalization, the internal representation, and process model of the object action involvement model are described and designed, according the aspects mentioned in section 2.2.1.

Syntax

The object action involvement model is a formal structure $OAI$, which consists of the following basic aspects [Fre97]:

1. A set $\mathcal{O}$ of object types. This set consists of the set $\mathcal{E}$ of elementary object types ($\mathcal{E} \subseteq \mathcal{O}$) and the set $\mathcal{Q}$ of complex or composed object types ($\mathcal{Q} \subseteq \mathcal{O}$).

2. A set $\mathcal{P}$ of predications. Predicators are abstracted roles in actions. A predictor is characterized by $\chi(p) = r$, when the object playing a role is responsible for the action, and characterized by $\chi(p) = i$, when the object playing a role is involved in the action ($r, i \in \Gamma$).
3.2 Abstracting the information grammar

3. A set \( \mathcal{A} \) of action types, \( (\mathcal{A} \subseteq \mathcal{Q}) \). The function \( \text{Action}(p) \) denotes the association of an action type with an predicator \( p \), and the function \( \text{Actor}(p) \) denotes the association of an object type, being the actor of this predictor \( p \).

4. A set \( \mathcal{G} \) of group types \( (\mathcal{G} \subseteq \mathcal{Q}) \). Group types are sets of instances of its element type.

5. A set \( \mathcal{S} \) of sequence types \( (\mathcal{S} \subseteq \mathcal{Q}) \). Sequence types are ordered sets of instances of its element type.

6. The function \( \text{Elt} : \mathcal{G} \cup \mathcal{S} \rightarrow \mathcal{O}/\mathcal{L} \) is the relation of group types and sequence types with their element type.

7. A set \( \mathcal{M} \) of module types \( (\mathcal{M} \subseteq \mathcal{Q}) \). Module types are compositions of object types. The relation \( \text{compr} \subseteq \mathcal{M} \times \mathcal{O}/\mathcal{L} \) describes the decomposition of a module type into its components, \( x \text{ compr} y \) is the denotation of: \( y \) is part of module \( x \).

8. The relation \( \text{gen} \) is the generalization structure for object types, \( x \text{ gen} y \) is the denotation of: \( x \) is a generalization of \( y \).

9. The relation \( \text{spec} \) is the specialization structure for object types, \( x \text{ spec} y \) is the denotation of: \( x \) is a specialization of \( y \).

Figure 3.4 shows a PSM schema modeling of the syntax of the object action involvement model. In this schema \( \mathcal{O} \) and \( \mathcal{O}/\mathcal{L} \) are already distinguished to anticipate the next model, the object property model.

The set \( \mathcal{O}/\mathcal{L} \) is the disjoint union of \( \mathcal{Q} \) and \( \mathcal{E} \), which is modeled with subtypes \( \mathcal{Q} \) and \( \mathcal{E} \) of set \( \mathcal{Q}/\mathcal{L} \) and the total exclusion constraint between these subtypes. In this way also is indicated that \( \mathcal{Q} \) is the disjoint union of \( \mathcal{M}, \mathcal{A} \) and \( \mathcal{G} \cup \mathcal{S} \), and that \( \mathcal{G} \cup \mathcal{S} \) is the disjoint union of \( \mathcal{G} \) and \( \mathcal{S} \).

Power type \( \mathcal{A} \) has as element type \( \mathcal{P} \), modeling that action types are sets of predicators. Each predicator is characterized as responsible for or involved in an action, occurs in exactly one action and only has one actor.

The fact types \( \text{Gen} \) and \( \text{Spec} \) model the generalization and specialization relations. The exclusion constraints ensure that if \( a \text{ Spec} b \), then neither \( a \text{ Gen} b \) nor \( b \text{ Gen} a \). Fact type \( \text{Compr} \) consists of module types with their elements, where fact type \( \text{Elt} \) consists of group types or sequence type with their corresponding element type.

The exclusion constraints deal with the general and subtyping rules of this model and are disregarded in the more detailed elaboration in this thesis, because of the earlier mentioned assumption.
Design of automated support

Figure 3.4: syntax Object Action Involvement Model in PSM with specialization

Naming and verbalization

After modeling the syntax of the object action involvement model, the *naming* and verbalization has to be modeled. By naming the elements, the model is made communicable. Now active objects are distinguished by the set $I$, as the objects playing a role in some action type. This set $I$ is defined by $I = \mathcal{O} \setminus \mathcal{A} \cup \{x \mid x \in \mathcal{A} \cap \exists p[\text{Actor}(p) = x]\}$, the set of objects without the action types which are not objectified. The naming structure consists of the following functions:

1. The function $\text{INm}$ assigns active object type names to identify active object types.

2. The function $\text{ANm}$ assigns action type names to identify action types, with the notion that the names for active object types and action types are different.
3.2 Abstracting the information grammar

3. The function $\text{PN}_m$ assigns predicator names to identify predicators, with the notion that they are used to denote how an actor of a predicator is involved in an action.

4. The function $\text{IVerbs}$ assigns a verbalizing set for paraphrasing to each active object type.

5. The function $\text{AVerbs}$ assigns a verbalizing set for paraphrasing to each action type.

As well as these functions, modeled in figure 3.5, the function $\text{FN}_m$ is included for active object types, action types and predicators. This function assigns the family name to an element for unique denotation of this element, in this way elements in different modules can have the same name. In figure 3.5 the fact types $\text{IN}_m$, $\text{IVerbs}$, $\text{FN}_m$, $\text{AN}_m$, $\text{AVerbs}$ and $\text{PN}_m$, correspond to those functions, naming and verbalizing the active objects, actions and predicators. Note that this model is an extension of the model in figure 3.4.

As well as these functions, modeled in figure 3.5, the function $\text{FN}_m$ is included for active object types, action types and predicators. This function assigns the family name to an element for unique denotation of this element, in this way elements in different modules can have the same name. In figure 3.5 the fact types $\text{IN}_m$, $\text{IVerbs}$, $\text{FN}_m$, $\text{AN}_m$, $\text{AVerbs}$ and $\text{PN}_m$, correspond to those functions, naming and verbalizing the active objects, actions and predicators. Note that this model is an extension of the model in figure 3.4.

Figure 3.5: naming and verbalization in PSM

**Internal representation**

To derive an internal representation of the syntax of the object action involvement model, the PSM-schema in figure 3.4 is flattened and modeled in APSM [Kal97] as in figure 3.6. Within this subtype hierarchy, the subtypes are pushed into their supertypes, together with the fact types in which they play a role. Now the fact types with constraints as subtype defining rules indicate which instances of supertype $\mathcal{O}$ can play a role in the according fact type, and the former total exclusion constraints between subtypes expire. To be able to
distinguish the different objects with the subtype defining rules, a label type (type) is connected with $O$ through fact type $Id$. In this way each object can be identified.

Power type $A$ is also pushed into its supertypes and the implicit fact type becomes an ordinary fact type connected to $P$. The exclusion constraints in the PSM-schema remain in the APSM-schema.

In APSM a subrule constraint is used to indicate which instances can play a role within the predicator, which base was a subtype before the transformation. Now the subtype defining rules of the object action involvement model from figure 3.6 are, in SQL:

**Subrule ($M$)**

```sql
Select Id.object from Id
where Id.type = 'M'
```

**Subrule ($G \cup S$)**

```sql
Select Id.object from Id
where Id.type = 'G' or Id.type = 'S'
```

**Subrule ($E$)**

```sql
Select Id.object from Id
where Id.type = 'E'
```

**Subrule ($A$)**

```sql
Select Id.object from Id
where Id.type = 'A'
```
3.2 Abstracting the information grammar

\[
\text{Subrule } (\mathcal{O}/\mathcal{L}) = \ Select \ ld.object \ from \ ld \ where \ ld.type \neq \ 'L'.
\]

After normalization of the eight fact types from the APSM-schema, fact types \text{Id} and \text{Elt} are joined in the identification table and fact types \text{Actor}, \text{Ea} and \text{Char} are joined in the predicator table. The table structure for this model is:

**Identification table : Id**

<table>
<thead>
<tr>
<th>object</th>
<th>type</th>
<th>elt</th>
</tr>
</thead>
</table>

**Predicator table : Predicator**

<table>
<thead>
<tr>
<th>predicator</th>
<th>action</th>
<th>actor</th>
<th>char</th>
</tr>
</thead>
</table>

**Specialization table : Spec**

<table>
<thead>
<tr>
<th>elem. object</th>
<th>abstract object</th>
</tr>
</thead>
</table>

**Generalization table : Gen**

<table>
<thead>
<tr>
<th>elem. object</th>
<th>abstract object</th>
</tr>
</thead>
</table>

**Compression table : Compr**

<table>
<thead>
<tr>
<th>module</th>
<th>abstract object</th>
</tr>
</thead>
</table>

In an identification-record, each object is identified by a subset of \( \mathcal{O} \) in the required field \text{type}. The field \text{elt} assigns an element type from \( \mathcal{O}/\mathcal{L} \) to the object, if it is a group type or sequence type.

In a predicator-record, the field \text{action} points to the action type in which the predicator is active. The field \text{actor} consists of the actor of the predicator. The fact that a predicator must be either responsible or involved can be found in the field \text{char}. All fields in this record are required.

In a specialization-record, the required field \text{elementary object} consists of the elementary object, which is a specialization of an abstract object in the required field \text{abstract object}. The constraint expresses that an elementary object can be a specialization of one or more abstract objects and an abstract object can be specialized into one or more elementary objects.

In a generalization-record, the required field \text{elementary object} consists of the elementary object, which is generalized, by an abstract object in the required field \text{abstract object}. The constraint expresses that an elementary object can be generalized by one or more abstract objects and an abstract object can generalize one or more elementary objects.
In a compression-record an abstract object is assigned to a module. Both fields *abstract object* and *module* are required. The constraint expresses that an abstract object belongs to one or more modules and a module can be decomposed into one or more abstract objects.

The naming and verbalization from the PSM-schema in figure 3.5 is flattened and modeled in APSM as in figure 3.7.

The subtypes /G2C and /G24 are pushed into their supertype /G32, together with the fact types in which they play a role. Now the fact types with constraints as subtype defining rules indicate which instances of supertype /G32 can play a role in the according fact type. The label type (type), added to the model in figure 3.6, is used to distinguish the different objects with the subtype defining rules. Now the subtypes defining rules of the naming and verbalization are, in SQL:

```
Subrule (I) = Select id.object from id
             where id.object <> '/G24'
             or (id.object = '/G24' and id.object = predicator.actor)
             group by id.object

Subrule (A) = Select id.object from id
             where id.type = '/G24'

Subrule (I ∪ A) = Select id.object from id
```
3.2 Abstracting the information grammar

After normalization of the fact types from the APSM-schema, disregarding the fact types involved with family naming and predicator naming, fact types \(INm\) and \(ANm\) are joined with the earlier created identification table. Family naming and predicator naming are disregarded because they are derivable from input of naming and characterization. Fact types \(IVerbs\) and \(AVerbs\) are put respective into the \(IVerbs\) table and the \(AVerbs\) table. The table structure for this model is extended with:

Identification table: \(Id\)

<table>
<thead>
<tr>
<th>object</th>
<th>type</th>
<th>elt</th>
<th>(INm)</th>
<th>(ANm)</th>
</tr>
</thead>
</table>

Verbalizing active object table: \(IVerbs\)

<table>
<thead>
<tr>
<th>object</th>
<th>(IVerbs)</th>
</tr>
</thead>
</table>

Verbalizing action table: \(AVerbs\)

<table>
<thead>
<tr>
<th>action</th>
<th>(AVerbs)</th>
</tr>
</thead>
</table>

The identification-record is extended with the naming fields \(INm\) and \(ANm\), respective assigning a name to an active object type and an action type.

In an \(IVerbs\)-record a verbalization is assigned to an active object. Both fields \(IVerbs\) and \(object\) are required. The constraint expresses that a verbalization can belong to one or more active objects and an active object can have one or more verbalizations.

In a \(AVerbs\)-record a verbalization is assigned to an action. Both fields \(action\) and \(AVerbs\) are required. The constraint expresses that a verbalization can belong to one or more actions and an action can have one or more verbalizations.

**Process model**

The object action involvement model, represented by task \(OAIM\) depicted from figure 3.3, is decomposed into its parts in figure 3.8. A decision has to be made, to manipulate an item or to return to choose a model. The subtasks are described briefly.

Each of the tasks \(Elem.object\), \(Group\), \(Sequence\), \(Module\), \(Action\), \(Specialization\) and \(Generalization\) consists of the functions \(view\) (with search, list, prior, and next), \(add\), \(edit\) and \(delete\). The three functions \(add\), \(edit\) and \(delete\) also need a confirmation function. The subtasks also consists of these functions.

A subtask \(IVerbs\) is assigned to the tasks \(Elem.object\), \(Group\), \(Sequence\), \(Module\), and \(Action\). Starting from one object, the above-mentioned functions can be used on the \(I\)-verbalizations of that object. The tasks themselves use queries based on the earlier mentioned subrules.
The according verbalizations are selected by this query with the parameter \textit{object}:

\begin{verbatim}
QueryIVerbs = Select * from IVerbs where IVerbs.object = :object
\end{verbatim}

\textit{Compr} is a subtask of the task \textit{Module}, which is concerned with the according decompositions of a module type parameterized by \textit{module}:

\begin{verbatim}
QueryModuleCompr = Select * from Compr where Compr.module = :module
\end{verbatim}

The task \textit{Action} has also a subtask \textit{AVerbs}, which is constructed in the same way as \textit{QueryIVerbs} with parameter \textit{object}, and a subtask \textit{Predicator}, which is concerned with the predicators of an action, selected with parameter \textit{action}:

\begin{verbatim}
QueryAVerbs = Select * from AVerbs where AVerbs.object = :object
QueryActionPred = Select * from Predicator where Predicator.action = :action
\end{verbatim}

The task \textit{Specialization} and \textit{Generalization} are used in a different way and access other tables, selecting all specializations and generalizations:

\begin{verbatim}
QuerySpec = Select * from Spec
QueryGen = Select * from Gen
\end{verbatim}
3.2 Abstracting the information grammar

The dashed lines in figure 3.8 indicate the data interchange between the tasks and the underlying table structure with the tables Id, Compr, Predicator, Spec, Gen, IVerbs and AVerbs.

Within the tasks Group, Sequence, Gen, Spec and subtasks IVerbs, AVerbs, Predicator, Compr, lookup fields based on queries or tables are used when the functions add, edit or delete, are applied. This is an advantage for the user, because the models can be constructed for the biggest part and violating input is excluded. The queries could be extended later to apply the axioms of the object action involvement model and prevent more violating cases (see Appendix B for more details).

3.2.2 Object property model

In this subsection, the syntax, the naming and verbalization, the internal representation, and process model of the object property model are described and designed, according the aspects mentioned in section 2.2.1.

Syntax

The object property model requires the extension of the abstract object types from the object action involvement model, with the concrete object types, the set $L$ of label types ($L \subseteq \mathcal{O}$). With this, a strict separation between abstract object types and concrete object types is required for the object action involvement model:

i. Label types cannot be involved in subtyping hierarchies, i.e. $\text{spec} \cup \text{gen} \subseteq \mathcal{E} \times \mathcal{O}/L$.

ii. Label types cannot act as an element type, i.e. $\text{Compr}(x), \text{Elt}(x) \notin L$.

The object property model is a formal structure $\mathcal{OP}$, which consists of the following basic aspects [Fre97]:

1. A set $\mathcal{B}$ of properties. A property can be seen as the bridge between an object type and a label types. For this, each property $p$ is associated with a retrieval action $\text{Retrieve}(p)$ and an update action $\text{Update}(p)$.

2. A relation $\text{Triggers}$ between a triggering action type and the triggered update action type of a property. $\text{Triggers}(a, \text{Update}(p))$ is the denotation of: action type $a$ triggers update action type $\text{Update}(p)$.

3. A relation $\text{IsDen}$ between an object types and its accompanying properties. $\text{IsDen}(x, p)$ is the denotation of: property $p$ is a denotation of instances of object type $x$.

Figure 3.9 shows a PSM schema modeling of the syntax of the object property model. In this schema $\mathcal{O}$ and $\mathcal{O}/L$ are distinguished by the addition of the set $L$ of label types.
A property can be connected to one or more abstract objects in fact type Props, has an update and retrieve action, respective modeled with fact types Update and Retrieve. A property also can be used to denote instances of an object, modeled with fact type IsDen. The objectified fact type Update is triggered by an action type from the set $\mathcal{A}$, modeled with fact type Triggers.

**Naming and verbalization**

After modeling the syntax of the object property model, the *naming structure* has to be modeled. By naming the elements, the model is made communicable. Now also naming and verbalization for label types will be assumed, and thus the functions $\text{INm}$ and $\text{IVerbs}$ from the object action involvement model can be applied. The naming structure consists of the following functions:

1. The function $\text{BNm}$ assigns property names to identify properties
2. The function $\text{BVerbs}$ assigns a verbalizing set for paraphrasing to each property.

![Diagram of Object Property Model in PSM with specialization](image)

*Figure 3.9: syntax Object Property Model in PSM with specialization*

![Diagram of Naming and Verbalization in PSM](image)

*Figure 3.10: naming and verbalization in PSM*
3.2 Abstracting the information grammar

As well as these functions, modeled in figure 3.10, the function \( \text{FNm} \) is included for properties. This function assigns the *family name* to an element for unique denotation of this element, in this way elements in different modules can have the same name. In figure 3.10 the fact types \( \text{FNm}, \text{BNm} \) and \( \text{BVerbs} \), correspond to those functions, naming and verbalizing the properties. Note that this model is an extension of the model in figure 3.9.

Also the retrieval action types and update action types have names and verbalization rules. These are constructed according the same principles as the ordinary action types of the object action involvement model. Only the names and verbalization rules for retrieval action types and object action types can be generated (see [Fre97]).

Internal representation

To derive an internal representation of the syntax of the object property model, the PSM-schema in figure 3.9 is flattened and modeled in APSM [Kal97] as in figure 3.11.

![Figure 3.11: syntax Object Property Model in APSM](image)

Within this subtype hierarchy, subtype \( l \) and \( O/l \) are pushed into their supertype \( O \), together with the fact types in which \( O/l \) plays a role. Now the fact types with constraints as subtype defining rules indicate which instances of supertype \( O \) can play a role in the according fact type. To be able to distinguish the different objects with the subtype defining rules, label type \( (\text{type}) \) - the same label type as in the APSM-schema of the object action involvement model - is added to this schema.

In APSM a subrule constraint is used to indicate which instances can play a role within the predicator, which base was a subtype before the transformation. Now the subtype defining rules of the object property model from figure 3.11 are, in SQL:

\[
\text{Subrule (A)} = \quad \text{Select} \quad \text{ld.object from ld}
\text{where} \quad \text{ld.type = 'A'}
\]

\[
\text{Subrule (O/L)} = \quad \text{Select} \quad \text{ld.object from ld}
\text{where} \quad \text{ld.type <> 'L'}
\]
After normalization of the two fact types from the APSM-schema, fact types Retrieve and Update are joined in the property table. Fact type Id is the same as within the object action involvement model and has already been elaborated. The identification table from the object action involvement model is extended with fact type IsDen. The table structure for this model is:

Identification table : Id

<table>
<thead>
<tr>
<th>Object</th>
<th>Type</th>
<th>elt</th>
<th>INm</th>
<th>ANm</th>
<th>IsDen</th>
</tr>
</thead>
</table>

Property table : Property

<table>
<thead>
<tr>
<th>Property</th>
<th>Retrieve</th>
<th>Update</th>
<th>Trigger</th>
<th>BNm</th>
</tr>
</thead>
</table>

In an identification record a denotation property can be assigned to an abstract object where the required field IsDen is added. A denotation property can belong to one or more abstract objects.

In a property-record, one retrieve action, one update action and one trigger of the update action are assigned to a property. All fields are required.

The naming and verbalization of the label types, retrieval action types and update action types can be stored within the table structure of the object action involvement model. Storing naming and verbalization of retrieval action types and update action types is not even necessary, because that is derivable. By adding these items, the subtype defining rules of the naming and verbalization rules of the object action involvement model from figure 3.11 do not have to be changed.

After normalization of the fact types from the PSM-schema from figure 3.10, disregarding the fact types involved with family naming, fact type BNm is joined with the earlier created property table. Family naming is disregarded because it is derivable from the input of naming. Fact types BVerbs is put into the BVerbs table. The table structure for this model is extended with:

Property table : Property

<table>
<thead>
<tr>
<th>Property</th>
<th>Retrieve</th>
<th>Update</th>
<th>Trigger</th>
<th>BNm</th>
</tr>
</thead>
</table>

Verbalizing property table : BVerbs

<table>
<thead>
<tr>
<th>Property</th>
<th>BVerbs</th>
</tr>
</thead>
</table>

The property-record is extended with the required naming field BNm, assigning a name to a property.

In a BVerbs-record a verbalization is assigned to a property. Both fields BVerbs and property are required. The constraint expresses that a verbalization can belong to one or more properties and a property can have one or more verbalizations.
3.2 Abstracting the information grammar

**Process model**

The object property model, represented by task OPM depicted from figure 3.3, is decomposed into its parts in figure 3.12. A decision has to be made, to manipulate an item or to return to choose a model. The task Property requires that each of the tasks Update, Retrieve and Trigger are processed.

Each of the tasks Label, Denotation, and Property consists of the functions view (with search, list, prior, and next), add, edit and delete. The three functions add, edit and delete, also need a confirmation function.

![Figure 3.12: task structure of OPM](image)

A subtask IVerbs is assigned to the task Label. Starting from one object, the above-mentioned functions can be used on the I-verbalizations of that object. The task itself uses a query based on the earlier mentioned subrule for label types. The according verbalizations are selected by this query with the parameter object:

```
QueryIVerbs = Select * from IVerbs
             where IVerbs.object = :object
```

The task Denotation is used in a different way and has access to the identification table, selecting all abstract objects with their denotation property:

```
QueryIsDen = Select Id.object, Id.IsDen from Id
             where Id.type <> '/G2F'
```

The task Property has a subtask BVerbs, which is constructed in the same way as QueryIVerbs with parameter property:
Design of automated support

\[
\text{QueryBVerbs} = \text{Select} \ast \text{from BVerbs where BVerbs.property = :property}
\]

The dashed lines indicate the data interchange between the tasks and the underlying table structure with the tables Id, IVerbs, IsDen, Property, and BVerbs.

Within the tasks Denotation, Property and subtasks IVerbs, BVerbs, lookup fields based on queries or tables are used when the functions add, edit or delete, are applied. The queries could be extended later, to apply the axioms of the object property model and prevent more violating cases (see also Appendix B).

### 3.2.3 Object life model

In this subsection, the syntax, the internal representation, and process model of the object life model are described and designed, according the aspects mentioned in section 2.2.1.

**Syntax**

The *object life model* adds the generalized predicator to the object action involvement model and the object property model. The object life model is a formal structure $\mathcal{OL}$ consisting of the following basic aspects [Fre97]:

1. The set $\mathcal{C}$ of *generalized predictors or components*. This set consists of a set $\mathcal{P}$ of *predicators*, a set $\mathcal{D}$ of *surrogates or deputies*, a set $\mathcal{X}$ of *repetition predictors*, a set $\mathcal{Y}$ of *choice predictors* and a set $\mathcal{Z}$ of *merge predictors*.
   - The set $\mathcal{T}$ of *tasks or task predictors* consists of predictors and surrogates ($\mathcal{P} \subseteq \mathcal{T}$ and $\mathcal{D} \subseteq \mathcal{T}$). The repetition predictors, choice predictors and merge predictors together form a set of *structural components* or *structural predictors* ($\mathcal{C}\setminus\mathcal{T}$).

2. The function $\text{Pred} : \mathcal{D} \rightarrow \mathcal{P}$ assigns a predicator to its surrogate.

3. The function $\text{Succ} : \mathcal{C} \mapsto \mathcal{C}$ describes the sequential order of components. $\text{Succ}(c)$ is the denotation of: *successor* of component $c$.

4. The function $\text{Decomp} : \mathcal{C} \mapsto \mathcal{C}\setminus\mathcal{T}$ provides initial components which are an element of a structural component. $\text{Decomp}(c) = d$ is the denotation of: component $c$ is a *decomposition alternative* of structural component $d$.

5. The relation $\text{Connect} \subseteq \mathcal{C} \times \mathcal{C}$ shows both sequential order and decomposition.

6. The relation $\text{HasInit} \subseteq \mathcal{O} \times \mathcal{C}\setminus\mathcal{Z}$ describes the components which are the initialization of an object type.

Figure 3.13 shows a PSM schema modeling of the syntax of the object life model. The set $\mathcal{C}$ is the disjoint union of $\mathcal{T}$ and $\mathcal{C}\setminus\mathcal{T}$, which is modeled with subtypes $\mathcal{T}$ and $\mathcal{C}/\mathcal{T}$ of set $\mathcal{C}$.
and the total exclusion constraint between these subtypes. In this way also is indicated that \( T \) is the disjoint union of \( P \) and \( D \), and that \( C/T \) is the disjoint union of \( X \), \( Y \) and \( Z \). In addition, subtype \( C/Z \) indicates that only a component which is not a merge predicator can be an initialization component of an object.

![Figure 3.13: syntax Object Life Model in PSM with specialization](image)

Connections of components are modeled in the fact types \( \text{Decomp} \) and \( \text{Succ} \), where fact type \( \text{Pred} \) assigns a predicator to its surrogate. Finally, fact type \( \text{HasInit} \) contains the components which are the initialization of an object type.

The exclusion constraint of fact type \( \text{Succ} \), expresses that if \( \text{Succ}(a) = b \) then \( \text{Succ}(b) = a \) is excluded. The other exclusion and total role over several roles constraints show, that each component that is not equal to some initialization component is either involved in a decomposition relation or a successor relation.

The exclusion and total role over several roles constraints deal with the general rules of this model and are disregarded in the more detailed elaboration in this thesis, because of the earlier mentioned assumption.
Internal representation

To derive an internal representation of the syntax of the object action involvement model, the PSM-schema in figure 3.13 is flattened and modeled in APSM [Kal97] as in figure 3.14. Within this subtype hierarchy, the subtypes are pushed into their supertypes, together with the fact types in which they play a role. Now the fact types with constraints as subtype defining rules indicate which instances of supertype C can play a role in the according fact type, and the former total exclusion constraints between subtypes expire. To be able to distinguish the different components with the subtype defining rules, a label type (type) is connected with C through fact type Id. In this way each component can be identified.

The exclusion and total role over several roles constraints in the PSM-schema remain in the APSM-schema.

In APSM a subrule constraint is used to indicate which instances can play a role within the predicator, which base was a subtype before the transformation. Now the subtype defining rules of the object life model from figure 3.14 are, in SQL:

```
Subrule (P) = Select Id.object from Id
where Id.type = 'P'

Subrule (D) = Select Id.object from Id
where Id.type = 'D'

Subrule (C/T) = Select Id.object from Id
where Id.type <> 'P' and Id.type <> 'D'

Subrule (C/Z) = Select Id.object from Id
where Id.type <> 'Z'
```
3.2 Abstracting the information grammar

After normalization of the five fact types from the APSM-schema, fact types Id, Succ, Decomp, HasInit and Pred are all joined in the component table. The table structure for this model is:

Component table: Component

<table>
<thead>
<tr>
<th>component</th>
<th>type</th>
<th>successor</th>
<th>decomposition</th>
<th>initialization</th>
<th>deputy</th>
</tr>
</thead>
</table>

In a component-record, the field type is the type of an component. If the component has a successor, then that successor can be found in the field successor, when it is a decomposition of a structural component, then that structural component can be found in the field decomposition. The initialization field points to the abstract object which is initialized by this component. Finally, the deputy field consists of the predicator for which it is a surrogate.

Process model

The object life model, represented by task OLM depicted from figure 3.3, is decomposed into its parts in figure 3.15. A decision has to be made, to manipulate an item or to return to choose a model.

![OML Diagram](image)

Figure 3.15: task structure of OLM

Each of the tasks Deputy, Repetition, Choice, Merge, Initialization and Successor, consists of the functions view (with search, list, prior, and next), add, edit and delete. The three functions add, edit and delete, also need a confirmation function.

A subtask Decomp is assigned to the tasks Repetition, Choice and Merge. Starting from one structural component, the functions add and delete, can be used on the decomposition.
of that component. The tasks themselves use a query based on the earlier mentioned sub-rules for structural components. The according decomposition in components is selected by this query with the parameter component:

\[
\text{QueryDecomposition} = \text{Select component, decomposition from Component}
\]

\[
\quad \text{where Component.decomposition = :component}
\]

The tasks Deputy, Initialization and Successor use queries based on their subrules. The dashed lines indicate the data interchange between the tasks and the underlying table structure with the table Component.

Within the tasks Deputy, Initialization, Successor and subtask Decomp, lookup fields based on queries or tables are used when the functions add, edit or delete, are applied. The queries could be extended later to apply the axioms of the object life model and prevent more violating cases (see also Appendix B).

### 3.2.4 Information grammar

Frederiks introduces in his work a paraphrasing mechanism for the object action involvement model, the object property model and the object life model. By implementing the paraphrasing rules, and the paraphrasing structure procedures, the information grammar could be abstracted for use in communication, which will be described in the next section.
3.3 Communicating the information grammar

In this section all aspects concerning the communication of the information grammar are elaborated. Through identification of object instances and the population structure of objects is worked towards sentence generation and validation as described in section 2.2.2.

3.3.1 Identification of object instances

Before examining populating the analysis models, the identification of the objects from the object action involvement model has to be explained, where exists a difference between abstract instances and concrete values.

With the input of the object analysis models, the objects are first uniquely named e.g. by $A_1$, $a_2$, $A_3$, $S_1$ etc. After that a name, verbalizations and at least one property are assigned to an object. To improve readability, object instances are denoted with the concrete values of a label type, modeled as a property in the object property model. This occurs when the population is viewed, not where it is stored. Stored instances of objects consists of object identifiers or event identifiers.

All objects except label types, have instances of the set $\Omega$ of object identifiers. Action types do have instances of this set of object identifiers, only these are identifiers of the events. This is required to deal with instances when an action type is objectified, and thus plays roles in other action types. So each instance of an object is identified by a number.

Example 3.1

This is a sample population of object $A_1$ with object naming “Person”, object $A_2$ with object naming “Company”, and action $a_3$ with action naming “work at”:

$\text{Pop (}A_1\text{)} = \{1, 2\}$, denoted as $\{\text{Patrick, Michael}\}$
$\text{Pop (}A_2\text{)} = \{1, 2\}$, denoted as $\{\text{University of Nijmegen, Propago Software}\}$
$\text{Pop (}a_3\text{)} = \{\{p_1: 1, p_2: 1\}, \{p_1: 2, p_2: 2\}\}$

Then the verbalization of the action is as follows:

“Patrick works at University of Nijmegen”
“Michael works at Propago Software”

Because an instance number in a predicator has only one actor there are no misunderstandings about the denotation with a concrete value.
3.3.2 Population structure of objects

Now the identification of objects has been clarified, the population structure of objects can be explored. For each object its structure and internal representation is explained on the basis of PSM-schemes, sample populations and table structures. The PSM-schemes are considered as extensions of those of the object action involvement model and the object property model. Finally the process model of populating the analysis models is explained.

Label type

In contrast with other object, the instances of the population of label types are concrete values. Each label type can have several of those instances, and an instance could be part of the population of several label types.

![Figure 3.16: populating label types](image)

From figure 3.16, the following *label type table* can be derived easily, and added to the internal representation of the population structure:

Label type table : PopL

<table>
<thead>
<tr>
<th>label type</th>
<th>instance</th>
</tr>
</thead>
</table>

Elementary object type

The population of elementary object types vary from the population of label types, where the first population consists of object identifiers, and the second consists of concrete values. An elementary object type can have several instances as object identifiers, which is constructed in the same way for each elementary object.

![Figure 3.17: populating elementary object types](image)

The following *elementary object type table*, derived from figure 3.17, adds populating elementary object types to the internal representation of the population structure:

Elem. object type table : PopE

| elem. obj. type | instance |
3.3 Communicating the information grammar

**Group type**

Group types are sets of instances of its element type. The population or instances of a group type, also consists of object identifiers, and are connected with several instances from the population of its element type. In figure 3.18 the element type is not included, which is derivable from the object action involvement model, and it causes no problems with the population of PopG, contrary to the population of module types, because a group type is connected to one single element type.

![Figure 3.18: populating group types](image)

The extensional uniqueness constraint expresses that only one instance from <group type, instance> can be connected to a specific set of instances from <Elt instance>, otherwise a group type may consist of duplicate instances.

**Example 3.1**

A population violating the extensional uniqueness constraint of the PSM-schema in figure 3.18 is:

\[
\{ \{ p_1 : G_1, p_2 : #g_1, p_3 : #a_1 \}, \{ p_1 : G_1, p_2 : #g_1, p_3 : #a_2 \}, \\
\{ p_1 : G_1, p_2 : #g_2, p_3 : #a_1 \}, \{ p_1 : G_1, p_2 : #g_2, p_3 : #a_2 \} \}
\]

This population is violating because the instances #a_1 and #a_2 from the element type of group type G_1 are as well connected to instance #g_1 as instance #g_2 of group type G_1.

The **group type table** which comes with populating group types is:

<table>
<thead>
<tr>
<th>group type</th>
<th>instance</th>
<th>Elt instance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note, that for group types, the EU-constraint is not considered yet.
Sequence type

Sequence types are sets of instances of its element type with a particular order. The population or instances of a sequence type, also consists of object identifiers, and are connected with several instances from the population of its element type. In figure 3.19 the element type is not included, in accordance with the statement mentioned with the group types. For the population of a sequence type, a label type (I) is included, to state which position an element occurs in a sequence.

![Figure 3.19: populating sequence types](image)

The extensional uniqueness constraint expresses that only one instance from `<sequence type, instance>` can be connected to a specific set of instances from `<Elt instance, (I)>`, otherwise a sequence type may consist of duplicate instances.

**Example 3.2**

A population violating the extensional uniqueness constraint of the PSM-schema in figure 3.19 is:

\[
\{ \{ p_1 : S_1 , p_2 : \#s_1 , p_3 : \#a_3 , p_4 : 1 \}, \{ p_1 : S_1 , p_2 : \#s_1 , p_3 : \#a_4 , p_4 : 2 \}, \\
\{ p_1 : S_1 , p_2 : \#s_2 , p_3 : \#a_3 , p_4 : 1 \}, \{ p_1 : S_1 , p_2 : \#s_2 , p_3 : \#a_4 , p_4 : 2 \} \}
\]

This population is violating because the instances #a_3 and #a_4, respective numbered with 1 and 2, from the element type of sequence type S_1 are as well connected to instance #s_1 as instance #s_2 of sequence type S_1.

For sequence types the *sequence type table* is added to the internal representation of the population structure:

<table>
<thead>
<tr>
<th>sequence type</th>
<th>instance</th>
<th>Elt instance</th>
<th>(I)</th>
</tr>
</thead>
</table>

Note, that for sequence types, the EU-constraint is not considered yet.
3.3 Communicating the information grammar

Module type

Module types are compositions of object types, described by decomposition relations between a module type and its components. The population or instances of a module type, consists of object identifiers, and are connected with several instances from the population of one or more decomposed objects. In figure 3.20 the object type from the Compr relation is also included, to be able to distinguish the object identifiers for each decomposed object.

The extensional uniqueness constraint expresses that only one instance from \langle module, instance \rangle can be connected to a specific set of instances from \langle compr, compr instance \rangle, otherwise a module type may consist of duplicate instances.

Example 3.3

A population violating the extensional uniqueness constraint of the PSM-schema in figure 3.20 is:

\[
\{ \{ p_1 : M_1, p_2 : #m_1, p_3 : A_1, p_4 : #a_1 \}, \{ p_1 : M_1, p_2 : #m_1, p_3 : S_1, p_4 : #s_1 \}, \\
\{ p_1 : M_1, p_2 : #m_2, p_3 : A_1, p_4 : #a_1 \}, \{ p_1 : M_1, p_2 : #m_2, p_3 : S_1, p_4 : #s_2 \} \}
\]

This population is violating because the instances #a_1 and #s_1, respective instances from object A_1 and S_1, are as well connected to instance #m_1 as instance #m_2 of module type M_1.

All this leads to the construction of the module type table as shown below:

Module type table : PopM

<table>
<thead>
<tr>
<th>module type</th>
<th>instance</th>
<th>Compr object</th>
<th>Compr instance</th>
</tr>
</thead>
</table>

Note, that for module types, the EU-constraint is not considered yet.
**Action type**

Action types are populated with the notion of *events*, which is more complicated than populating previous objects. The population or instances of an action type, consists of events and associates, in figure 3.21 represented by fact type PopA, which connects an event several associates, and could connect an associate to several events. An event is an action at a certain time, represented by fact types Time and Action, and an associate is an object instance which plays the predicator role in that action, represented by fact types Predicator and Instance.

An event can occur as object instance in Instance, when the object is an objectified action type. Note, that the involved objects are derivable as actors of the predicators.

Uniqueness constraint expresses that a combination of a predicator and an instance is a unique one, and thus does not have to be captured by an associate more than once.

After normalization of the eight fact types from the schema in figure 3.21, the fact types Time and Action are joined in the *event table*, and the fact types Predicator and Instance are joined in the *associate table*. The *action type table* relates events and associates. These tables are added to the internal representation of the population structure for populating action types:
3.3 Communicating the information grammar

Event table: event

| event | time stamp | action |

Associate table: associate

| associate | predicator | object instance |

Action type table: PopA

| event | associate |

Note that, for action types, the uniqueness constraint from figure 3.21 is not considered yet.

**Process models**

Because the object structure is also used within the object action involvement model and object property model, the process models of these models are extended with those functionality and data access of the population.

The object action involvement model, represented by the task OAIM in figure 3.8, is extended with functionality and data access in figure 3.22.

To each of the tasks, Elem.object, Group, Sequence, Module and Action, a subtask Instances is assigned, to add and delete instances of an according object. Except the task Elem.object, all tasks use lookup fields to construct their instances (see also Appendix B).
The dashed lines in figure 3.22 indicate the added data interchange between the tasks and the underlying table structure with the tables PopE, PopG, PopS, PopM, PopA, Associate and Event.

The object property model, represented by the task OPM in figure 3.12, is extended with functionality and data access in figure 3.23.

![Figure 3.23: extension data access of OPM](image)

A subtask *Instances* is assigned to the task *Label*, to add and delete instances of an label type. The dashed lines in figure 3.23 indicate the added data interchange between the tasks and the underlying table structure with the table PopL.

### 3.3.3 Sentence generation and validation

Sentence generation and validation can be done when the information grammar is abstracted, which has to be done after the models are put into the database system, an analysis is loaded, or something within an analysis has been changed. Using the report function within Delphi generation and validation function can be easily connected to the information within the database system.
3.4 Summary and outlook

The design of the CASE-tool has been elaborated in this chapter, using the aspects of designing automated support considered in chapter 2. As well as the input of the object-oriented analysis models towards abstracting the information grammar, as populating these analysis models towards communicating the information grammar, has been designed.

In chapter 4 the implementation of the CASE-tool is considered, using the aspects designed in this chapter. The user-interface is constructed according the designed process models. After creating the tables, the user-interface is connected to these tables and the events are handled.

Chapter 5 considers the conclusions and further research of this thesis.
Chapter 4

Prototype: Square Support

According the design of automated support in chapter 3, the implementation of the CASE-tool is considered in this chapter. In section 4.1 the graphical user-interface is build using the designed process models. Section 4.2 considers creating tables and queries. Then in section 4.3 the user-interface is connected to the tables and the queries through database components and the events are handled, so the functionality of the user-interface is completed. This all is implemented with the Delphi programming environment.

4.1 From process models to graphical user-interface

This section shows how the graphical user-interface is constructed from the required functionality presented with the process models in chapter 3. The components on which this user-interface is created are called forms.

The TForm component is at the center of Delphi applications. An application is designed by putting other components on a form. Forms can be used as windows, dialog boxes, or simply as forms, such as data-entry forms. Figure 4.1 shows the forms, which are used in this CASE-tool, derived from the process model of the application in figure 3.2: the main form Square Support and the forms New, Open, Save as, Models, Information Grammar, Communication and About. The arrows indicate the activation and de-activation of the forms. To display a form that is not currently active in the application, the ShowModal method is called within this CASE-tool, and to close a form, the exit item from Border-Icons is used.

![Figure 4.1: forms used with the CASE-tool](image_url)
All of these forms are showed and explained in the sequel of this section, but the real data access, as it has been designed, is only added in section 4.3 when the tables and queries from section 4.2 are connected to the user-interface.

**Form: Square Support**

The main form of the CASE-tool is the *Square Support form*, showed in figure 4.2. This form is designed as a kind of toolbar, which can activate the forms concerning an analysis.

![Figure 4.2: graphical form Square Support](image)

A *menu* is designed with the TMainMenu component, which encapsulates a menu bar and its accompanying drop-down menus for a form. The items on the menu bar and in its drop-down menus are specified as designed with the process model of the application, which is decomposed into its parts in figure 3.2.

All items on the menu bar and in its dropdown menus, except the analysis-exit an help-about function, are also put on the form as TSpeed-Button components. *Speed buttons* are buttons that usually have graphical images on their faces that users use to execute commands or set modes. Different images are used to represent the different states of the speed buttons. For example, a folder with a document and a floppy disk is used to indicate the save-as function of an analysis.

When the application is started there is no analysis selected. Only the menu items and speed buttons with the analysis-new and open functions and the menu item help-about are enabled, the others menu items and speed buttons are disabled. When the analysis-new or open function is successfully executed, the other functions will be enabled, and the analysis-new and open functions disabled. After closing an analysis, the state of the menu items and speed buttons will be the same as when the application is started, the no analysis selected state.

Underneath the speed buttons some TLabel components are placed. A *label* is a nonwindowed control displaying text on a form that the user cannot edit. The labels on this main form display the name, directory and comment of the selected analysis.
Forms: New / Open / Save as

The subtasks New, Open and Save as of the task Analysis, depicted from figure 3.2, are executed through menu items or speed buttons from the main form as dialogue boxes. Dialogue boxes are on-screen controls that the system displays to provide contextual information. The dialogue boxes used with the functions analysis-new, open and save as, are modal, to force the user to respond to some question before executing the function, because all other controls are frozen. In this way the user is forced to take a decision to avoid an irreversible but possibly unwanted or dangerous situation.

In figure 4.3 the New form is showed. This dialogue box asks the user for an analysis name, a directory where to put the analysis and possibly some comment. A TDirectory-Listbox component is used, which enables the user to select a directory.

![New form](image)

*Figure 4.3: graphical form New*

The Save as form is constructed in the same way as the New form, showed in figure 4.3. The only difference between these forms is the functionality, which was explained with the process model of the application in figure 3.2.

The final dialogue box is the Open form, showed in figure 4.4. This dialogue box asks the user to select an analysis, by clicking in the list with stored analyses. On a click, the according name, directory and comment of the selected analysis are showed.

The required confirmation buttons are also put on the New, Open and Save as forms. After input of the contextual information, the function could be either continued, and finished by clicking the OK button, or canceled without changes by clicking the cancel button. Then the function is executed or aborted, and the form is de-activated. The final component, the help button, is intended to jump to the according information in a possibly help document.
Form: Models

By clicking the view-models menu item or speed button from the menu of the main form, the Models form is activated, which is also modal, and represents the subtask Models of task View from figure 3.2. This task Models is on its turn decomposed into its parts in the process model from figure 3.3, which are input and populating the:

- object action involvement model (task OAIM),
- object property model (task OPM),
- object life model (task OLM).

These three models are integrated within the Models form. Each model is activated on a TPageControl component, and only one model at a time can be activated (at first activation of the Models form that is the object action involvement model). A pagecontrol is a page set, which is used to make a multiple page dialogue box. It displays multiple overlapping tabsheets, which are TTabSheet components. The user selects a page by clicking the page's tab that appears at the top of the control.

Object action involvement model

Figure 4.5 shows the Models form where the pagecontrol of the object action involvement model is activated. This pagecontrol is the only difference with the other analysis models, because the menu and the speed buttons are the same for each analysis model. The menu items and speed buttons connected to operations on records, like prior, next, edit, add and delete, can be used with each selected pagecontrol and tabsheet. The speed buttons with the letters A (action), P (property), L (life), represent the three models from the menu item Model, clicking them will switch the pagecontrols.
4.1 From process models to graphical user-interface

Each tabsheet from the pagecontrol of the object action involvement model represents a task from the process model in figure 3.8. The view on the Models form in figure 4.5 represents the task Group. On the Group tabsheet, three panels are visible, a listbox, a search-box and an object panel with confirmation buttons. List- and searchboxes are always focussed on the main item of the object panel (also property or component panels occur in stead of object panel). In this case a list of group types is visible, and a group type can be searched for with the groupbox. All tabsheets of the three pagecontrols are organized in this way.

From the PSM-schemas of the syntax in figure 3.4, the naming and verbalization in figure 3.5, and the table structure, the required editable data components are derived and put on the object panel. A group type for example, consists of group type, one element and one name. But a group type can also be verbalized by several verbalizations. Therefore a subtask IVerbs was added to the task Group from the process model in figure 3.8.

Such subtasks are added to the user-interface with their own listbox, searchbox and object panel. In figure 4.5 a button is added to switch to the subtask IVerbs. After switching by clicking that button, the current listbox, searchbox and object panel disappear and those of

---

**Figure 4.5: graphical form Models – pagecontrol OAIM**
the subtask become visible. The object panel of the subtask contains, beside confirmation buttons, only one other button, to return to the main task of the tabsheet. The other button in this figure, is added to switch to the subtask Instances, which was added to the task Group from the process model in figure 3.22. In this way all tabsheets of the three pagecontrols are constructed.

Object property model

With the pagecontrol of object action involvement model some common aspects have been considered. With figure 4.6, which shows the Models form with an active pagecontrol of the object property model, some different aspects are considered.

![Object Property Model](image)

*Figure 4.6: graphical form Models – pagecontrol OPM*

Each tabsheet from the pagecontrol of the object property model represents a task from the process model in figure 3.12. The view on the Models form in figure 4.6 represents the task Property. The list- and searchbox of the property tabsheet are focussed on the property item.

One thing that attracts attention in this figure is that all but the confirmation buttons are disabled. This can be explained with the modes in which the tabsheets occur. These modes are the view, edit, add and delete mode. If one of the last three modes is selected, which is
in this figure the edit mode, the confirmation buttons are enabled and the speed and sub-
task buttons and the list- and search box are disabled, to avoid an unwanted or dangerous
situation. After possibly input of the contextual information, the function with the mode
could be either continued, and finished by clicking the OK button, or canceled without
changes by clicking the cancel button. Then the function is executed or aborted, and the
form is restored to the view mode. This is in fact a sort of dialogue box within a tabsheet.

Object life model

Figure 4.7 shows the Models form where the page control of the object life model is
activated. Each tabsheet from the pagecontrol of the object life model represents a task
from the process model in figure 3.15. The view on the Models form in figure 4.7
represents the task Deputy. The list- and searchbox of the property tabsheet are focussed
on the deputy item.

![Figure 4.7: graphical form Models – pagecontrol OLM](image)

The last aspect that will be considered is the difference between editboxes. In figure 4.7
the deputy item is showed within a TDBEdit component, and the predicator item within a
TDBLookupComboBox. The database editbox is a data-aware edit box with all the capa-
bilities of an ordinary edit box. Unlike an ordinary edit box, you can use the database edit
box to enter data into a field, or to simply display data from a field in a dataset. A lookup
combobox provides the user with a convenient drop-down list of lookup items for filling in fields that require data from another dataset. Also the listbox is not ordinary but a database listbox. Linking and configuring these editboxes will be explained in section 4.3.

**Form: Information Grammar / Communication**

The *Information Grammar form* and the *Communication form* can be seen as reports. These reports could be created with Delphi’s Report Smith, a visual database reporting and query tool. It provides a streamlined approach to creating reports using database files and tables. The input of the analysis can be used to report the abstraction of the information grammar, and the population of these models to report the textual description on base of this information grammar.

**Form: About**

Figure 4.8 shows the *About form*, which is the only help item in this prototype for now. This form shows information of the prototype in relation to this thesis.

![Figure 4.8: graphical form About](image)

Figure 4.8: graphical form About
4.2 Creating tables and queries

This section considers creating tables and queries based on their design in chapter 3 and how these tables and queries are used within the Delphi programming environment and in the application.

First of all the in chapter 3 derived tables are created as Paradox tables by defining fields, specifying field types, keys and secondary indexes with Delphi’s Database Desktop, and then saved in some directory which is used for the database. Appendix B gives an overview of these created Paradox tables.

Working with tables and queries requires usage of TDataModule components. A *data-module* is a specialized Delphi class for centralized location and handling of any non-visual component in an application. A data module can be used to place all data access components in a single visual container at design time instead of duplicating them on each application form or design tables and queries once for use with many forms instead of recreating them separately for each form.

Figure 4.9 shows the datamodules used in this CASE-tool. DMAnalysis is used to connect the table with common information about the analysis. The three other datamodules are related to the tables derived with design of input and population of the object action involvement model (DMAction), object property model (DMProperty) and object life model (DMLife).

![Figure 4.9: data modules used with the CASE-tool](image-url)
Now TTable and TQuery components are defined in the datamodules to connect the Paradox tables and be able to use the data within the Delphi programming environment and in the application.

A **table** component is the most fundamental and flexible dataset component class in Delphi. It gives access to every row and column in an underlying database table. A tabled component is designed by placing it in a data module and set the DatabaseName of the component to the name of the database to access and the TableName property to the name of the table in the database.

A **query** component encapsulates an SQL statement that returns a set of related rows and columns from one or more database tables. A query component differs from a table component in two significant ways. It can access more than one table at a time (called a "join" in SQL) and automatically access a subset of rows and columns in its underlying table(s), rather than always returning all rows and columns and requiring to set ranges and filters to restrict row access. The data returned by a query component depends on the specified selection criteria in the component’s SQL property at design time or at run time. A selection criterion can be as well static, where all parameters in the statement are specified at design time, or dynamic, where some or all of a statement’s parameters are set at run time. A query is created just like a table, but also a SQL statement has to be specified in the SQL property of the component, and optionally any parameters for the statement in the Params property.

On base of the process models and queries designed in chapter 3, tables and queries are created within the datamodules as described above.

### 4.3 Data access through database components

In this section the user-interface form section 4.1 is connected to the datamodules with tables and the queries from section 4.2, through database components. But first a short introduction is given to the use of database components.

Delphi provides two kinds of database components, **data access** components and **data control** components. Data access components simplify database access by encapsulating database source information, such as the database to connect to, the tables in that database to access, and specific field references within those tables. Examples of these data access components are TTable and TQuery, which have been considered in the section 4.2.

Data control components display database information in forms. They are like standard user-interface components, except that their contents can be derived from or passed to database tables. Examples of these data control components are TDBEdit, TDBLookupComboBox and TDBListBox, which have been considered in section 4.1.

Datasets such as TTable and TQuery, are not visible at run time, but provide applications their connection to data through Borland Database Engine (BDE). Data control compo-
4.3 Data access through database components

Components are attached to dataset components by a TDataSource component, to provide a visual interface to data.

Figure 4.10 shows how data access and data control components relate to data, to one another, and to the user-interface in a Delphi database application.

![Database Components Architecture](image)

**Figure 4.10: Database components architecture**

To be able to visualize the data with the data control components of the user-interface designed in section 4.1, TDataSource components are created for each table and query of the datamodules from section 4.2.

Then, to use a component from a data module in a form, the data module must be added to the form’s uses clause, so methods or properties of that component can be used in the form. For data access components, the DataSource property of each data-aware control (TDBEdit, TDBLookupComboBox, TDBListBox components) in the form must be set to point to the data source in the module. After that, the field in the dataset must be specified to access as the value of the DataField property. Figure 4.11 shows, which forms from section 4.1, are connected to which data modules from section 4.2.

Finally, the events handling data must be handled. These events are the functions executed by the menus and buttons, like prior, next, edit, add, delete and search. On the prior and next functions the Prior and Next method of a table or query must be called. On the edit and add function the Insert method must be called, and after confirmation with the OK button, the Post method to post a record to the table. Only after confirmation with the OK button the Delete method is called on the delete function. Whenever the cancel button is used, the Cancel method is called. The search function is not a standard function but had to be designed and implemented.
4.4 Summary and outlook

This chapter gave an overview from beginning to end, why and how has been acted with the implementation of the CASE-tool, according the design of automated support in the previous chapter. With explanation of the use of the Delphi programming environment and its components this chapter contributed to the description of the practical part of this thesis. The prototype of the CASE-tool is on the floppy disk belonging to this thesis. The contents of this disk are described in appendix C.
Chapter 5
Conclusions and further research

5.1 Conclusions

Developing towards a prototype of a CASE-tool to support the object-oriented modeling technique based on information grammars requires a precise consideration how such a tool has to be designed in the most optimal way. This depends on the sequence of developing restricted parts of the modeling technique and unfortunately also on (parts of) the used programming environment, which means that the possibilities of that programming environment must be well known.

Automated support

Designing automated support of the conceptual model in the analysis phase of this modeling technique is based on meta-models in PSM of the object action involvement model, object property model and object life model. After modeling these three analysis models, the meta-models are translated through APSM-models to an internal representation, the table structure.

Designing process models links the table structure with the functionality of the prototype, which strongly influences the design of the graphical user-interface. Using parameterized queries and lookup references to access the table structure, the idea of constructing the analysis models rises, in stead of just inserting records. This is one of the greatest advantages of using a CASE-tool.

Besides, concerning automated support of a modeling technique, especially when the technique is developed at the University of Nijmegen, is also interesting in some other ways:

- further research of a PhD thesis written at the University of Nijmegen;
- student teaching at the School of Computing Science.

I think I have added a great contribution towards both of these aspects.
Abstracting and communicating the information grammar

Implementation of verification and extension of the axioms mentioned in the work of Frederiks [Fre97] is required, to be able to make the information grammar completely communicable. The prototype for now is not suitable for adding the abstraction and communication, because of the minor performance of update and retrieval actions. The abstraction of the information grammar and the communication with it, can, as described earlier in chapter 3, easily be implemented using Delphi.

5.2 Further research

Redundancy

Within the tables of the population of the analysis models much of redundancy occurs. Update and retrieval suffer little from this redundancy when an analysis contains many objects, but time and especially space also suffer when an analysis contains only a few objects with some instances. Each instance of a group type, sequence type or module type for example has to be assigned to the same object, because of the non-unique instance numbering. One approach to improve this is to develop something like a unique code in which as well the object as an instance occurs. The most evolutionary approach to improve this problem is to design the database using SQL3. That approach allows assigning tables to fields, in other words a table within a table, which eliminates the redundancy. Using SQL3 also provides bigger efforts when implementing the verification, when constructing models, and extensions of the axioms mentioned in the previous section.

I would have used the tool InfoModeler 3, which uses SQL3 and database generation, with this thesis to design and extract the database, but I could not acquire a licensed version.

Field constructions

The question has raised, how to store predicators in action types and how to store verbalization. When a construction is used within a field this also requires a decomposition algorithm. Using SQL3, this construction can be replaced easily, with a constructed type within the current table structure.

Programming aspects

The following programming aspects are interesting for further research to improve the automated support of this prototype:

- automatic support assigning a label instance to an object instance identifier;
- automatic assigning object numbering when constructing new objects: L1, M1 etc;
- remove object in each table where it occurs through relations (joins).

Note that using SQL3 in stead of standard SQL, leads to a better and easier to develop improvement, because of the more dynamic table structure that can be assigned.
Appendix A

Graphical Notation

This appendix contains an overview of the graphical conventions of the various techniques in this thesis.

E is an entity type

L is a label type

E is an entity type identified by label type L

role/predicator

binary fact type

binary fact type represented as an object type

B is a power type with element type A

B is a sequence type with element type A

C is a schema type with object types A, B and f in its decomposition

A is a specialisation of B

A is a generalisation of B

Figure A.1: construction mechanisms of PSM
Appendix A
Graphical Notation

Figure A.2: graphical constraints in PSM

Figure A.3: task structure concepts (process models)
Appendix B
Paradox Tables

This appendix is intended to give an overview of all Paradox tables used in the CASE-tool within a certain task. For each table, the fields with type, size, key and requirement are listed. Whenever it is applied, secondary indexes and lookup fields are also listed.

Secondary indexes, which are unique, determine whether records can have duplicate values in the secondary index field or fields. If Unique is checked and two or more records have the same value in the secondary index field, the attempt to define the secondary index fails.

The lookup fields are used to:
- refer to another table to look up acceptable values for a field,
- automatically copy values from the lookup table to the edited table (automatic fill in),
- require that the entered values into a field exist in the first field of another table.

When a lookup table for a field is specified, the field can contain only values that exist in the first field of another specified table, the lookup table. The major advantage of table lookup is its ability to automatically enter correct values in a table.

### B.1 Input of analysis models

#### Tables

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Size</th>
<th>Key</th>
<th>Req.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>Type</td>
<td>A</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elt</td>
<td>A</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INm</td>
<td>A</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANm</td>
<td>A</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IsDen</td>
<td>A</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Size</th>
<th>Key</th>
<th>Req.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>Type</td>
<td>A</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Successor</td>
<td>A</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decomposition</td>
<td>A</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initialization</td>
<td>A</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deputy</td>
<td>A</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Appendix B

#### Paradox Tables

**Specialization table : Spec.db**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Size</th>
<th>Key</th>
<th>Req.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elementary</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>Abstract</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

**Generalization table : Gen.db**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Size</th>
<th>Key</th>
<th>Req.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elementary</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>Abstract</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

**Compression table : Compr.db**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Size</th>
<th>Key</th>
<th>Req.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>Abstract</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

**Iverbs table : Iverbs.db**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Size</th>
<th>Key</th>
<th>Req.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>IVerbs</td>
<td>A</td>
<td>30</td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

**AVerbs table : AVerbs.db**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Size</th>
<th>Key</th>
<th>Req.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>AVerbs</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

**BVerbs table : Bverbs.db**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Size</th>
<th>Key</th>
<th>Req.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>BVerbs</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

**Property table : Property.db**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Size</th>
<th>Key</th>
<th>Req.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>Retrieve</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>Trigger</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>BNm</td>
<td>A</td>
<td>30</td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

**Predicator table : Predicator.db**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Size</th>
<th>Key</th>
<th>Req.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicator</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>Action</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>Actor</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>Char</td>
<td>a</td>
<td>1</td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

**Lookup fields**

<table>
<thead>
<tr>
<th>Table</th>
<th>Field</th>
<th>Lookupfield</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVerbs table</td>
<td>Object</td>
<td>QueryAction.object</td>
</tr>
<tr>
<td>BVerbs table</td>
<td>Object</td>
<td>QueryProperty.property</td>
</tr>
<tr>
<td>Component table</td>
<td>Successor</td>
<td>QuerySucc.component</td>
</tr>
<tr>
<td></td>
<td>Decomposition</td>
<td>QueryDecomp.component</td>
</tr>
<tr>
<td></td>
<td>Initialization</td>
<td>TableObject.object</td>
</tr>
<tr>
<td></td>
<td>Deputy</td>
<td>TablePredicator.predicate</td>
</tr>
<tr>
<td>Compression table</td>
<td>Module</td>
<td>QueryModule.object</td>
</tr>
<tr>
<td></td>
<td>Abstract</td>
<td>QueryAbstract.object</td>
</tr>
<tr>
<td>Generalization table</td>
<td>Elementary</td>
<td>QueryElementary.object</td>
</tr>
<tr>
<td></td>
<td>Abstract</td>
<td>QueryAbstract.object</td>
</tr>
<tr>
<td>IVerbs table</td>
<td>Object</td>
<td>QueryElementary.object v QueryGroup.object v QueryModule.object, QuerySequence.object v QueryAction.object v QueryLabel.object</td>
</tr>
<tr>
<td>Object table</td>
<td>Elt</td>
<td>QueryAbstract.object</td>
</tr>
<tr>
<td></td>
<td>IdDen</td>
<td>QueryProperty.property</td>
</tr>
<tr>
<td>Predicator table</td>
<td>Action</td>
<td>QueryAction.object</td>
</tr>
<tr>
<td></td>
<td>Actor</td>
<td>TableObject.object</td>
</tr>
<tr>
<td>Property table</td>
<td>Retrieve</td>
<td>QueryAction.object</td>
</tr>
<tr>
<td></td>
<td>Update</td>
<td>QueryAction.object</td>
</tr>
<tr>
<td></td>
<td>Trigger</td>
<td>QueryAction.object</td>
</tr>
<tr>
<td>Specialization table</td>
<td>Elementary</td>
<td>QueryElementary.object</td>
</tr>
<tr>
<td></td>
<td>Abstract</td>
<td>QueryAbstract.object</td>
</tr>
</tbody>
</table>
Note: field object of the IVerbs table uses a lookup field, depending on the object, which is involved in the process at a certain time.

### Secondary indexes

<table>
<thead>
<tr>
<th>Table</th>
<th>Index type</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVerbs table</td>
<td>Unique</td>
<td>&lt;Object, AVerbs&gt;</td>
</tr>
<tr>
<td>BVerbs table</td>
<td>Unique</td>
<td>&lt;Property, BVerbs&gt;</td>
</tr>
<tr>
<td>Component table</td>
<td>Unique</td>
<td>&lt;Successor&gt;</td>
</tr>
<tr>
<td>Compression table</td>
<td>Unique</td>
<td>&lt;Module, Abstract&gt;</td>
</tr>
<tr>
<td>Generalization table</td>
<td>Unique</td>
<td>&lt;Elementary, Abstract&gt;</td>
</tr>
<tr>
<td>IVerbs table</td>
<td>Unique</td>
<td>&lt;Object, IVerbs&gt;</td>
</tr>
<tr>
<td>Specialization table</td>
<td>Unique</td>
<td>&lt;Elementary, Abstract&gt;</td>
</tr>
</tbody>
</table>

### B.2 Populating the analysis models

#### Tables

**Label type table**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Size</th>
<th>Key</th>
<th>Req.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label type</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>Instance</td>
<td>A</td>
<td>30</td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

**Module type table**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Size</th>
<th>Key</th>
<th>Req.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module type</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>Instance</td>
<td>I</td>
<td></td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>Compr object</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>Compr instance</td>
<td>I</td>
<td></td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

**Group type table**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Size</th>
<th>Key</th>
<th>Req.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group type</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>Instance</td>
<td>I</td>
<td></td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>Elt instance</td>
<td>I</td>
<td></td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

**Associate table**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Size</th>
<th>Key</th>
<th>Req.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Associate</td>
<td>I</td>
<td></td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>Predicator</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>Instance</td>
<td>I</td>
<td></td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

**Event table**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Size</th>
<th>Key</th>
<th>Req.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>I</td>
<td></td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>Time stamp</td>
<td>@</td>
<td></td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

**Sequence type table**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Size</th>
<th>Key</th>
<th>Req.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence type</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>Instance</td>
<td>I</td>
<td></td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>Elt instance</td>
<td>I</td>
<td></td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>(I)</td>
<td>S</td>
<td></td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

**Elementary type table**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Size</th>
<th>Key</th>
<th>Req.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elem. obj. type</td>
<td>A</td>
<td>10</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>Instance</td>
<td>I</td>
<td></td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

**Action type table**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Size</th>
<th>Key</th>
<th>Req.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>I</td>
<td></td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>Associate</td>
<td>I</td>
<td></td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>
### Lookup fields

<table>
<thead>
<tr>
<th>Table</th>
<th>Field</th>
<th>Lookupfield</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action table</td>
<td>Event</td>
<td>QueryEvent.event</td>
</tr>
<tr>
<td></td>
<td>Associate</td>
<td>QueryAssociate.associate</td>
</tr>
<tr>
<td>Associate table</td>
<td>Predicator</td>
<td>QueryPredicator.predicator</td>
</tr>
<tr>
<td></td>
<td>Instance</td>
<td>QueryObjectInstance.instance</td>
</tr>
<tr>
<td>Elementary type table</td>
<td>Elem. obj. type</td>
<td>QueryElementary.object</td>
</tr>
<tr>
<td>Event table</td>
<td>Action</td>
<td>QueryAction.object</td>
</tr>
<tr>
<td>Group type table</td>
<td>Group type</td>
<td>QueryGroup.object</td>
</tr>
<tr>
<td></td>
<td>Elt instance</td>
<td>QueryGroup.elt</td>
</tr>
<tr>
<td>Label type table</td>
<td>Label type</td>
<td>QueryLabel.object</td>
</tr>
<tr>
<td>Module type table</td>
<td>Module type</td>
<td>QueryModule.object</td>
</tr>
<tr>
<td></td>
<td>Compr object</td>
<td>QueryCompr.abstract</td>
</tr>
<tr>
<td></td>
<td>Compr instance</td>
<td>QueryObjectInstance.instance</td>
</tr>
<tr>
<td>Sequence type table</td>
<td>Sequence type</td>
<td>QuerySequence.object</td>
</tr>
<tr>
<td></td>
<td>Elt instance</td>
<td>QuerySequence.elt</td>
</tr>
</tbody>
</table>

### Secondary indexes

<table>
<thead>
<tr>
<th>Table</th>
<th>Index type</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action type table</td>
<td>Unique</td>
<td>&lt;Event, Associate&gt;</td>
</tr>
<tr>
<td>Elementary type table</td>
<td>Unique</td>
<td>&lt;Elem. obj. type, Instance&gt;</td>
</tr>
<tr>
<td>Group type table</td>
<td>Unique</td>
<td>&lt;Group type, Instance, Elt instance&gt;</td>
</tr>
<tr>
<td>Label type table</td>
<td>Unique</td>
<td>&lt;Label type, Instance&gt;</td>
</tr>
<tr>
<td>Module type table</td>
<td>Unique</td>
<td>&lt;Module type, Instance, Compr object, Compr instance&gt;</td>
</tr>
<tr>
<td>Sequence type table</td>
<td>Unique</td>
<td>&lt;Sequence type, Instance, Elt instance, (I)&gt;</td>
</tr>
</tbody>
</table>
# Appendix C

## Program files

A:\ Bin

<table>
<thead>
<tr>
<th>Lib</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tables</strong></td>
<td><strong>Paradox table</strong></td>
</tr>
<tr>
<td>Associate</td>
<td>Paradox table</td>
</tr>
<tr>
<td>AVerbs</td>
<td>Paradox table</td>
</tr>
<tr>
<td>BVerbs</td>
<td>Paradox table</td>
</tr>
<tr>
<td>Component</td>
<td>Paradox table</td>
</tr>
<tr>
<td>Compr</td>
<td>Paradox table</td>
</tr>
<tr>
<td>Event</td>
<td>Paradox table</td>
</tr>
<tr>
<td>Gen</td>
<td>Paradox table</td>
</tr>
<tr>
<td>Id</td>
<td>Paradox table</td>
</tr>
<tr>
<td>IVerbs</td>
<td>Paradox table</td>
</tr>
<tr>
<td>PopA</td>
<td>Paradox table</td>
</tr>
<tr>
<td>PopE</td>
<td>Paradox table</td>
</tr>
<tr>
<td>PopG</td>
<td>Paradox table</td>
</tr>
<tr>
<td>PopL</td>
<td>Paradox table</td>
</tr>
<tr>
<td>PopM</td>
<td>Paradox table</td>
</tr>
<tr>
<td>PopS</td>
<td>Paradox table</td>
</tr>
<tr>
<td>Predicator</td>
<td>Paradox table</td>
</tr>
<tr>
<td>Property</td>
<td>Paradox table</td>
</tr>
<tr>
<td>Spec</td>
<td>Paradox table</td>
</tr>
</tbody>
</table>

**Analysis**

| Paradox table |

**Square**

| Application |

69
<table>
<thead>
<tr>
<th>A:\ Src</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>About</td>
<td>Delphi Form File</td>
</tr>
<tr>
<td>About</td>
<td>Delphi Pascal Unit</td>
</tr>
<tr>
<td>About</td>
<td>Delphi Compiled Unit</td>
</tr>
<tr>
<td>DMAction</td>
<td>Delphi Form File</td>
</tr>
<tr>
<td>DMAction</td>
<td>Delphi Pascal Unit</td>
</tr>
<tr>
<td>DMAnalysis</td>
<td>Delphi Compiled Unit</td>
</tr>
<tr>
<td>DMAnalysis</td>
<td>Delphi Form File</td>
</tr>
<tr>
<td>DMAnalysis</td>
<td>Delphi Pascal Unit</td>
</tr>
<tr>
<td>DMLife</td>
<td>Delphi Compiled Unit</td>
</tr>
<tr>
<td>DMLife</td>
<td>Delphi Form File</td>
</tr>
<tr>
<td>DMLife</td>
<td>Delphi Pascal Unit</td>
</tr>
<tr>
<td>DMPROPERTY</td>
<td>Delphi Compiled Unit</td>
</tr>
<tr>
<td>DMPROPERTY</td>
<td>Delphi Form File</td>
</tr>
<tr>
<td>DMPROPERTY</td>
<td>Delphi Pascal Unit</td>
</tr>
<tr>
<td>Main</td>
<td>Delphi Compiled Unit</td>
</tr>
<tr>
<td>Main</td>
<td>Delphi Form File</td>
</tr>
<tr>
<td>Main</td>
<td>Delphi Pascal Unit</td>
</tr>
<tr>
<td>Models</td>
<td>Delphi Compiled Unit</td>
</tr>
<tr>
<td>Models</td>
<td>Delphi Form File</td>
</tr>
<tr>
<td>Models</td>
<td>Delphi Pascal Unit</td>
</tr>
<tr>
<td>New</td>
<td>Delphi Compiled Unit</td>
</tr>
<tr>
<td>New</td>
<td>Delphi Form File</td>
</tr>
<tr>
<td>New</td>
<td>Delphi Pascal Unit</td>
</tr>
<tr>
<td>Open</td>
<td>Delphi Compiled Unit</td>
</tr>
<tr>
<td>Open</td>
<td>Delphi Form File</td>
</tr>
<tr>
<td>Open</td>
<td>Delphi Pascal Unit</td>
</tr>
<tr>
<td>Save</td>
<td>Delphi Compiled Unit</td>
</tr>
<tr>
<td>Save</td>
<td>Delphi Form File</td>
</tr>
<tr>
<td>Save</td>
<td>Delphi Pascal Unit</td>
</tr>
<tr>
<td>Square</td>
<td>DOF File</td>
</tr>
<tr>
<td>Square</td>
<td>Delphi Project File</td>
</tr>
<tr>
<td>Square</td>
<td>DSK File</td>
</tr>
<tr>
<td>Square</td>
<td>Application</td>
</tr>
<tr>
<td>Square</td>
<td>Intermediate File</td>
</tr>
</tbody>
</table>
Bibliography

[Bom95] P. van Bommel. 


*Paraphrasing as a Technique to Support Object-Oriented Analysis.* Technical report csi-r9603, Computing Science Institute, University of Nijmegen, The Netherlands, January 1996.


*From a File-Oriented View to an Object-Oriented View.* Technical report csi-r9601, Computing Science Institute, University of Nijmegen, The Netherlands, January 1996.

*Grammar Based Information Modeling*. Technical report csi-r9414, 
Computing Science Institute, University of Nijmegen, The Netherlands, 
October 1994.

*Schema transformaties; met als doel een interne representatie*. Master’s 

An Axiom and Architecture for Information Systems. In E.D. Falkenberg 
and P. Lindgreen, editors, *Information Systems Concept: An Indepth 
Analysis*, pages 157-175. North-Holland/IFIP, Amsterdam, The 
Netherlands, 1989.

[Van92]  Prof. Dr. J.A. Vandenbulcke  
*Databasesystemen voor de praktijk*. Kluwer Bedrijfs- 
wetenschappen, 
Index

A
action type ................................. 21
population .................................. 44
retrieval ...................................... 29
update .......................................... 29
analysis phase ............................. 29
APSM ........................... See modeling technique
automated support ...................... 11
B
bridge ......................................... 29
C
component ..................................... 34
decomposition alternative .......... 34
structural .................................. 34
successor .................................. 34
D
database
  Data Dictionary / Directory ........ 17
  user ........................................... 17
Delphi programming .................. 14
database editbox ....................... 55
datamodule ......................... 57
dialogue box ......................... 51
form ........................................... 49
lookup combobox ....................... 56
menu .......................................... 50
pagecontrol .................................. 52
speed button .................................. 50
tabsheet ..................................... 52
domain expert ............................. 7
E
elementary object type
  population ......................... 13, 39
  informal specification .......... 9
  information grammar ...... 2, 9
  abstraction ......................... 12
  communication .................. 13
  initial specification .......... 9
  internal representation ...... 12
I
identification ....................... 13, 39
informal specification ............. 9
information grammar ............. 2, 9
abstraction ....................... 12
communication ................ 13
initial specification .......... 9
internal representation ...... 12
L
label type ................................. 29
  population ....................... 13, 40
  lexicon .................................. 9
lookup field ....................... 29, 34, 38, 65
M
modeling .............................. 9, 12, 13
modeling process ................... 7
modeling technique
  APSM ..................................... 12
  PSM ....................................... 10
  PSM\(^2\) .................................. 10
module type ........................... 21
  population ....................... 43
N
naming .................................. 12
natural language .................... 2
normal form specification .......... 9
O
object
  identification ....................... 13, 39
  naming .................................. 12
  population .......................... 13, 40
  verbalization ...................... 12
object action involvement model
  internal representation .... 3, 10, 20
  naming and verbalization .... 23
  population ....................... 22
  representation ................. 23
  naming and verbalization .... 22
  population ....................... 40
  process model ................ 27, 45
  syntax ............................. 20
  user-interface .................. 52
object life model .................. 3, 10, 34
Index

internal representation ........................................... 36
process model ...................................................... 37
syntax ........................................................... 34
user-interface ..................................................... 55
object property model ........................................... 3, 10, 29
internal representation ......................................... 31
naming and verbalization ......................................... 30
population .......................................................... 40
process model ...................................................... 33, 46
syntax ............................................................... 29
user-interface ...................................................... 54
object type .......................................................... 20
abstract ............................................................ 29
complex ............................................................. 20
composed ............................................................ 20
concrete ............................................................. 29
denotation .......................................................... 29
object-oriented ...................................................... 3
object-oriented analysis model ................................... 3, 9, 12, 13
internal representation ........................................... 12
population .......................................................... 13
syntax ................................................................. 12

surrogate .......................................................... 34
task ................................................................. 34
problem area ....................................................... 1
problem statement ............................................... 3
process model ...................................................... 12
property ............................................................ 29
denotation .......................................................... 29
PSM .................................. See modeling technique
PSM² .................................. See modeling technique

S

secondary index ................................................... 65
sequence type ..................................................... 21
population .......................................................... 42
specialization ..................................................... 21
SQL ................................................................. 15
syntax ............................................................... 12
system analyst ..................................................... 8

textual description ............................................. 10

U

user-interface ......................................................... 13, 49

V

verbalization ....................................................... 12

W

way of working .................................................... 9
way of modeling ................................................... 10, 12
way of supporting ............................................... 11