Explanation Generation in Information Systems Engineering

Jon Atle Gulla

Information Systems Group
Faculty of Electrical Engineering and Computer Science
The Norwegian Institute of Technology
University of Trondheim
Abstract

In information systems engineering, conceptual models are constructed as part of the problem analysis and requirements determination process. The importance of formal conceptual modeling languages has been stressed, though the languages — which are given detailed syntactic and semantic definitions — can often lead to models that are quite complex and difficult to interpret. Yet, user participation in conceptual modeling is encouraged, and powerful strategies for enhancing the comprehensibility of conceptual modeling are then needed. Although many strategies are already available, like model execution and complexity reduction, it is still problematic for unskilled users to take part in modeling processes and fully understand the models constructed.

In this thesis, we suggest to build and integrate an explanation component into the conceptual modeling environment. This component is utilized by the users of the environment, answering questions related to the modeling language, the conceptual model, and the results from analyzing or executing conceptual models. In our new environment, the explanation component generates explanatory texts — possibly supplemented with graphical views of the conceptual model — that are intended to enhance the comprehensibility of formal conceptual modeling. These texts are given in a language with which the user is familiar, they provide a different point of view, and they abstract away details not relevant to the reader.

The explanation generation process is divided into two subprocesses, deep explanation generation and surface explanation generation. The deep generation process determines the explanations' content, structure, and presentation media, and is the focus of this work. In surface explanation generation, the results from the deep generation process are realized and displayed as complete explanations. For our deep generation part, we develop some principles and a formalism for generating user-tailored and context-tailored coherent deep explanations. The principles make it possible to explain properties of meta models, conceptual models, and execution traces in a uniform manner, and also to construct a generic component for deep explanation generation that is independent of any modeling language.

A component realizing the deep explanation generation process is constructed and tried out on the PPP CASE environment's modeling language. We show how the desired explanations are generated and also how the explanations are tailored to both user and context. At last, we indicate how the component can be extended to form a complete integrated explanation component for CASE environments.
Preface

This thesis is submitted to the Norwegian Institute of Technology for the doctoral degree “doktor ingeniør”. The work reported here has for the most part been carried out at the Information Systems Group, Faculty of Electrical Engineering and Computer Science, the Norwegian Institute of Technology, The University of Trondheim, Norway, under the supervision of Professor Arne Sølvberg. I have also had a shorter research stay at the Computer Science Engineering Department, the University of Texas at Arlington, USA, from February to August 1992.

In the spring term of 1990, I was employed as a researcher by SINTEF. From July 1990 to July 1993 I have had a scholarship from the Norwegian Research Council for Science and the Humanities (NAVF).

Acknowledgements

This work would not have been possible without the support and encouragement of a number of people. I would in particular like to thank the following people:

- Professor Arne Sølvberg for his continuous encouragement and constructive criticism, and for all that he has taught me during my stay here at the Norwegian Institute of Technology.

- All my colleagues at the Information Systems Group for an inspiring working environment and a nice social atmosphere. Odd Ivar Lindland and Geir Willumsen deserve some special thanks for the many interesting discussions on CASE environments and model validation.

- Professor David Chenho Kung for giving me the opportunity to come to the University of Texas at Arlington.

- My brother Bjørn Gulla for introducing me to the wonders — and intricacies — of computer science.

Trondheim, September 1993

Jon Atle Gulla
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td></td>
<td>ix</td>
</tr>
<tr>
<td>List of Tables</td>
<td></td>
<td>xiii</td>
</tr>
</tbody>
</table>

## I Introduction and Motivation

### 1 Introduction

1.1 Problem Description and Assumptions | 3 |
1.2 Approach | 5 |
1.3 Major Achievements | 6 |
1.4 Structure of Thesis | 7 |

### 2 Motivations and Objectives

2.1 Using Explanations in Conceptual Modeling | 9 |
2.2 Improving Conceptual Modeling | 17 |
2.3 The Explanation Component | 18 |

## II State of the Art and Requirements

### 3 Information Systems Engineering and Conceptual Models

3.1 Basic Terminology | 23 |
3.2 Information Systems Development | 25 |
3.3 Requirements Engineering | 27 |
3.4 Conceptual Modeling | 30 |
3.5 The Dual Nature of Conceptual Models | 32 |
3.6 Modeling Environments | 35 |

### 4 Comprehension in Conceptual Modeling

4.1 Characterizing Comprehensibility | 41 |
4.2 Combining Strategies | 46 |

### 5 Explanations in Communication

5.1 Basic Terminology for Communication | 49 |
5.2 Quality of Communicated Messages | 51 |
5.3 The Concept of Explaining | 56 |
5.4 Types of Explanation | 56 |
5.5 Representing Explanations | 60 |
5.6 Two Levels of Explanation .................................. 66

6 Deep Explanation Generation ................................ 69
   6.1 The Main Principles of Explanation Generation ....... 69
   6.2 Deep Generation ......................................... 73
   6.3 Surface Generation ...................................... 89

7 Requirements to Explanation Generation in Conceptual Modeling ........ 91
   7.1 Explanation Needs ....................................... 91
   7.2 Integration Requirements ................................. 93
   7.3 Requirements to Explanation Component ............... 95

III The Deep Explanation Generation Approach .................. 97

8 A CASE Environment with Explanation Facilities .......... 99
   8.1 Architectural Basis ....................................... 99
   8.2 Representations and Components ......................... 100
   8.3 Internal Structures of Explanation Component .......... 103
   8.4 Generality of Architecture ............................... 105
   8.5 The Conceptual Modeling Cycle ........................... 106

9 Principles for Deep Generation in Conceptual Modeling .... 109
   9.1 Basic Principles ......................................... 109
   9.2 Architecture ............................................ 111
   9.3 Deep Generation Process ................................ 114
   9.4 Generality of Component ................................ 120

10 A Formalism for Deep Generation ........................... 123
    10.1 An Explanation Modeling Language EML .............. 123
    10.2 Predefined Attributes ................................ 129
    10.3 Representing Explanation-Relevant Knowledge ....... 133
    10.4 Explanation Generation Algorithm .................... 140

11 Building a Strategic Component Core ....................... 151
    11.1 The Construction of a Generic Component Core ...... 152
    11.2 Parts of an Extended Source Model .................... 161
    11.3 Generating Deep Explanations ........................ 167
    11.4 Tactical Component .................................... 173

IV Realization and Evaluation .................................. 177

12 Realization of Strategic Component Core .................. 179
    12.1 PPP with Explanation Component ....................... 179
    12.2 EML Representations in PPP ............................ 180
    12.3 Explanation Generation in PPP ......................... 181
    12.4 Interface to Other Components ........................ 194
    12.5 Realization in Other Environments ..................... 199
CONTENTS

12.6 Integration Requirements ......................................................... 203

13 Quality of Deep Explanation Generation Process .............. 205
   13.1 A Conceptual Modeling Scenario ........................................ 205
   13.2 Meeting the Standards of Textuality .................................... 209
   13.3 Some Problems and Limitations ........................................... 213

14 Related Explanation Generation Systems .......................... 219
   14.1 Existing Explanation Generation Systems ......................... 219
   14.2 Deep Generation Principles .............................................. 224
   14.3 Internal Formalism .......................................................... 227

V Extensions and Conclusions ....................................................... 229

15 Towards a Complete Integrated Explanation Component ......... 231
   15.1 Improving Generic Strategic Component Core .................... 231
   15.2 Extending Meta Model ...................................................... 232
   15.3 Adding a Tactical Component ............................................ 232
   15.4 Additional Routines in Explanation Component ................. 233

16 Conclusions ................................................................................. 235
   16.1 Major Achievements .......................................................... 235
   16.2 Further Work and Conclusions ............................................. 236

VI Appendices and Bibliography ..................................................... 239

A Conceptual Model for a Banking System ................................. 241
   A.1 System Description .............................................................. 241
   A.2 A Language for Representing the System Model ................. 242
   A.3 Conceptual Model .............................................................. 243

B The PPP CASE Environment ....................................................... 247
   B.1 Conceptual Modeling in PPP ................................................ 248
   B.2 Constructing a PPP Model ................................................... 250
   B.3 Validating the PPP Model .................................................... 258

C Outline of Functional Grammar ................................................. 261
   C.1 Knowledge Representation Language ................................... 262
   C.2 Expression Rules ............................................................... 269

D Translation from EML to Functional Grammar ......................... 271

E Discourse Strategies ................................................................... 275

Bibliography .................................................................................... 287
# List of Figures

1.1 The dual nature of conceptual models. ........................................ 4
1.2 An explanation component for conceptual modeling. .................. 5

2.1 An explanation showing what a process looks like. .................... 11
2.2 Illegal constructions in $PrM_e$. In (a) an a posteriori rule is violated, in (b) an a priori rule. ........................................ 12
2.3 The explanation component. ................................................. 19

3.1 The activity-oriented process model used by IFIP Working Group 8.1. .. 26
3.2 Requirements engineering and design engineering (adapted from [202]). .. 31
3.3 The modeling approach to information systems development. .......... 36

4.1 (a) Part of a textual data model in Gist. (b) Part of a graphical data model in BNM. ........................................ 43
4.2 A horizontal abstraction of the diagram in Figure A.3. ................. 44
4.3 A Statechart specifying the top-level reactive behavior of our banking system. ........................................ 45

5.1 A simple communication model. .............................................. 50
5.2 Explaining how a light dependent resistor looks like (from [38]). ..... 55
5.3 Main types of explanations in current generation systems. ........... 59
5.4 Text structure and two possible realizations (adapted from [213]). ... 61
5.5 (a) The generic RST schema. (b) An alternative notation for the schema. 63
5.6 RST structure of the functional description of P3. .................... 64
5.7 An RST structure from [205]. ............................................. 65
5.8 Deep explanation in EES (adapted from [174]). ....................... 67
5.9 Deep explanation for multimedia explanation (adapted from [163]). ... 68

6.1 Models and steps in explanation generation. ........................... 73
6.2 Entity relationship model with two generalization hierarchies. ........ 76
6.3 Declarative discourse schema for describing the constituents of an object in TEXT. ........................................ 78
6.4 The procedural discourse schema used in TAILOR. .................... 79
6.5 Two partial deep explanations generated by Hovy’s structurer. (a) “Know is C4. It will arrive on 4/24.” (b) “Know, which is C4, is heading SSW. It will arrive on 4/24.” ........................................ 84

8.1 Components and models in our CASE architecture. .................... 101
8.2 The internal structures of the explanation component. ................ 104
8.3 Integrated CASE environment for constructing and validating conceptual models. ................................................. 107

9.1 The main principles of our deep explanation generation process. ................................................................. 110
9.2 The strategic component architecture. ........................................................................................................... 112
9.3 Informal deep explanation. ......................................................................................................................... 120
9.4 DSM elements are generalizations and/or classifications of source model elements. ........................................... 122

10.1 The attribute hierarchy. ............................................................................................................................... 130
10.2 (a) Declaration of process P3. (b) Process P3 can receive the flow RequestedJoan. .............................................. 135

11.1 The structures of the domain structure model. ............................................................................................. 153
11.2 The development of a discourse strategy. ..................................................................................................... 157
11.3 Overall description of the concept of "process". ............................................................................................ 171
11.4 Detailed syntactic explanation of a process. ................................................................................................. 172
11.5 Description of process P3. ........................................................................................................................ 173
11.6 Explaining the result of an execution. ........................................................................................................... 174
11.7 Transformations in the inter-clause realization part. ..................................................................................... 175

12.1 Representations in the PPP environment for explanation generation. ......................................................... 180
12.2 Request and explanation structure for the question about cardinalities in PhM. .......................................... 184
12.3 Describing the concept "entity" from a syntactic point of view. ...................................................................... 186
12.4 Explaining the violation of an a posteriori verification rule. ...................................................................... 188
12.5 Describing an account. .............................................................................................................................. 189
12.6 Describing a PLD diagram. ....................................................................................................................... 190
12.7 Assuming that modeling deliberations are integrated in the PPP model, the explanation component can justify model elements like process P1.2. ................................................................. 192
12.8 The history explanation. ........................................................................................................................... 193
12.9 An input justification. ................................................................................................................................... 195
12.10 Translating PPP models to EML representations. ...................................................................................... 196
12.11 Constructing explanation requests and extended source models. .............................................................. 198
12.12A simple Statechart. .................................................................................................................................... 201
12.13 Only states in Statecharts are dynamic concepts. ....................................................................................... 201
12.14 Both states and transitions in Statecharts are dynamic concepts. ............................................................... 201
12.15 Integration of process and rule in TEMPORA (adapted from [136]). ......................................................... 202
12.16 Explanation of rule in TEMPORA. ........................................................................................................... 203

13.1 Simple description of entity classes. ........................................................................................................... 206
13.2 An explanation of accounts tailored to user's level of knowledge. .............................................................. 208
13.3 Input justification for analysts. .................................................................................................................. 209
13.4 Problematic PLD diagram for system behavior explanations. ................................................................. 215
13.5 The relationship between conceptual models and explanations. ............................................................... 217

A.1 (a) XOR port. (b) AND port. (c) Input construction saying that a is received, but only one of b and c. (d) Flow a triggers the process, whereas b is used during the execution of the process. .................. 242
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.2</td>
<td>The overall diagram P0 Register customers and process loan requests.</td>
<td>243</td>
</tr>
<tr>
<td>A.3</td>
<td>Decomposition of P3 Process loan request.</td>
<td>244</td>
</tr>
<tr>
<td>B.1</td>
<td>Conceptual modeling in PPP.</td>
<td>249</td>
</tr>
<tr>
<td>B.2</td>
<td>PhM model for the banking system.</td>
<td>252</td>
</tr>
<tr>
<td>B.3</td>
<td>Top-level diagram for the PrM model.</td>
<td>253</td>
</tr>
<tr>
<td>B.4</td>
<td>Decomposition of process P1.</td>
<td>253</td>
</tr>
<tr>
<td>B.5</td>
<td>Decomposition of process P2.</td>
<td>254</td>
</tr>
<tr>
<td>B.6</td>
<td>A PLD describing process P1.2 Verify amount.</td>
<td>256</td>
</tr>
<tr>
<td>B.7</td>
<td>PLD diagram for process P1.1 Verify transaction.</td>
<td>257</td>
</tr>
<tr>
<td>B.8</td>
<td>PLD diagram for process P2.1 Process application.</td>
<td>257</td>
</tr>
<tr>
<td>B.9</td>
<td>Generated Ada code and parts of a recorded trace.</td>
<td>259</td>
</tr>
<tr>
<td>B.10</td>
<td>(a) A model view resulting from horizontal abstraction. (b) Ports abstracted away and layout is improved.</td>
<td>260</td>
</tr>
<tr>
<td>C.1</td>
<td>Representational levels in Functional Grammar (simplified from [67]).</td>
<td>263</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Davis's hypotheses about requirements engineering [58] ............... 28
5.1 Mathematical formulas may be used to explain events in XPLAIN [234] ... 55
6.1 Discourse schema for misattribution in ROMPER ..................... 77
6.2 Plan operator in Dalianis's system [55]. An entity is explained by giving an instance as an example ........................................... 80
6.3 Plan operators in EES. Effect is the operator's discourse goal, constraints its preconditions and nucleus and satellites its subgoals .......... 81
6.4 The RST-relation sequence formalized as a plan operator in [116] .... 83
6.5 Plan operator for multimedia explanations [163]. Header is the operator's goal, constraints and preconditions its preconditions, and decomposition its subgoals ....................................................... 89
9.1 Content of a typical explanation request .................................. 115
9.2 Informal declaration of an element of the extended source model ........ 116
9.3 Informal discourse strategies. <list of goals> means that only one of the goals should be used, {list of goals} that as many as possible should be used 118
9.4 Some elements of the domain structure model .......................... 121
10.1 F-structures ........................................................................ 124
10.2 A grammar for EML .............................................................. 125
10.3 EML structure for "P3 can receive Requested loan" .................... 126
10.4 Discourse strategies structures and dynamics represented in EML (the strategies will be made more complicated in Chapter 12) ........ 126
10.5 A generic EML structure and its interpretation .......................... 127
10.6 The five bottom structures are EML-resolutions of the upper structure 129
10.7 Attributes used in EML .......................................................... 131
10.8 Attributes used in EML .......................................................... 132
10.9 Simplification rules in EML .................................................... 133
10.10 Example showing how the simplification rules work .................... 133
10.11 Process is declared as a dynamic concept ............................... 134
10.12 Including formal language expressions in EML using formula(...) .... 135
10.13 (a) Offer_recommendation declared as an instance of item. (b) 801 declared as an instance of offer_recommendation ....................... 136
10.14 Annotating source model elements with view information ........... 136
10.15 A user characterization submodel specifying the views associated with a user in some context .............................................. 137
10.16 (a) General structure of schema-based discourse strategy. (b) General structure of operator-based strategy ............................. 138
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.17</td>
<td>Explanation request represented in <em>EML</em>.</td>
<td>139</td>
</tr>
<tr>
<td>10.18</td>
<td>Example of a deep explanation.</td>
<td>141</td>
</tr>
<tr>
<td>10.19</td>
<td>Unification of attribute value structures.</td>
<td>142</td>
</tr>
<tr>
<td>10.20</td>
<td>The predicate <code>generate_deep_explanation</code> for generating deep explanations.</td>
<td>145</td>
</tr>
<tr>
<td>10.21</td>
<td>The predicate <code>expand_subgoal</code> for generating deep explanations.</td>
<td>146</td>
</tr>
<tr>
<td>11.1</td>
<td>Requesting a short explanation of the term “process”.</td>
<td>156</td>
</tr>
<tr>
<td>11.2</td>
<td>Discourse strategies.</td>
<td>159</td>
</tr>
<tr>
<td>11.3</td>
<td>Discourse strategies.</td>
<td>160</td>
</tr>
<tr>
<td>11.4</td>
<td>Discourse strategies.</td>
<td>161</td>
</tr>
<tr>
<td>11.5</td>
<td>Some structure elements in the meta model of <em>PrM_r</em>.</td>
<td>166</td>
</tr>
<tr>
<td>11.6</td>
<td>Two model constraint elements in the meta model of <em>PrM_r</em>.</td>
<td>166</td>
</tr>
<tr>
<td>11.7</td>
<td>Some of the constraints on structure elements in the meta model of <em>PrM_r</em>.</td>
<td>167</td>
</tr>
<tr>
<td>11.8</td>
<td>A constraint on a model constraint element in *PrM_r_’s meta model.</td>
<td>168</td>
</tr>
<tr>
<td>11.9</td>
<td>Elements of the explanation conceptual model for the system in Appendix A.</td>
<td>168</td>
</tr>
<tr>
<td>11.10</td>
<td>Portions of a trace from executing the bank model in Appendix A.</td>
<td>169</td>
</tr>
<tr>
<td>11.11</td>
<td>Request and deep explanation for an answer to “What is a process?”</td>
<td>170</td>
</tr>
<tr>
<td>11.12</td>
<td>Two possible clause realizations in intra-clause realization.</td>
<td>176</td>
</tr>
<tr>
<td>12.1</td>
<td>Elements of the explanation meta model of PPP.</td>
<td>181</td>
</tr>
<tr>
<td>12.2</td>
<td>Combining two elements of the conceptual model.</td>
<td>182</td>
</tr>
<tr>
<td>12.3</td>
<td>Elements of the conceptual model.</td>
<td>182</td>
</tr>
<tr>
<td>12.4</td>
<td>Elements of the execution trace.</td>
<td>183</td>
</tr>
<tr>
<td>12.5</td>
<td>The detection of a model error.</td>
<td>187</td>
</tr>
<tr>
<td>13.1</td>
<td>User knows that customers have accounts.</td>
<td>207</td>
</tr>
<tr>
<td>14.1</td>
<td>In EES the user response “Huh?” initiates an elaboration explanation of what the system thinks was not understood in the last explanation (adapted from [174]).</td>
<td>222</td>
</tr>
<tr>
<td>14.2</td>
<td>Explanation types supported in various systems.</td>
<td>224</td>
</tr>
<tr>
<td>14.3</td>
<td>Optional satellites in EES encoded in <em>EML</em>.</td>
<td>227</td>
</tr>
<tr>
<td>14.4</td>
<td>Hovy’s growth points encoded in <em>EML</em>.</td>
<td>227</td>
</tr>
<tr>
<td>A.1</td>
<td>Process logic rules.</td>
<td>242</td>
</tr>
<tr>
<td>A.2</td>
<td>Process logic rules for process P1 Check customer.</td>
<td>244</td>
</tr>
<tr>
<td>A.3</td>
<td>Process logic rules for process P2 Register customer.</td>
<td>245</td>
</tr>
<tr>
<td>A.4</td>
<td>Process logic rules for process P3.1 Calculate maximum size of loan.</td>
<td>246</td>
</tr>
<tr>
<td>A.5</td>
<td>Process logic rules for process P3.2 Respond to request.</td>
<td>246</td>
</tr>
<tr>
<td>C.1</td>
<td>Semantic roles used in FG.</td>
<td>265</td>
</tr>
<tr>
<td>C.2</td>
<td>Various levels of FG predications (adapted from [67]).</td>
<td>267</td>
</tr>
<tr>
<td>C.3</td>
<td>Operators included in FG predications.</td>
<td>268</td>
</tr>
<tr>
<td>C.4</td>
<td>Satellites included in FG predications.</td>
<td>268</td>
</tr>
<tr>
<td>D.1</td>
<td>The function $f_{EML\rightarrow FG}$.</td>
<td>272</td>
</tr>
<tr>
<td>D.2</td>
<td>Example of an <em>EML</em> structure.</td>
<td>273</td>
</tr>
<tr>
<td>E.1</td>
<td>Rhetorical relations used by our discourse strategies.</td>
<td>276</td>
</tr>
</tbody>
</table>
Part I

Introduction and Motivation

This part introduces the problems in conceptual modeling that we will be dealing with in this thesis, our approach to these problems, and the main objectives and motivations guiding the work.

Chapter 1 The problem statement is presented, followed by the chosen solution approach, the main achievements, and the structure of the thesis.

Chapter 2 Using scenarios and illustrations, the chapter motivates the work and puts forward the main objectives of the thesis.
Chapter 1

Introduction

This work is about comprehension in conceptual modeling. Recognizing the importance of formal modeling languages in conceptual modeling environments and the need for unskilled users to participate in the modeling and understand the models constructed, we present a generic component that can present model-related information in a manner more easily understood by users. This component can explain aspects of modeling languages, conceptual models, and model executions, and can be added to several modeling environments with only minor modifications of existing components of the environments.

We start out by describing the characteristics of conceptual modeling that lead to the problems of comprehension. Concluding that there seems to be an inherent conflict between comprehensibility and formality, we summarize the approach taken in this thesis in Section 1.2 and the major results achieved in Section 1.3. The structure of the thesis is outlined in Section 1.4.

1.1 Problem Description and Assumptions

In information systems engineering, conceptual models are constructed as part of the problem analysis and requirements determination process. They reflect the analysis of existing or desired information systems and form a common reference frame for the parties involved in the development project. As the projects go on, the validated conceptual models serve as directives for the design and the subsequent implementation of the systems.

The quality of the conceptual modeling process is associated with the notions of comprehension and communication. In the process of modeling, a particular language is used to record and structure properties of the information system. For the modeling to be effective, the modeler should feel comfortable with the concepts of the language as well as be able to combine them to form a model. This, in turn, requires a basic understanding of the modeling language’s syntax and semantics. Later on, the conceptual model is validated to
reach an agreement on the properties of the information system. The importance of user participation at this stage of the project is now generally accepted, and the model’s success as a communication basis depends on the parties being able to read and comment on the model. Since these people tend to bring different roles, skills and attitudes into the project, it can be a challenging task to ensure that the model is correctly and uniformly interpreted. Still, as the validation of the conceptual model is vital to the outcome of the project, all the parties involved should feel certain about the content of the model parts relevant to them.

Research in conceptual modeling has addressed the comprehensibility of both the process and the product of modeling. What modeling languages are concerned, graphical symbols, abstraction mechanisms and user-oriented concepts have been introduced to make the languages more accessible to the people modeling or reading the models. And there are now tools that guide the modeling process and support the validation process through heavy model analyses. The outcome of these analyses, however, depends on the precision and expressiveness of the conceptual models. The employment of formal languages has been encouraged in this respect, since these are highly amenable to automatic analysis or execution and can ease the designers’ burden of interpreting possibly ambiguous conceptual models. Tools relying on formal modeling languages, then, offer functions that help their users comprehend the model being constructed and get an impression of the consequences of using it as an implementation goal. However, increased formality tends to complicate both the syntax and the semantics of the modeling language. This impairs the language’s comprehensibility and can easily impede or slow down the process of modeling. Also, the results from automatically analyzing or executing formal models may often be difficult to utilize for people not specially trained for that purpose.

From a design perspective, formal conceptual models have proven very useful. They allow rigorous tool-supported development, in which more or less automatic model transformations and analyses ease the transition to design activities, and a precisely defined language semantics frees the designers from possessing domain knowledge not specified in the conceptual model. And as the development project proceeds, the conceptual model serves as a standard for measuring the quality of the design and implementation of the system.
In sum, there seems to be a conflict between comprehensibility on the one hand and formality and tool support on the other hand. Comprehensibility encourages user participation in the project and strengthens the confidence that the conceptual model meets the intention of the users. Formality opens for tool support, which has proven vital when more complicated model manipulations are needed and the transition to design activities is made. The duality of conceptual models is illustrated in Figure 1.1.

In this work, we distinguish between representation and presentation of models. We assume the conceptual models to be represented in a manner appropriate for automatic analysis, execution, and transformation. From a representational point of view, thus, powerful formal languages are assumed to be used in the modeling environment. Comprehensibility, as we will use the term here, concerns the presentational side of conceptual modeling, and in this work the focus is on how the presentation of conceptual models and the results from analyzing and executing them can enhance user comprehension.

1.2 Approach

We will investigate how a presentation component can enhance user comprehensibility in formal conceptual modeling. The component is to take part in dialogues with the users, answering questions related to the modeling language, the conceptual model, or the results from analyzing or executing conceptual models (see Figure 1.2). Since the answers, which are to be tailored to both user and context, are called explanations, the whole presentation component is referred to as an explanation component.

Building such a component, we confront three basic questions related to explanation generation:
What to explain? The assessment of explanation needs in conceptual modeling is restricted by the information available in the environment. We base our assessment on the representations found in contemporary CASE environments and meta environments.

How to explain? In developing principles for explanation generation, we draw on theories of linguistic communication and natural language generation systems from other domains.

How to realize the component? We develop a special formalism that can represent all the information relevant to explanation generation as well as the explanations themselves. In doing so, we integrate formalisms used for knowledge representation, linguistic representation, and natural language generation.

The requirements to the component are detailed as the work goes on and the complexities of the component are revealed.

Building an explanation component is here not necessarily claimed to be the best approach to model comprehensibility. The task here is to investigate its potential, so that the approach later can be evaluated and compared with other available approaches.

1.3 Major Achievements

The outcome of the work is a generic explanation component core that can explain properties of languages and models, and describe results from analyzing or executing conceptual models. It generates representations called “deep explanations”, which specify contents, explanation structures, and presentation media for the final explanations. The deep explanations are realized as natural language texts, possibly supplemented by views of the conceptual model or formulas in some formal language, and the component is meant to be added to graphical conceptual modeling environments (CASE environments).

In more detail, we have the following major achievements:

- We show how explanations can be useful in understanding syntactic and semantic aspects of modeling languages, modeling deliberations, properties of conceptual models, and their executions.

- We present a general CASE architecture that integrates explanation facilities in conceptual modeling environments with facilities for view generation and model execution.

- We suggest explanation generation principles that enable the generation of user-tailored and context-tailored coherent deep explanations.
1.4 Structure of Thesis

- We introduce meta modeling in explanation generation, so that the explanation component can be used on several modeling languages and in several CASE environments.

- We work out a special formalism for representing all the knowledge pertaining to the deep explanation generation process.

These contributions are brought together in the implementation of a strategic component core. This is a prototype system that is used to determine the content and structure of explanations, as well as the media for presenting their various parts. The core is tried out on models found in the PPP CASE environment, and it is shown how it generates the desired explanations.

1.4 Structure of Thesis

There are five main parts of the thesis: introduction and motivation, state of the art and requirements, the deep explanation generation approach, realization and evaluation, and extensions and conclusions. They all contain a number of chapters and are briefly described below.

Part I — Introduction and Motivation introduces the thesis and provides some scenarios and basic objectives for the work carried out. It contains two chapters:

Chapter 1 describes the problems motivating the work, the research approach, the major results achieved, and the structure of the thesis.

Chapter 2 sketches some scenarios showing how explanations can fit into the conceptual modeling process. The result is a set of overall objectives for the work.

Part II — State of the Art and Requirements discusses issues connected to both conceptual modeling and explanation generation. Its purpose is threefold: In the first place, it constitutes a point of departure, presenting earlier works and results that underly our explanation component. Secondly, it provides the necessary terminology for understanding the rest of the thesis, and thirdly, it leads to the formulation of rather detailed requirements to the component that we shall build.

Chapter 3 presents some problematic issues in information systems engineering and discusses the roles of conceptual models and CASE environments in the development of information systems.

Chapter 4 presents various strategies, among them explanation generation, for enhancing user comprehension in conceptual modeling.

Chapter 5 introduces linguistic theories and concepts for describing explanation structures and assessing explanations' qualities in communication processes.

Chapter 6 reviews tasks, strategies, and techniques found in current natural language generation systems, focusing on the early stages of generation.
Chapter 7 outlines the detailed requirements to the work of this thesis.

Part III — The Deep Explanation Generation Approach presents a general CASE architecture with explanation facilities, some principles and a formalism for explanation generation and a component realizing the generation of deep explanations.

Chapter 8 describes a general CASE architecture, in which an explanation component co-operates with a view generation component and an execution component to produce presentations of conceptual modeling information.

Chapter 9 suggests some principles for generating deep explanations, which specify the final explanations' content, structure, and associated presentation media.

Chapter 10 defines a formalism, the EML language, for representing the knowledge used in the deep explanation generation process.

Chapter 11 shows how a simple deep explanation generator, the strategic component core, is built using the formalism and the principles already introduced.

Part IV — Realization and Evaluation validates the work, showing how the strategic component core fits into the PPP CASE environment and relating the explanations generated to the requirements from Chapter 7. Also, our component core is compared to other natural language generation systems.

Chapter 12 uses the strategic component core in the PPP CASE environment and shows how the desired explanation types are supported and how the component is integrated into the environment.

Chapter 13 indicates the explanations' qualities with respect to coherence, user-tailoring and context-tailoring, but also some problems and limitations with the approach.

Chapter 14 relates the strategic component core to other generation systems.

Part V — Extensions and Conclusions discusses how the strategic component core can be extended to a complete explanation component and concludes the thesis.

Chapter 15 summarizes the additional components and routines needed to construct a complete integrated explanation component from the strategic component core.

Chapter 16 sums up the major achievements of the thesis and suggests some directions for further work.

The example in Appendix A is used in the first three parts, whereas the example illustrating the PPP language in Appendix B is referred to in the evaluation part.
Chapter 2

Motivations and Objectives

In this thesis we investigate how explanation generation technologies can be applied in requirements engineering. We will be dealing with both the process and the product of conceptual modeling, and the underlying goal is to enhance users' comprehension in the conceptual modeling of information systems. In the process of modeling, the objective is to generate explanations to help the modelers understand and use the language, as well as to inform her when illegal constructions are made. On the product side, we will see how explanation generation can make it easier for all the parties involved to read conceptual models and interpret executions of them. When executing conceptual models, then, we want the explanation generation component both to guide the execution and explain the observed behavior of the model.

Our focus is on how an explanation generation component can be constructed and integrated into environments for conceptual modeling, CASE environments\(^1\). As a basis for discussing the detailed objectives with respect to the component, we first illustrate the use of explanations with a few, representative examples. The way these kinds of explanations can alleviate the problems of comprehension is also given some thoughts.

2.1 Using Explanations in Conceptual Modeling

As a provisional definition, let us assume an explanation to be a text, or a combination of text and graphics, that describes a phenomenon and satisfies some user's need for information. Although vague and superficial, this definition suffices for the presentation to follow here\(^2\). We make the general assumption that both the content and the structure of an explanation should vary according to the properties of the user requesting the explanation, the reasons for requesting it, and the context in which it is requested.

\(^1\)CASE is an acronym for Computer-Aided Software Engineering.

\(^2\)In Section 5.3, there is a more comprehensive discussion of the concept of explanation.
What this indicates, is the explanations’ dependence on general text linguistic theories about how humans communicate with each other. Let us then take a typical conceptual modeling process. We can distinguish between two process activities, the construction and the validation of models, assuming both to be able to take advantage of explanation generation technologies. In the modeling, there are people involved that bring with them different roles, skills, and attitudes.

In the following, we will demonstrate the usefulness of explanations with reference to the construction and validation of conceptual models, using illustrations from the domain described in Appendix A. The conceptual modeling language used is $PrM_r$, which is also introduced in that appendix.

Construction of Models

Ideally, the modelers have intimate knowledge of both the syntax and the semantics of the conceptual modeling language. They feel comfortable with the language, and find it easy and effective to use in the modeling process. In practice, though, one cannot always assume this to be the case:

- Formal languages are notoriously detailed and complex. The emphasis on compact, unambiguous and effective ways of representing information reflects a gap of philosophy between these languages and natural languages that is not easily bridged.

- Since a single modeling language is usually attributed to only one perspective of a system, one will often have to switch between languages, or use several in combination. This complicates the learning of the languages, since one do not get the time to develop a proper understanding of them.

- To an increasing extent, domain experts or end-users are getting involved in the modeling process. These cannot always be expected to know and be familiar with the conceptual modeling language used in the project.

- The complexities and uncertainties of the domain itself make it difficult to concentrate on the details of the modeling language. Since it is more important to investigate the domain than to construct a syntactically correct model in the beginning, complicated features of the language tend to be neglected if they disturb the analysis of the domain.

In search of improvements in conceptual modeling, a wide range of works has been done within the fields of modeling languages and tool support. Abstraction mechanisms, user-oriented concepts and graphical symbols are now generally recognized as important to a language’s comprehensibility, but the effect of these strategies is hampered by the overall need for formal and expressive languages. And even though tool support like syntax-oriented editors, model checking routines, and help facilities is now often included in modeling environments, there are still problems with the comprehensibility of the modeling process.
2.1. Using Explanations in Conceptual Modeling

Figure 2.1: An explanation showing what a process looks like.

Explanations can help the modelers with the language’s syntax and semantics, and if a verification module is included in the environment, they can also provide an interface to error messages from verification checks. Verification rules may be classified as \textit{a priori} or \textit{a posteriori} [242]. An a priori rule can never be violated (e.g. a flow in a \textit{PrM} model can never go from one store directly to another), whereas an a posteriori rule can be temporarily violated in the modeling process as long as the final model obeys the rule (e.g. all processes in a \textit{PrM} model must have at least one output flow).

In connection with the construction of conceptual models, there are at least two important explanation contexts, the \textit{meaning} context and the \textit{reference} context. In the meaning context, the focus is on how terms refer to underlying meanings, in the reference context on how terms refer to constructs of the modeling language.

\textbf{Language semantics} The modelers can ask questions concerning the meaning of language concepts. Since the semantics of conceptual modeling languages seldom are explicitly represented, we will probably have to suggest — rather than to precisely specify — concept meanings. It is possible, however, to indicate semantic properties by describing for example the concept’s type, purpose and/or structural relationships to other concepts. Consider the dialogue below:

\begin{itemize}
  \item \textbf{user:} \textit{"What is a process?"}
  \item \textbf{system:} \textit{"A process is a dynamic concept that generates a number of outflows on the basis of a number of inflows. Its content is described as a separate diagram or as a set of process logic rules. It is triggered by the arrival of a flow."}
\end{itemize}

The first sentence tells us about the type and purpose of processes, whereas the second one says how they are related to other language concepts. In this context, the question was assumed to refer to the underlying meaning of processes (i.e. the meaning context was used). Looking at the conceptual model in Figure A.2, a user might also wonder what graphical symbols are referring to processes. She is then in the \textit{reference} context, and the question \textit{"What is a process?"} could then lead to the graphical explanation in Figure 2.1. Note that explanations with graphical elements involve the generation of \textit{views} of the conceptual model.
Language syntax Questions concerning syntactic properties of the modeling language are all in the reference context. They request information about how constructs of the language are related to other constructs, and can draw on a priori rules as well as a posteriori rules. The question below refers to the processes used in $PrM_r$.

user: “How are processes used?”

system: “A process has an identifier and a purpose. It must receive at least one data flow and send at least one data flow. All the inflows must be included in an input port construction, all the outflows in an output port construction. At least one inflow must be marked as triggering. A process’s content is described as a new diagram or as a set of process logic rules.”

If the system has reasons to believe that the modeler is familiar with the traditional data flow diagrams, it can leave out certain structural properties that are already known to her:

system: “All inflows to a process must be included in an input port constructions, all outflows in an output port construction. At least one inflow must be marked as triggering. A process’s content is described as a new diagram or as a set of process logic rules.”

In this way, the response is tailored to the user’s knowledge. If the user has no experiences with process-oriented languages, the original explanation accompanied by a model example like the one of P3 in Figure 2.1 could be appropriate.

Verification checks Most modeling environments include syntax-oriented editors, in which only constructs from the alphabet of the language are available in the modeling process. The environment will usually also offer some routines for checking the syntax, the consistency, and internal completeness of the model being built. If these verification routines are run on the model in Figure 2.2(a), for example, the lack of flows out from the process is detected, and the explanation below might be generated:
2.1.用解释说明概念建模

系统："The diagram is illegal, since P3 does not produce any flow. A process must produce at least one flow."

The rule used above is an a posteriori verification rule. If a flow connects two data stores—as is the case in Figure 2.2(b)—an a priori rule is violated and we can get a message like the one below:

系统："The origin and the destination of a data flow cannot both be data stores."

Note that a good editor would not allow the user to violate a priori verification rules. In that case, an explanation component could be used to inform the modeler why her construction was not accepted.

Validation of Models

The product of modeling is a conceptual model that serves as a basis for the design of the information system. Since conceptual modeling is an iterative process of model building and model analysis, though, there will be many partial and temporary products before a final, satisfactory conceptual model has evolved. An important task in the analysis of these products is to determine their appropriateness with respect to user's needs and intended requirements, i.e. model validation. In order to do that, the parties involved in the modeling must be able to understand the properties and the consequences of those parts of the models that are relevant to them.

Apart from the formality and unfamiliarity of the modeling language, though, these models may be hard to validate due to their scope and complexity.

- The scope of the conceptual model concerns both the number of phenomena modeled and to what level of detail they are described. In many cases, the model includes much more information than what is relevant to a particular user reading the model. The large scope of the model can disturb the user's ability to concentrate on and validate those parts that she is knowledgeable in.

- A conceptual model of an information system easily gets very large and complex. There are perceptual problems in reading very intricated models, and the details of these models tend to overshadow their overall structures and behavior.

So, in realistic information systems development projects, even models represented in familiar and not so formal languages can be hard to interpret due to the inherent complexities of information systems.
Strategies for enhancing user comprehension

The previously mentioned strategies for improving the comprehensibility of modeling languages have also profound effects on the validation of the corresponding models. In addition to these, there are other strategies that expose model properties through heavy analyses or model manipulations, like view generation, model execution, and model translation. These strategies are all valuable, though they are restricted to specific aspects and have certain limitations.

What validation is concerned, explanations can help the users read the conceptual models correctly. If modeling decisions are recorded, they can also justify the inclusion of elements in the model. At last, they can guide and explain executions of these models, and in that sense they provide an interface between executable model or trace, and user.

Model inspection The users can ask questions about properties of the conceptual model. The questions may be related to structural or dynamic properties, and the answers are user-tailored and context-dependent. An explanation can be adapted to the user and the context, as indicated by the three alternative explanations below:

user: How is P2 Register customer activated?

system1: “P2 is activated when either Customer.name is sent from the user or No_registration is sent from P1. The user sends Customer.name if she wants to register a new customer. P1 checks the customer and is triggered by a Loan_request from the user. No_registration is sent if the customer is not found in the database and No.customer is returned to P1.”

system2: “P2 is activated when either Customer.name is sent from the user or No_registration is sent from P1.”

system3: “P2 is activated if the user wants to register a new customer or a customer not already registered requests a loan.”

The first explanation could be suitable for a modeler inspecting the details of the model. The next explanation could be used in the same inspection context, but in this case either the user knows the details left out from the first one or the context makes the origin of Customer.name and No_registration irrelevant. In the third one, we have changed to a context where user’s interaction with the system is emphasized. The explanation could be given to a user unfamiliar with the details of the system, but with a good understanding of how the system is expected to be used.

The next dialogue illustrates how different levels of abstraction can be exploited to tailor a particular explanation:
2.1. Using Explanations in Conceptual Modeling

user: "How is Loan_limit computed?"

system1: "Loan_limit is produced by P3.1. If New_customer.data is received, then Loan_limit.maximum = New_customer.data.income. Loan_limit.name is set to New_customer.data.name. If Customer.data is received, there are three cases: If Customer.data.classification is 'bad', then Loan_limit.maximum is 0. If Customer.data.classification is 'neutral', then Loan_limit.maximum is Total_savings.savings + Customer.data.income - Total_loans.loans. If Customer.data.classification is 'good', then Loan_limit.maximum is Total_savings.savings + Customer.data.income - Total_loans.loans. Loan_limit.name is set to Customer.data.name."

system2: "Loan_limit is computed on the basis of Total_savings, Total_loans, and either New_customer.data or Customer.data."

The first explanation describes in detail how Loan_limit is computed, and would be appropriate when the model is validated by users very knowledgeable in the domain. The second one gives a functional overview of the computation. It is assumedly most helpful in situations where the overall structure of the model is in focus, or the user cannot be expected to be able to get anything out of all the details or be able to comment meaningfully on the basis of them.

The following dialogue is an example of a request about structural properties:

user: "What is the content of flow Loan_request?"

system: "Loan_request contains two items, name and amount."

Of course, the content of these explanations could also differ according to explanation context and user characteristics.

**Modeling deliberations** Modeling deliberations typically express the arguments for including elements in the conceptual model. They can be made a part of the conceptual model, and can be used by the explanation generation component to justify the content of the model.

Consider this typical question about modeling deliberations:

user: "Why is P2 needed?"

system: "P2 is needed because the bank must be able to accept new customers."

Depending on the sophistication of the recorded deliberations, we can justify the inclusion of elements (as above), explain why elements are considered a correct representation of some phenomenon, or in general provide arguments that support or refute the existence or use of model elements. Provided that more detailed deliberations are included in the conceptual model, the following dialogue could be possible:
user: “Why is P2 correct?”

system: “To register a new customer, only her name and income is necessary. Since the bank has not served the customer before, no classification is available, and it seems reasonable to assume a ‘neutral’ classification to start with.”

Modeling deliberations are seldom formalized, so the explanation component’s task would normally be to track the appropriate arguments and just make them available to the user. In the conceptual models, deliberations are modeled as special kinds of model elements and are linked to the other elements of the model. We will from now on not distinguish between conceptual model and modeling deliberations, but assume that a conceptual model may include deliberations in addition to the other kinds of information.

**Model execution**   Explanation generation technology can be combined with the execution of conceptual models to explain both the external and the internal dynamics of models. This requires, however, that the behavior of the models during execution is well documented in the form of detailed traces of computations and instantiations. Additionally, since executable conceptual models do not include any complete user interface specification, explanations can be used to explain what the system is doing and why certain inputs are necessary.

The two dialogues below illustrate the use of explanations in model execution environments. In the first dialogue, two possible explanations are included, each of them tailored to a specific context and/or user.

**user:**  “Why was the loan application rejected?”

**system**

- **system**

  "The requested loan was $100,000. The loan limit was $80,000. Using the rule

  \[
  \text{If Loan}\_\text{limit}\_\text{maximum} \text{ is less than } \text{Requested Loan}\_\text{amount},
  \]

  \[
  \text{then Recommendation}\_\text{result} = \text{rejected and Recommendation}\_\text{offer is equal to Loan}\_\text{limit}\_\text{maximum};
  \]

  \[
  \text{it was decided to send Loan}\_\text{rejection}.
  \]

- **system**

  "Because the requested loan was $80,000, the customer’s income $50,000, her classification ‘neutral’, her total savings $50,000, and her total loans $70,000.”

The dialogue below helps the users in the execution of the conceptual model.

**user:**  “Why do you need Customer\_\text{income}?”

**system:** “The customer was not registered. I need the customer’s income to register the customer and to process the loan request.”

In more advanced execution environments, symbolic or hypothetical executions are available. In symbolic executions, symbolic inputs are used instead of concrete values, and
2.2. Improving Conceptual Modeling

a trace specifying all the possible execution paths is generated. Hypothetical executions are typically initiated by “What-if” questions and force the execution environment to investigate the consequences of inputting certain values or using certain alternative model elements in the execution.

2.2 Improving Conceptual Modeling

The explanation generation component co-operates with other components in the conceptual modeling environment. Techniques for verifying models, generating model views, or executing models are all desirable to take full advantage of the explanation generation component.

What the explanation generation component does, is to make the available information understandable to the users. It serves as a front-end to conceptual models and other validation techniques, and its purpose is to sort out the irrelevant information and present the remaining parts in terms and structures familiar to the user. In that sense, the technique’s aim is to enhance comprehension both in the construction and the validation of conceptual models. It makes the modeling process more transparent, and instead of letting the formal modeling languages exclude users, it exploits its knowledge of human communication to include them.

More specifically, we can point out a number of areas that can benefit from these explanation facilities:

User involvement in the modeling process is naturally encouraged when all the users can understand the modeling language and the resulting models. Explanations confirm the conceptual model as a real communication frame to which all the users can refer in the course of modeling. Domain experts and end-users can actively use the explanations to learn how constructs of the language are used and how model constructions are to be interpreted.

Model views are available in many modeling environments. As opposed to in most environments, however, we can here relate views of the conceptual model to communication processes. The environment is not only offering general views of the model, but is able to use them purposely and consciously to make the explanations more illustrative.

Model execution is supported by explanation facilities at two levels. First, they provide a general help module that can guide the execution of these models. This is valuable in itself, and since conceptual models tend to be partial and poorly equipped with interface-related information, the help module gets even more crucial. Secondly, for the execution to be really useful, it should be feasible to trace experienced behavior back to elements of the conceptual model. Generating explanations, we can explain both the observable and the unobservable behavior of the model in terms of its elements, and in this way it gets easier to see how deficiencies exposed in the execution are rooted in the underlying model.
Modeling deliberations have so far often been added to the conceptual models in the forms of informal arguments, or just written as a separate document. An explanation generation component will provide the same interface to the deliberations as to the model, thereby making the deliberations an even more integrated part of the product of modeling.

Documentation is automatically generated by collecting the explanations of all the elements of the conceptual model. Furthermore, since the explanations may be tailored to different user groups, we can generate several documentations, one for each user group involved. Contrary to manually produced documentation, this approach is very flexible with respect to maintenance. If the model is modified, the new documentation can be generated immediately without any involvement of humans.

These exploitations of the explanation generation component are related to the examples in the previous section. If other techniques were available in the modeling environment, we could use the component as a front-end to these, too. For instance, in some CASE environments modeling methods, performance requirements, or whole development process are formalized as some kind of models. Explanation generation systems form a separate component that can be added as a front-end to all these model representations available in the environment. In this work, however, we will concentrate on the way explanations were used above.

2.3 The Explanation Component

Our explanation component is intended to be integrated into CASE environments. It is meant to form a separate component that can be added to an extant environment without major modifications to already included components. The generality of the component should make it adaptable to a wide range of CASE environments available today.

An overall architecture of the explanation component is indicated in Figure 2.3. The component works on model representations and results from various techniques used in the environment. In our case, this involves information from the conceptual modeling module, results from executing or verifying the models, or instances of model elements in general. All this information is transformed to a uniform representation and made available to the explanation generation process. Besides, views of the conceptual model may be directly included in the explanations from a view generation component.

Generating explanations, the process needs to consult both information system theories and linguistic theories. Techniques and knowledge from the field of information systems are used to assess the structures of conceptual models and see how these structures can serve as a basis for explanation planning. Linguistic knowledge is needed to find a suitable linguistic realization of the sentences of the explanation. The final explanation will mainly be a natural language text, but graphical illustrative views of the conceptual model may also be included.
A Full-Fledged Explanation Generator

The ultimate goal would be to construct a complete component as shown in Figure 2.3. This would require that a number of tasks were done:

- The complete explanation process must be implemented. In particular, one need to consider how information system knowledge affects the planning of explanations, and how linguistic knowledge governs the realization of sentences included in them.

- We need a conceptual modeling module and modules for verifying, executing, and showing views of conceptual models. To the extent that these modules are missing or imperfect in the environment to start with, we must extend the functionality of the CASE environment.

- An interface for accepting and interpreting explanation requests has to be be implemented.

- A suitable formalism for representing information used in the generation process has to be developed.

- Routines translating various model representations and results from verifying and executing models have to be made. These transformations should make all the necessary information available to the generation process, using the specially developed formalism from above.

- Although not shown in the figure, the explanation component may very well be needed in multi-language environments. To be maximally flexible, it should be
based on representations and principles that are independent of both conceptual modeling language and CASE environment.

Reaching this very ambitious goal would be very difficult within the limits of our work. Each of the tasks can form its own separate project, with its own problems and challenges. Their complexities should not be underestimated, and there are today several projects dealing with the inherent problems of each of them. For instance, in the last few years due attention has been devoted to the development of general CASE environments with execution facilities. And as far as the linguistic part is concerned, there are many projects working on problems in single-sentence generation, and rather few have seen it feasible yet to proceed to the generation of paragraph-length text.

As a consequence, we will not consider this goal realistic for the work to be done here. The intention, however, must be that our results should make it easier to pursue this highly desirable goal at a later stage.

More Realistic Objectives

Our work is to be carried out within the context of existing CASE technology and transaction-based information systems. A major objective is that this thesis can be a first step towards a complete explanation component. Our product should be extendible with additional functionality as well as amenable to a proper evaluation of the ideas underlying it. The concrete objectives of the work are as follows:

- We will develop a suitable uniform formalism for representing explanation-relevant knowledge.
- On the basis of techniques and theories of information systems, we will formulate principles for explanation planning. The result of the planning process must be suitable for a subsequent linguistic and graphical realization.
- The formalism and construction principles should be independent of both conceptual modeling language and CASE environment.

We will assume that all necessary information is available in our representation formalism, and that the linguistic and graphical decisions are separable from the ones planning the explanations. Also, since the interface for receiving explanation requests would be part of a larger user interface for the whole environment, we will assume that a full CASE user interface have to be implemented later and not develop a separate interface just for this project.
In this part, we present the state of the art in conceptual modeling and explanation generation systems, and formulate the requirements to the work.

**Chapter 3** The chapter describes problematic issues in information systems engineering and explains how conceptual models are used in developing information systems.

**Chapter 4** The comprehensibility of conceptual modeling is addressed. Various strategies for enhancing user comprehension are characterized along four dimensions of comprehensibility.

**Chapter 5** The chapter discusses the notion of explanation, its function in communication processes and its desired properties.

**Chapter 6** In this chapter, we present and evaluate tasks, strategies, and techniques found in contemporary explanation generation systems.

**Chapter 7** The detailed requirements to our explanation generation component for conceptual modeling are outlined.
Chapter 3

Information Systems Engineering and Conceptual Models

3.1 Basic Terminology

As an introduction to conceptual modeling, we provide some definitions of basic terms used throughout the whole thesis. These definitions are not meant to be controversial in any sense, and they are all followed by references to works elaborating on the meaning of the terms. They are, however, all motivated from a conceptual modeling’s point of view, and other disciplines may of course prefer slightly different definitions.

Information systems engineering The practice of developing information systems [141, 185].

Requirements engineering The part of information systems engineering that is concerned with the investigation of business areas and user requirements and the construction of implementation-independent descriptions of future information systems [36, 68].

Model A simplified or incomplete representation of an artifact or phenomenon intended to explain some aspects of it or answer a set of well-defined questions about it to a tolerance adequate for a stated purpose [58, 147, 178, 237]. There is a distinction between descriptive models and prescriptive models, of which the first one only describes facts and relationships without forming a basis for evaluation of the system and the second one describes how things ought to be and establishes standards of correctness or prescription of rules [237, 248].
**Language** A vocabulary, a set of rules for composing well formed formulas or models from that vocabulary, and a semantics specifying the interpretation of every meaningful formula or model construction [58]. The language is used for expression of knowledge and belief and/or communication among various kinds of parties [147]. An executable language has an operational semantics, which means that the dynamics of the model can be experienced through execution [224].

**Method** A set of guidelines for modeling with a language (called "way of working" in [242]).

**Technique** A collection of well-defined concepts and a number of guidelines and/or rules for using these concepts to construct models. A technique encompasses a language and a method for modeling with that language [30, 104, 183, 108].

**Methodology** An organized collection of principles, techniques, tools, and procedures to aid the development of information systems [44, 95, 104, 184, 108].

**Model verification** The process of checking whether a model, specified according to a certain technique, conforms to the rules of the technique’s language [108]. These rules are of many kinds, e.g. conformance (syntax) rules, consistency rules, completeness rules, and accuracy rules [33, 137, 228].

**Model validation** The process of checking whether the model correctly and adequately describes a piece of reality and/or the users’ intended requirements [24, 33, 137, 108].

**Stakeholder** Any person or organizational unit involved in or affected by the information systems development project [134]. A user is defined relative to some computer system (e.g. the computerized information system to be built or a tool supporting the development) and denotes a person or organizational unit that might be using services offered by the system. In addition, we need the more specialized notions of end-users (i.e. someone that might use the computerized information system directly), business experts or domain experts (i.e. people knowledgeable in the business area), systems developers (i.e. information systems experts responsible for the development of information systems), and analysts (i.e. systems developers responsible for the construction of various models in requirements engineering).

One central term is missing in the list, and that is "information system" (IS). Even though there have been some attempts to define the concept of information systems and to contrast it to other types of computer-supported systems, there is hardly any well established rigorous definition available. Instead, "information systems" tends to be used rather loosely and with an emphasis on what distinguishes these systems from other kinds of software system.

Informally stated, an information system is understood as a system that has storage, distribution, and manipulation of information as its primary goals (see for example [9, 226]). This, however, is not a particularly concise definition — for example, what is captured by "information" and what does "primary goals" mean — so in practice it is very much left
3.2. Information Systems Development

to the users of the term to draw their own precise terminological boundaries. Another approach to clarifying the characteristics of these systems has been to list all possible types of information system on the basis of their relationship to users and environment, or to focus on the kinds of service and information they provide to their environments [141, 226]. A feature of information systems that has received wide recognition, though, is their inclusion of both manual and automated parts that are highly integrated. When only the automated part is addressed, the term “computerized information system” is usually favored. Also, the information system is assumed to be rooted in someone’s perception of the properties of a real-world system [244].

Our information systems embrace both manual and automated parts and are of the transaction-oriented type. The main emphasis here is put on the transformations of information and the way databases support the functionality of these systems. In Harel’s terminology, they are referred to as “transformational systems” [96].

3.2 Information Systems Development

Seen from the outside, the development of information systems typically includes an analysis of some real-world phenomena, a determination of system requirements, and coordinated work among a number of parties. Several kinds of stakeholders may be negotiating (e.g. end-users, business experts, and analysts) in the process, and the result is a product that guides the successive realization of the computerized part of the system. The whole development process has been referred to as a change process due to its overall goal of replacing existing structures or systems with new ones [155, 157], or — focusing on the intra-dependencies between system requirements and various system representations — as a series of transformations [22].

The activities of development projects can be organized and related through the formulation of process models. The perception of these models tend to vary according to the point of view taken (e.g. a project management’s view of a software development’s view), and they can be classified as activity-oriented models, product-oriented models, or decision-oriented models [122]. As an example of an activity-oriented process model, consider the model shown in Figure 3.1. This is part of a more comprehensive one used by IFIP Working Group 8.1 in their assessment of information systems methodologies [183], but for our purposes we only need to consider the four stages shown here. Each stage denotes an activity, and as a stage culminates, a stage product documenting the results from the stage is made available to the subsequent stage. The stages are:

1. Information systems planning At this stage there is a broad brush analysis of the enterprise’s use and needs of information. The main business objectives and information requirements are identified, and a feasibility study is conducted to determine the possible alternatives for proceeding further. The planning product describes — in business-related terms — which systems, if any, need to be designed and implemented.
2. **Business analysis** The second stage calls for an examination of the existing state of affairs in a given *business area* of the enterprise. The data and activities relevant to the future information system are analyzed, and the models included in the business analysis product are descriptive of nature.

3. **System design** Contrary to business analysis, system design is a prescriptive kind of work. The future information system, reflecting the requirements from the user’s side, is described in terms independent of the tools used to construct the system. The scope of the business area covered at this stage may be more restricted than that of the business analysis stage, though both manual and automated parts may be considered.

4. **Construction design** The computerized part of the information system is now designed on the basis of the hardware and software tools available. This set of tools is called the *construction environment*, and they necessitate a consideration of implementation details in the construction design product.

Although defined as a linear sequence of activities here, there may be loops back from a stage to any preceding stage. If the construction design uncovers something that is clearly a misconception about the business area, for example, one might go directly back to the business analysis stage and resume (parts of) the project from there.

This model is of course not the only one describing the development process as a pattern of activities. Also, the terms used in the model are not necessarily consistent with similar activities or concepts found in other models. To take a few examples, "*business area*" is sometimes referred to as "*universe of discourse*" (e.g. [120, 108, 236, 248]) or "*application domain*" (e.g. [22, 68, 138]), whereas "*system design product*" corresponds more or less to the notions of "*(software) requirements specification*" (e.g. [58, 141, 256]) or
"system specification" (e.g. [235]). For the purposes of this presentation, though, we have chosen to stick to the terms used in the IFIP process model.

3.3 Requirements Engineering

Common to many process models is the clear separation of what the system is supposed to do, as seen from a user's point of view, and how this functionality is to be realized. The former is concerned with the requirements to the information system and leads to a description including those system properties that are deemed relevant to the users' validation and approval. The description is said to address the systems' external behavior [58, 106] and is often referred to as a system specification as opposed to a system construction. The latter is focused on how the system is realized in a specific construction environment, and as long as the requirements to the system are properly specified, the users do not have to be involved here. These two views of information systems mirror the two disciplines of requirements engineering (RE) and design engineering (SE), respectively [36].

The Importance of Requirements Engineering

The importance of requirements engineering has been recognized by many. In [58], Davis has suggested and discussed five hypotheses about requirements engineering that should motivate the work in the field. As indicated by these hypotheses (see Table 3.1), there are good reasons for taking requirements engineering seriously [153, 154]:

- The requirements specified govern the whole project down to the implemented computerized system. If there is an error here, it has profound effects on all subsequent development stages and can be very expensive to fix. Even though requirements analysis consumes merely five percent of the effort in information systems development, it affords about fifty percent of the leverage to influence improved system quality [187].

- Requirements analysis errors can often be very difficult to detect. Whereas design and coding errors are exposed as representations and results are compared, there is usually no clear explicit representations against which the requirements can be tested.

Moreover, as opposed to the situation in design engineering, there are often no good criteria for deciding the correctness of a requirements engineering product [89]. Instead of correct and incorrect requirements, one will have to compare and choose among requirements that are all correct in some sense, and the choices here may lead to quite different information systems.
hypotheses

1 “The later in the development life cycle that a software error is detected, the more expensive it will be to repair.”
2 “Many errors remain latent and are not detected until well after the stage at which they are made.”
3 “There are many requirements errors being made.”
4 “Errors made in requirements specifications are typically incorrect facts, omissions, inconsistencies, and ambiguities.”
5 “Requirements errors can be detected.”

Table 3.1: Davis’s hypotheses about requirements engineering [58].

Some Problematic Issues

Requirements engineering is not only an important task — it is also a rather difficult one. Below is a discussion of some of the more complicated issues in the field.

Independence of Requirements Engineering Viewing requirements engineering as independent of systems engineering has received some criticism. Although desirable for the organization of the project, it can sometimes turn out that features of the construction environment necessitate a revision of the stated requirements and that the best approach is to conceive the two tasks as intertwined. In particular, the available hardware and software in the construction environment restrict the requirements that can be accommodated and can make it too expensive to pursue some of the more challenging desiderata for the system [235].

Separating “What” and “How” According to Davis [58], the aim of requirements engineering is to specify the external behavior of the system without dwelling on how the system is to work internally. The basic idea has been to enhance user understanding and participation by concealing system properties that are not directly observable to system users, but this has turned out to be rather impractical for many requirements. It is not necessarily the case that a requirement is easiest specified in terms of external observable behavior — in fact, many requirements are naturally explained or understood as a procedure of some kind. Besides, the solution to one requirement may lead to a number of new requirements at the next level [225]. In some development approaches (as indicated by the IFIP model), they try to solve this by defining requirements engineering as problem-oriented rather than externally oriented, which means that the product of requirements engineering is extended to include all features of the future information system that are independent of the construction environment (e.g. [256]). This means that the internal behavior can also be considered as long as it sheds light on the requirements and is not tied to any construction environment. Not surprisingly, however, these approaches tend to lead to RE products that are complex and difficult to understand.

Multidisciplinary nature of information system Information systems are employed for an organization to function at a higher level. Technical excellence is important in
this respect, but unless social and organizational considerations are also made, the system may very well fail to provide the desired improvements of the organization [77, 141, 156].

**Uncertainty, volatility, and subjectivity of requirements** Another complication in requirements engineering is rooted in the whole nature of information systems. Their environments might be dynamic or turbulent, which complicates the process of acquiring a consistent, stable, and reliable set of requirements. Sometimes, the information system itself may even change its environment, producing an ever-lasting cycle of requirements revision and information system modifications [141, 226]. Besides, as the environment often reflects the conflicting views or interests of organizational units or people, requirements tend to be subjective and inconsistent, and the approach taken to resolve the conflicts may give vastly different interpretations of the problems at hand [22, 105, 156]. In Balzer’s opinion, the assessment of requirements is the hardest part of information systems engineering [119].

**Representation of system and requirements** It is normally impossible to specify all the properties of an information system completely [22]. The recorded requirements, thus, are restricted to what is considered relevant for the stakeholders’ discussion and use, and several checklists for expressing these requirements have been suggested (e.g. [58, 204, 255]). However, many requirements are hard to specify or relate to other requirements in intricate ways, and there is no firm criterion for what level of detail the requirements should be specified at. Apart from this, the variety of requirements involved and the span of use of these requirements have made it difficult to provide suitable means for representing and presenting RE products [122].

Developing information systems is a complex and challenging task. The conception of requirements engineering and design engineering has given rise to some optimism, but there is still a long way to go until all intricacies of the process have been sorted out. Even though the importance of requirements engineering is now generally accepted, there are some very problematic issues hampering the effective assessment of requirements to the systems. In his widely debated article, “No Silver Bullet: Essence and Accidents of Software Engineering” [32], Brooks identifies the essential difficulties of software engineering as the complexity, conformity, changeability, and invisibility of software. These difficulties distinguish software engineering from many other engineering disciplines, and in Brooks’s view they are of a character that make them extremely problematic for humans to cope with. His arguments, which applies equally well to information systems engineering, make him conclude that no extant approaches to software engineering can be assumed to yield any substantial improvements of the discipline. Others have later opposed this rather gloomy view of software engineering and information systems engineering (e.g. Blum [22], Harel [98], and Lowry [153]), but the prevailing perception of requirements engineering and design engineering as very demanding tasks is still likely to dominate the research in the years to come.
3.4 Conceptual Modeling

The Notion of Conceptual Models

The notion of conceptual models is closely related to the analysis of real-world phenomena. The concepts underlying these models are assumed to represent real-world entities and activities [25, 26], and the models are said to be constructed at the level of human concept formulation [255]. The domain for the models, thus, is some universe of discourse or — using the term introduced in [183] — some business area.

As pointed out by several authors [89, 137], conceptual models are not unique with respect to a specific business area. Rather, they depend on the users' understanding of the area and consequently may vary according to the business knowledge, backgrounds, and attitudes brought into the modeling process. In recognition of this, Kung considers a conceptual model to be a model of a business area as perceived by the user's community and the development team [138]. A similar definition is also suggested by Wieringa [248].

However, there has been some divergence of views about the conceptual model being restricted to only real-world phenomena. As part of the specification of some future computerized information system, the model must also be able to serve as a requirements model [237] or as a system design model independent of the construction environment and any implementation concerns [25, 58, 183]. This broadened interpretation of conceptual models is the one adopted in most integrated tools for information systems development (e.g. [87, 98]).

In our work the term "conceptual model" will be used for both descriptive representations of business areas or prescriptive construction-independent representations of intended information systems. In both cases, though, the models are assumed to reflect the perception of people involved in the modeling process. With reference to the model introduced by IFIP WG8.1 (see Figure 3.1), our understanding of conceptual models relates it to the business analysis stage as well as to the system design stage.

As a consequence, a conceptual modeling language is said to be any language that can be used to construct conceptual models. We do not distinguish between "modeling language" and "specification language" (as is done in [137]), and the languages can differ in perspective (data, process, and behavior [107]), style (e.g. declarative, deductive, operational [182]), abstraction level, purpose (communication, verification, prototyping, etc.), and other aspects. Though, we assume the languages to have a basically graphical syntax.

Conceptual Models in Requirements Engineering

Conceptual models included, the development of information systems embraces three realms that have to be considered [137]:

Figure 3.2: Requirements engineering and design engineering (adapted from [202]).

- the real-world realm, which involves the business area relevant to the information system;
- the conceptual realm, which corresponds to the conceptual model constructed to analyze the business and specify requirements to the information system; and
- the datalogical realm, which is concerned with the representation and use of data in a computerized information system.

As indicated in [22, 202], these three realms are closely related to the activities of requirements engineering and design engineering (see Figure 3.2). At the requirements engineering stage, phenomena grounded in the real-world realm are abstracted and give rise to the conceptual model of the conceptual realm. The conceptual model is then guiding a reification process leading to a computerized information system at the design engineering stage.

Four roles of conceptual models can be identified in these activities [64, 122, 137, 248]. The conceptual model is a representation of system and requirements, a vehicle for communication, a basis for construction and implementation, and a documentation of the system. In the following, each of the roles are discussed with respect to their influence on the requirements engineering process and the desired properties of the conceptual modeling language.

**Representation of system and requirements** The conceptual model represents properties of the business area and requirements to the system as perceived in the real-world realm. According to Borgida [25], the whole information system can be viewed as a model of the real world. The conceptual model gives insight into the problems motivating the
development project, and enables the analysts to have a better understanding of the system that is to be built. Moreover, by analyzing the model instead of the business area itself, one might deduce properties that are difficult if not impossible to observe from the real world directly. From an information systems engineering's point of view, these models smooth the transition from real-world phenomena to representations of the datalogical realm.

Vehicle for communication The conceptual model serves as a means for communication among stakeholders. Bridging the real-world realm and the datalogical realm, it opens for a more reliable and constructive exchange of opinions between users and information systems developers, and between different kinds of users. The importance of this communication for user participation has been underscored in many recent works (e.g. [20, 69, 138]).

Basis for construction and implementation A prescriptive model, approved by the users, specifies the desired properties of the information system. When constructing and implementing the computerized information system, the relevant parts of the model is guiding the development process. Similarly, the construction and implementation may afterwards be tested against the model to make sure that the representations be consistent. Tool support can be automatic for these transformations and analyses, in which case the design engineering part of information systems engineering projects may be improved by an order of magnitude [153].

Documentation The conceptual model is an easily accessible documentation of the information system. Due to its independence of the construction environment, it is less detailed than other system representations while still representing the system's basic functionality. Compared to manually produced textual documentations, the conceptual model is easier to maintain since it is naturally constructed as part of the process of developing the program system. With the introduction of more flexible methodologies and tool support, the conceptual model is also likely to be used in reengineering tasks, reverse engineering tasks, and when reusing components of previously built information systems [122].

3.5 The Dual Nature of Conceptual Models

As seen above, conceptual models serve several roles in information systems engineering. For the models to be effective in these roles, however, they are required to be of a certain quality. This quality can be formulated as a list of requirements to conceptual models that concerns both the modeling language used and the way the modeling is carried out. Summaries of such lists are found in [58, 89, 137, 204, 255].

The fact that conceptual models are given both communicative and representational roles is pronounced in the requirements put forward. As a consequence of this diversity of
roles, some of the requirements pull in opposite directions, and it is questionable whether there will be a conceptual model that can meet all these requirements by its own. What is good for the model’s abilities to aid the communication among humans can impair its representational properties, and vice versa.

*Comprehensibility* in conceptual modeling refers to the stakeholders’ ability to understand the modeling language and correctly read the models constructed. It is important for the process of modeling, since it allows people not skilled in modeling to participate, and speeds up the use of the modeling language in general. When it comes to the product of modeling, it gets easier for users to validate the models and suggest improvements. In relation to the roles presented above, comprehensibility affects the use of conceptual models as means for exploring the business area and exchanging information and opinions among stakeholders.

From a more detailed perspective, maximum comprehensibility in conceptual modeling implies — among other things — that the modeling language should

- have concepts close to the users’ perception of business area phenomena,
- have concepts for representing (1) vague, abstract, or informal knowledge, (2) certain, detailed, or formal knowledge, and (3) relationships between these two classes, and
- have an associated method that mirrors the natural way to explore or describe a business area.

The *representational* role of conceptual models says that it should be able to represent all necessary information in a notation suitable for its use. That is, all the relevant aspects of the business area should be representable, and the representations ought to be amenable to various kinds of tool-supported transformations, analyses, and checks. This tool-support can effectively improve the transition from requirements engineering to design engineering, but it is also useful in assessing business area and user requirements. Models can be verified automatically, and using tool-provided techniques like model translation, execution, abstraction, and view integration, we are assisted in the validation of these models. If the necessary representational power is present, thus, the tool may offer functions that also help us understand the conceptual models correctly.

The representational role of conceptual models is supported if the language is

- expressive (concerns both power of expression and resolution of detail),
- verifiable (consistency, internal completeness, etc.),
- independent of construction, and
- semantically formalized.
From this analysis, it should be clear that both comprehensibility and representational suitability are important in conceptual modeling. Looking at the requirements listed above, however, it seems rather difficult to meet all the requirements motivated from these two concerns. Many of the requirements promoting model and language comprehension can be attributed to \textit{informality} and \textit{naturalness}, whereas the representational requirements are oriented towards \textit{formality} and \textit{expressiveness}. To take an example, if the concepts of the modeling language are vague or informal, it may prove difficult to fill the roles attached to the model’s main task is to bridge the realms of real-world phenomena and datalogical artifacts, the model is just required to fill its assigned roles to be useful.

There are, however, several strategies for dealing with this dual nature of conceptual models:

1. \textbf{Ignore either comprehension requirements or representational requirements}
   This strategy has not been considered very auspicious \cite{73}. Ignoring representational suitability or modeling comprehension, we get: a language that is not able to fill the roles attached to it. As the model’s main task is to bridge the realms of real-world phenomena and datalogical artifacts, the model is just required to fill its assigned roles to be useful.

2. \textbf{Trade-off between requirements}
   Another strategy is to try to balance the comprehensibility of conceptual models with a certain degree of formality and expressiveness, as is done in Harel’s visual formalisms \cite{97}. On a firm formal basis, one attempts to devise a syntax and a semantics of the language that still make the model understandable to the users. A major obstacle in this respect, though, is the inherent complexity of information systems, but introducing various abstraction mechanisms one has been able to cope with at least some of these complications.

3. \textbf{Specially selected people}
   A tempting strategy is to select only very skilled people to join the development team. People that cannot be assumed to understand the formal powerful modeling language used, are simply ignored and left out of the project. However, since the stakeholders are so different in background, responsibilities and skills, this strategy would inevitably lead to the exclusion of important people from the project. It would be difficult to have a proper user participation, which is by many considered paramount for the project’s success \cite{16, 79, 192}.

4. \textbf{Training of stakeholders}
   Prior to the modeling task, one could allow a training period for people not familiar with the modeling environment. Although this is both possible and desirable in some situations, it is hardly a satisfactory solution for systems development in general. It might get very expensive and time-consuming to do this, and unless the people need the knowledge for later occasions, both the profitability and motivations from the stakeholders’ point of view may be doubtful.

5. \textbf{Compensate for lack of comprehension}
   Using this strategy, one chooses a powerful formal conceptual modeling language that enable the use of suitable tool-supported transformations, analyses and checks.
3.6 Modeling Environments

The tool support is meant to expose properties of the model that—due to the model’s complexity or unfamiliarity—are difficult to see from reading the model alone. To take some examples, model execution, verification checks, and various report facilities can all be useful in interpreting the content of a complex formal conceptual model (see for example PAISley [256], RSF [61], and Telos [176]). However, the results from tool-supported functions are also represented in some way or another, so if this strategy is to be successful, the results must be given in a form understandable to the users.

6. Distinguish between representation and presentation

At last, there have been attempts to separate between the representation and the presentation of conceptual models (e.g. [19, 129, 202, 232]). The representation is given in a formal and expressive language, whereas the presentations are assumed to be user-tailored both in notation and complexity.

In sum, the strategies 2, 5, and 6 are considered to be the most promising. Interestingly, these three strategies can all be supported in the same tool environment, which gives us a combined approach to this important problem in conceptual modeling. As a basis, one would then introduce a formal modeling language with features that could help the users read the models, like graphical symbols or powerful abstraction mechanisms. On request, the tool could transform the models to a language with which the user is more familiar, or hide parts of the model that is not considered relevant to that particular user. Tool-supported verification checks, analysis and execution techniques would reveal aspects of the model that are awkward to detect manually from complex models. Environments inspired from this philosophy are found in [87, 99, 129, 201, 250], among others.

3.6 Modeling Environments

The Process of Modeling

In a top-down modeling approach to information systems development, the information system is built from a series of increasingly detailed and focused models. By starting from an analysis of a business area, i.e. an unsatisfactory real-world system, a conceptual model is constructed and gradually refined to assess the problems and needs of the system. In the beginning, both information system and environment may be included, but as the final conceptual model is finished automation boundaries are introduced into the model. These decide which parts of the system are to constitute the computerized information system, and they guide the subsequent construction and implementation stages. From a modeling perspective, we can view these stages as a continuation of the design stage, in which construction details are added to realize the behavior of the design model. Having implemented the computerized information system, we put it into operation in the larger real-world information system. The whole process is illustrated in Figure 3.3, which is adapted from [90].
With reference to Figure 3.3 and the description above, the following elaborations of the modeling cycle should be noted.

**User Involvement** Information systems development is often viewed as a kind of negotiation process. Business experts provide real-world knowledge and explain their intentions for the information systems to be developed; systems developers — with their knowledge of information systems — help the business experts specify their requirements and provide solutions to their needs. A successful project requires a matching of users' needs and system developers' opportunity to satisfy these needs within the resources available.

**Model Transition** A new (version of a) model is created in one of three ways: (1) The developers are adding information to an existing model to make it more complete or precise, (2) a model is transformed — either automatically or manually — from another representation, and (3) previously constructed submodels are included as part of a larger model. The approach, thus, opens for extensive tool support as well as for the inclusion of reuse or bottom-up substrategies.

**Circular development** As indicated in Figure 3.3, the whole modeling cycle may be iterated several times for the users to be satisfied. This is the case, for example, if the computerized system developed is just a prototype of the final system. The prototype's task would be to clarify what is required of the new information system,
3.6. Modeling Environments

thereby sparking new iterations of the cycle. In addition, after the new information system is put into use, new requirements may emerge and the system has to be maintained or replaced by a new one, which also results in new modeling cycles.

The modeling approach is adopted by many environments supporting information systems development. Conceptually clean, it puts emphasis on the importance of various kinds of models and their integration, without loosing the overall picture of real-world systems and information systems development. It may be argued that the systems engineering part is rather simplistically presented, but as our focus in on requirements engineering, we will not be too concerned about that here.

The Product of Modeling

At various stages of the modeling process, the quality of the models constructed are assessed. In the conceptual modeling part, the notions of verification and validation are especially important.

There are several types of verification checks. Common to them all is their independence of human involvement and their focus on formal properties of the modeling language used. Rules used in these checks are conformance (syntactic) checks, consistency checks, completeness checks, and quality (accuracy) checks [228, 253]. If a general domain model is represented in the environment, one might also check the conceptual model’s completeness or consistency with respect to this model [253]. As indicated in Figure 3.2, one can in some cases also verify that a design or an implementation is consistent with the conceptual model.

Verifying a model, one makes sure that the model be formally correct — that is, all the necessary model elements are included, and no elements are in conflict. During a modeling session, however, it is not desirable to keep a model in accordance with all the rules at all times. For example, there is a completeness rule saying that all processes in a data flow diagram must produce at least one flow. Since it is not very practical to draw a process and a flow at the same time, this rule is automatically broken as a new process is introduced in the diagram. On the other hand, the conformance rule saying that two stores in a data flow diagram cannot be directly connected by a data flow, is usually enforced throughout the whole modeling session. In practice, then, one distinguish between two types of verification rules [242]:

A posteriori rules are rules that can be occasionally broken, like the first completeness rule above, but that has to be satisfied when the model is supposed to be finished. If supported by a tool, the rule is invoked by the user when she feels that (parts of) the model is ready.

A priori rules are rules that have to be satisfied at all times during the modeling process, like the conformance rule above. A good editor would in many cases prevent the user from building model constructions that violate a priori rules.
Validation of conceptual models requires that the users understand the models and their consequences for the behavior of the implemented systems. If the models are difficult to read and understand directly, various validation techniques are used to expose properties or aspects of them. These techniques may change the presentation of the models (e.g. the model’s language, perspective, medium, or layout is altered), user-tailor their levels of detail (e.g. produce abstractions), or reveal properties implied by them (e.g. executions, reachability checks and deadlock checks).

A model — or a part of a model — is usually validated after it has been verified, at a time when it is assumed to be meaningful to the users. Looking back at Figure 3.2, we see a small modeling cycle between business area and conceptual model that explains the close relationship between modeling and validation. Whereas analysis and modeling bring us from phenomena in the real world to constructions of the conceptual model, validation is performed by comparing these constructions to what they were meant to represent. The results from the comparisons are inputs to new cycles of analysis, modeling, and validation, and the process is continued until the model is deemed to be a satisfactory representation of the business area in question.

Possible Models Involved

In addition to process models and conceptual models of existing or future information systems, a development environment may often make use of explicit representations of meta models, user models, and user interface models. These models are briefly described below, as they will be referred to and used later in the thesis.

Meta model A meta model defines syntactic and/or semantic aspects of a modeling language\(^1\). It is usually represented using structures techniques (e.g. [30, 223, 227, 108, 242]), knowledge representation techniques (e.g. [129]), or formal grammatical or mathematical techniques (e.g. [3, 97, 80]). Most meta models so far focus on syntactic properties, but also the language’s layout and semantics have been modeled [80, 129, 223, 108].

User model The term “user models” has been used to denote various kinds of models [27, 130, 175, 178, 199]. Some of these models are represented by computers, others by the users themselves (mental user models), and even others by designers of future information systems. We will here only consider the user models that computers keep.

A user model in information systems engineering specifies properties of a user or user group. The properties include the user’s knowledge, role, background, interests, preferences, etc. — that is, all differences among users that might be relevant to some purpose. User models are represented as simple predicates, production rules, frames, or conceptual models. They can be classified as stereotypical or individual.

\(^1\)Brinkkemper defines two types of meta models, “meta-data models” and “meta-activity models” [30]. Our use of “meta model” corresponds to the term “meta-data model”.

3.6. Modeling Environments

implicit or explicit, short-term or long-term, static or dynamic, predictive or descriptive, individual or stereotypical, and inferred or specified [27, 175, 178, 199], and are found in environments for information systems development, help systems, tutorial systems, user interfaces, text generation systems, among others.

User interface model This model specifies aspects of an information system's user interface. Some user interface models specify only the control flow of the interface, whereas others include specifications of the actual objects used in the interface [102].

In [176], it is argued that four distinct subworlds have to be modeled during the development of an information system: (1) the subject world (i.e. the domain about which the information system is to provide information), (2) the system world (i.e. the information system itself), (3) the usage world (i.e. the environment's use of the information system), and (4) the development world (i.e. the process model). This extended modeling philosophy, constructing and relating different kinds of models, has also been supported by others [42].

CASE Environments

Software environments that assist the systems developers in the process of developing information systems are often referred to as CASE environments² [33, 60, 173, 229]. These support a variety of development tasks and manipulate different kinds of representations relevant to the information system being developed [157, 162].

In some terminologies, the term "CASE environment" is given a rather broad interpretation, embracing what has traditionally been called analyst workbenches, code generators, re-engineering environments, environments for strategic planning and enterprise modeling, life cycle products like Ipses³, and repositories. A distinction is then often drawn between upper CASE, which cover the early stages of information systems development, and lower CASE environments. However, more rigorous definitions of CASE environments have been put forward to pinpoint their advantages to earlier environments for software development. In [157], a CASE environment is required to support activities and representations at several stages of the development project. Mimo [173] adds the requirements that CASE environments must be interactive, must provide automatic analyses of model representations, and must be able to generate high-quality code from complete information system models. In the presentation here, though, we will focus on the conceptual modeling part of CASE environments and will not be too concerned with the exact definition of the term.

With respect to conceptual modeling, there are a number of functions or features characterizing the environments. These include methodology support, graphical user interfaces, information repositories, verification checks, execution facilities, various kinds of model presentations, code generators, facilities for re-engineering or reverse engineering, and

---

² "CASE" is an acronym for "Computer-Aided Software Engineering".
³ "Ipses" is an acronym for "Integrated Project Support Environment".
project management [11, 33, 42, 60, 173, 180, 241]. In Appendix B, an experimental CASE environment exhibiting many of these features is presented.

The last few years, commercial CASE environments have undergone major improvements, and there are research projects working on even more powerful environments for information systems development. Some key words for this research are integration, co-operative development, and meta environments.

**Integration** Both model representations and tools included in the environment could be more closely integrated than what they are today [121, 173, 180, 241]. There are now research environments, in which the relationships between representations at the same development stage as well as between stages are formally defined (horizontal and vertical integration, respectively) [87, 99, 123, 129]. Along the same line, Brinkkemper discusses the need for modeling transparency in CASE environments [31]. And in [41, 238], tool integration is proposed on the areas of data, control, presentation, and process. In the resulting CASE environment, the integrated CASE environment, transitions between development stages are supposed to be smoother, the user interface more consistent, and the tools included would cooperate more effectively and efficiently.

**Co-operative development** With the increased emphasis on user participation, the environment's ability to support team work has been considered paramount [42, 132, 180]. The research on this topic has lead to new methodologies (e.g. [9, 134]) and the active use of views of the business area and the conceptual model (e.g. [62, 75, 144, 179]). In general, the co-operative development approach is intended to improve information systems engineering by making the development process more understandable and transparent.

**Meta environments** In a meta environment⁴, methodologies can be altered or replaced to suit the needs of the particular development project. Modifiable meta models define the languages used in the environments, and the developers are free to build their own CASE environment using these models. Examples of meta environments are Toolbuilder [3], ALF [186], Meta view [227], and Ipsys [11]; in [223], the MetaEdit environment for meta modeling is presented.

The functionality indicated above is hardly achievable without a firm linguistic and modular basis. In particular, the conceptual modeling languages have become increasingly complex and formal for the environment to offer the desired functionality. When the conceptual models are given a rigorous semantic interpretation — and are sufficiently expressive — knowledge-based techniques can be employed to manipulate, analyze, and transform the representations. According to [13, 44, 153, 217, 241], the future of CASE environments is exactly the exploitation of knowledge-based techniques on formal model representations.

⁴ "Meta environments" are also referred to as "meta CASE" or "CASE shells".
Chapter 4

Comprehension in Conceptual Modeling

The importance of comprehension in conceptual modeling was discussed in the previous chapter. There, it was also indicated how the employment of formal modeling languages and the construction of large, complex conceptual models impair users' abilities to understand the whole modeling process.

In this chapter, we discuss four dimensions for characterizing the comprehensibility of conceptual modeling environments. We briefly describe various existing strategies for enhancing user's comprehension, and discuss our explanation generation approach using these four dimensions.

4.1 Characterizing Comprehensibility

A conceptual model can be difficult to comprehend due to the formality or unfamiliarity of the modeling language used, the complexity or size of the model, or the effort needed to deduce important properties of it. A conceptual modeling environment may make use of certain strategies to enhance user's comprehension. Looking at the linguistic aspects of conceptual modeling, we can describe such strategies along the following four dimensions:

- **Language perception** concerns user's ability to understand the concepts of the modeling language.

- **Content relevance** indicates the possibilities of separating between irrelevant and relevant model properties, so that at any time one is able to focus on just the relevant parts.
• **Structure analysis** depends on the environment’s abilities to analyze and expose structural properties of the conceptual model.

• **Behavior experience** is related to the model execution facilities offered.

A strategy provides an improvement to one or several of these dimensions, thereby enhancing the comprehensibility of conceptual modeling. There are other strategies that might have an indirect effect on this comprehensibility. Instead of focusing on the modeling language used, these strategies are rather independent of the language and more concerned with the methodological approach to conceptual modeling. For example, one might extend the modeling scope so that the model would also describe the interaction between user and system (e.g. [245]) or the system’s contribution in performing organizational tasks (e.g. [158, 159]). Or one might construct several independent conceptual models and deduce properties of the complete one by merging these views (e.g. [144]). From a methodological point of view, there are today several theories about how organizational analyses and task analyses could initiate development projects and enhance user participation and understanding (e.g. [9]). In general, these strategies may enhance a stakeholder’s interests in and commitment to the project, thereby helping her to actively take part in the conceptual modeling process. However, as our focus here is more on the modeling language part than on the methodological part, we will restrict ourselves to strategies dealing with the linguistic aspects of conceptual modeling.

In the following, we will describe the four dimensions above in some more detail, referring to strategies found in existing environments for conceptual modeling.

**Language Perception**

The modeling language can be given a definition that enhances user comprehension [25]. This concerns both its syntax and semantics, though it is not very clear what makes it user-friendly modeling concepts [89, 222].

An interesting approach here is the inclusion of user-oriented and user-defined language concepts. To take some examples, new modeling concepts can be introduced using *class concepts* in Telos [176] and *parameterized concepts* in ARIES [129], whereas Davis advocates an approach in which an application-oriented (and user-oriented) syntax is defined on top of an executable state transition formalism [57].

Another approach is to improve the layout of conceptual models by introducing graphical symbols and special routines for displaying these graphical models [148]. In Figure 4.1, for example, the graphical BNM model [138] in (b) represents about the same information as the rather complex Gist model [15] in (a): *A customer, which is characterized by the attributes name, classification, and income, can request any number of loans, and a loan is given the attributes number and amount.* Visual presentations have been well received in conceptual modeling environment and graphical modeling languages are today found in most experimental and commercial CASE environments [11].
4.1. Characterizing Comprehensibility

\[
\begin{align*}
\text{type customer(} & \text{Name | name, Classification | classification,} \\
& \text{Income | income, Request | loan :any :unique);} \\
\text{type loan(Amount | amount, Number | number):}
\end{align*}
\]

(a)

\begin{figure}[ht]
\centering
\includegraphics[width=0.8\textwidth]{figure4.1.png}
\caption{(a) Part of a textual data model in Gist. (b) Part of a graphical data model in BNM.}
\end{figure}

One may also start out with rather informal descriptions of information systems and translate these to formal models as the projects go on. The simplicity and/or familiarity of the informal descriptions makes them an attractive vehicle for requirements elicitation and communication among stakeholders, though the formal representations produced afterwards may not be very easy to read. In ALECSI [36, 202], a natural language description is automatically translated into a simple semantic net using a set of standardized sentence patterns. VDM specifications have been developed from structured analysis (SA) models [76], and TELL (a language based on temporal logic) specifications from larger bodies of NL text [208].

Distinguishing between the representation and the presentation of conceptual models, one can translate the models to languages that are easier to understand for the user at hand. From the Gist specification in Figure 4.1(a), for instance, the following paraphrase can be presented to the user [232]:

"There are customers and loans. Each customer has a name, a classification, and an income, and requests any number of loans. A loan is requested by a customer. Each loan has a number and an amount."

Similar paraphrasers are found in ALECSI [202], PRISMA [179], and ERAE [68], among others.

Translations from one modeling language to another can be useful on special occasions. On the one hand, the reader may request a model presentation to be given in a familiar language. On the other hand, the purpose of inspecting the model may influence the perspective that should underly the presentation, which in turn may necessitate a model translation. Facilities for translating between formal languages are today found in a variety of systems (e.g. [62, 128, 179]).
Content Relevance

A hypothesis common to most theories in linguistic communication is that the relevance of the communicated message affects the receiver’s abilities to understand it (see for example [231]). Along the same lines, increasing the relevance of the parts of the conceptual model presented to the user is assumed to increase the model’s comprehensibility [215].

A complexity-reducing technique is used to present views of the total conceptual model, in which irrelevant details are suppressed and relevant details are highlighted. Usually, one distinguish between two kinds of complexity reduction [215]:

- Model constructions can be combined to form simpler higher-level constructions that can be used instead of the original ones. Normally, this is done using the four traditional abstraction mechanisms (generalization, aggregation, association, and classification [70, 222]), and the techniques, thus, depend on the hierarchical concepts employed by the modeling language.

- Given a rather complex conceptual model, its complexity can be reduced by simply hiding those elements that are considered irrelevant in the context without introducing any new higher-level concepts. This strategy is not really language-dependent, but is more of a feature of the supporting modeling environment.

Selvveit [215] refers to these two kinds as vertical abstraction and horizontal abstraction, respectively. In Figure 4.2, horizontal abstraction is used to generate a view of a PrM diagram from Appendix A. Here, only control aspects are deemed relevant, and data flows and stores are therefore suppressed.

It should also be noted that the choice of presentation language and graphical symbols may influence the perceived relevance of the models (e.g. viewing systems from different perspectives [107]).

Structure Analysis

When modeling language or conceptual model gets complex, reading and interpreting the model can necessarily be difficult. Offering functions that analyze and expose aspects
of it, like verification checks and optimization guidances, is often useful when specific model properties are to be assessed [99, 228]. As such, the available structure analyses in the environment can potentially improve the comprehensibility of a conceptual modeling process.

The formal approach to conceptual modeling is considered paramount with respect to this dimension [22, 138, 98]. Described informally, a formal modeling language is defined on a firm mathematical basis and provides a well-defined rotation (syntactic domain), a universe of objects (semantic domain), and a precise rule specifying the mapping between these two domains. Employing such a formal modeling language, the CASE environment can support the modeling with a wide variety of structure analyses. These include verification checks (e.g. [209, 228, 253]), reachability and deadlock tests (e.g. [99]), tests for model simplifications (e.g. [4, 236]), and deductions of properties and behaviors (e.g. [61, 68]). For example, STATEMATE can check that all the states in the model in Figure 4.3 can be reached, and that no deadlocks can arise during execution.

A more thorough introduction to formal languages can be found in [137, 172, 251], whereas their potential in modeling environments is discussed in [13, 98].

Behavior Experience

Interpreting the dynamic properties of large information system models is demanding. The comprehensibility of these models is in many cases assumed improved if they can be subject to some kind of execution [32, 98, 138]. Exercising a model or an executable
representation generated from the model, the user can experience its dynamic properties in a more direct and tangible manner.

Central to this dimension is the notion of executable or operational modeling languages [58, 224, 257]. Characteristic to these languages is that their models can either be directly executed or transformed to a representation that is executable [98]. The conceptual model of the system, then, becomes a simulation model generating behaviors of the specified system [256]. Various kinds of execution are possible, like step-by-step execution, batch execution, programmed execution with break points or spy points, animation, or symbolic execution [45, 49, 98, 151, 249]. Animation is a special technique that requires a graphical conceptual model, so that the execution path can be shown by blinking or highlighting elements of the model (see for example [23, 140, 239]). Also, in some systems, an execution trace can be inspected during or after an execution (e.g. [23, 99, 151]), though the traces are often very detailed and complex to read [90].

If the Statechart in Figure 4.3 were supplemented by the appropriate Activity chart, it would be executable in the STATEMATE environment. The user would provide external events like request-loan and register, whereas internal events like registered and not-registered would come from the activity chart. During execution, active states of the diagram could be highlighted on the screen.

In general, one need not use an executable language to provide executable representations of the future information system. Prototypes consistent with the conceptual model can also be built manually or by means of semi-automatic translation algorithms [35, 257], and these are of course just as useful in exposing dynamic model properties [81, 103].

### 4.2 Combining Strategies

Many CASE environments have adopted an integrated approach to user comprehensibility in conceptual modeling. Several strategies are combined, and these can address all the four dimensions discussed above. A combination that has been embraced the last few years is the use of formal executable languages with vertical abstraction mechanisms and graphical notations, which is now found in PROTOB [12], STATEMATE [99], Teamwork/ES [23], USE [245], among many others. Routines for presenting conceptual models in natural language or in some other formal language are also implemented in some larger modeling environments (e.g. ARIES [129] and PRISMA [179]). In the PPP environment presented in Appendix B, a graphical executable modeling language is used in the conceptual modeling process. Some vertical abstraction mechanisms are included, and prototypes can be generated from the PPP conceptual models.
4.2. Combining Strategies

Explanation Generation

Commercial CASE environments with extensive explanation facilities are not available today. There have been some few attempts to use explanation generation techniques for some very simple representations, but this is still very much at an experimental stage [55, 124]. In principle, though, explanation generation seems to be well suited for integration with other comprehension-enhancing strategies. The technology may form a front-end to other strategies, presenting the results in user-tailored terms and structures. With respect to the four dimensions presented earlier, the following properties of explanations are envisioned:

**Language perception** Explanations are given in terms and structures tailored to the intended user and explanation context.

**Content relevance** Explanations only describe those parts of the conceptual model or execution trace that are relevant to the user. Also, complexity-reducing techniques are used to generate views illustrating the textual parts of the explanations.

**Structure analysis** Explanations can explain the results of structure analyses in a user-friendly and uniform manner.

**Behavior experience** Explanations can justify the inputs and explain the outputs during model executions.

In [44], it is claimed that "the natural-language approach helps the IS designer to communicate more naturally with the user and thus encourages more active user participation." Ryan [207] is more concerned with the integration of natural language generation with other strategies for enhanced comprehension. As he sees it, the technology should be supplemented with graphical presentation and model execution, and should also be extended to provide a question and answer facility to conceptual models. This is, of course, exactly what explanation generation is all about.
Chapter 5

Explanations in Communication

In this chapter, we discuss the characteristics of explanations, their types, structures and representations. Also, a basic terminology from text linguistics is introduced, followed by some standards for successful and reliable communication.

5.1 Basic Terminology for Communication

Basically, a communication session involves two parties, a sender and a receiver. The sender, who has the intention of communicating something, constructs (codes) a message and makes it available to the receiver. Interpreting (decoding) the message, the receiver’s cognitive state is changed; and hopefully, the receiver’s interpretation complies with the sender’s intended message. We will in this work use the following terms for describing such communication processes (see Figure 5.1):

**Message** The message communicated can be given in any medium. From our point of view, though, we will concentrate on purely textual messages and messages combining text and graphics.

**Source model** This is the sender’s knowledge of the topic, which can be both mentally and explicitly represented. Selected portions of the model constitute the content of the message.

**Communicative goal** The sender’s communicative goal is to achieve some cognitive effects on the receiver, like "make the user understand the concept of ‘process’". Other terms used here are *intentional goal* [163], *interpersonal goal* [114], and dis-
course goal [188].

Communicative act Stated rather informally, communicative acts can be regarded as actions that are performed, either intentionally or conventionally, by providing the text [59]. The acts reveal the sender’s intention of communicating the text, like informing about something or requesting some information, and are usually recognized by the receiver. In fact, according to some linguistics, true communication requires that the sender’s communicative intention be mutually known[212] or manifest to the receiver[231].

Communicative effect The effects on the receiver depend on how the receiver interprets the message, and are the results of complex cognitive and perceptual processes.

Discourse features These are features describing the conversation setting and the parties involved, like receiver’s knowledge, opinions, and expectations. Hovy’s claim is that these features may all be relevant in deciding what to include in a text and how to say it [114].
5.2 Quality of Communicated Messages

Starting with a textual message in natural language, we can outline a number of quality criteria and see how this message can be improved by means of graphical or formal language additions.

Textual Communication

Textual communication is restricted to the use of abstract and linguistic communicative acts. The linguistic acts include expressions like informing about some event or state of affair [38, 163, 174, 188], requesting some information [38, 163], warning about something [163], conceding that something is the case [163], recommending that something be done [174], or directing someone to do something [166], and are used by more abstract acts to achieve some communicative goals. The acts characterize the illocutionary force of one single utterance, so a text will usually be realized through a sequence of illocutionary acts.

A text, though, is not an arbitrary sequence of utterances. When looking back at Figure 5.1, it should be clear that texts are based on a wide range of decisions. To decide whether a text is really communicative, one has introduced the notion of textuality. De Beaugrande & Dressler [59] define a text as a “communicative occurrence that meets seven standards of textuality”. The communicative function of texts are essential, and only communicative texts are in their opinion complete texts\(^3\). The seven standards of textuality are cohesion, coherence, intentionality, acceptability, informativity, situationality and intertextuality.

Cohesion is syntactically determined and concerns the way the actual components of the surface text are mutually connected. These components may be dependent on each other, according to grammatical forms and conventions, and cohesion is used to account for how the text meets these grammatical dependencies. Among the most used constructions to achieve cohesion are junctions (conjunctions, disjunctions, contrajunctions and subordinations), proforms and temporal relations.

As noted by [21], cohesion is a surface phenomenon, and cannot guarantee the textuality of the text alone.

Coherence is connected to the configuration of concepts and relations that underlies the surface text, and describes how these are mutually accessible and relevant. As opposed to cohesion, coherence concerns the real content of the text, the textual world, and as such it is not necessarily reflected in any surface component of the text. Instead, coherence is present through semantic relations between utterances and between parts of an utterance.

Usually, the concepts included in a coherent text fall into two main categories: primary and secondary concepts. Primary concepts identify the deep structures re-

\(^3\) Actually, in their work non-communicative texts are called "non-texts".
flected in the topic of the text, and coherence is assessed on the basis of how the sec-
ondary concepts relate to these topical structures. In [59] objects, situations, events
and actions are listed as candidates for primary concepts; the secondary concepts
describe properties of primary ones and count 34 in number (e.g. reason, purpose,
and location).

Neither cohesion nor coherence takes the whole situation of communicating into
consideration. They are basically text-centered notions, and the textuality of a text
rests on its communicative functions as well as its internal text structures.

**Intentionality** When providing some text, the sender expects it to fulfill some intentions.
These intentions or goals are often vital to the interpretation of the actual text. The
utterance "The door is open!" (from [38]) may very well urge the receiver to shut the
door, even though this is not explicitly stated. Generally speaking, it is not enough
to understand the wording of the text alone; one must also recognize the intentions
behind it. Likewise, a good text will help the receiver understanding it by revealing
the sender’s communicative intentions.

**Acceptability** For a text to be accepted by the receiver, it must have some relevance to
her. The text can for example address the receiver’s goals, which are not always
that apparent. A request for information will often give only a partial description
of the desired reply content, the other parts may be determined by the requester’s
background or the purpose of requesting that particular piece of information. Hence,
inferring the goals that motivate the question is important to get the text accepted by
the receiver.

Intentionality and acceptability ensures the purposeful nature of communication,
thereby allowing some tolerance towards disturbances of cohesion and coherence.

**Informativity** The content of the text must be adapted to receiver’s knowledge. On the
one hand, the receiver should not be given too much information that is already
known to her. In that case the text will be boring, and she might not pay attention
to it. In many text generation systems, thus, routines are employed that prevent the
system from repeating information or including information that is already well-
known to the user. On the other hand, understanding a text usually requires that the
receiver be familiar with at least some parts of it in advance. If all the content of
the text is completely new, the receiver’s processing effort may become overloaded and
she might not be able to grasp the whole content.

**Situationality** As noted by Hovy [114], a text should take into account the whole con-
versation setting in which it is used. Knowledge of what is really going on and
what type of discourse the text is used in, should affect both the generation and in-
terpretation of texts. Taking the interlocutor’s wider context into consideration, he
is in PAULINE able to vary the content and presentation of texts according to 23
predefined discourse features.

**Intertextuality** This standard concerns the factors that make the utilization of one text
dependent upon knowledge of one or more previously encountered texts. For ex-
ample, Moore [174] keeps a record of the previously generated explanation, so that
follow-up questions from the user may be interpreted in accordance to that expla-
nation. In that system the question refers directly to previously generated text, but
normally the connections are more vague and subtle. In general, the intertextuality of texts has been responsible for the development of text types and classes of texts with typical patterns of characteristics [37].

The standards above comply with the work of Grice [83]. From his main principle of cooperative behavior,

"make your conversational contribution as is required, at the state at which it occurs, by the accepted purpose or direction of the talk exchange in which you are engaged,"

he has worked out four maxims, each of them directing the sender’s behavior in communication. Successful communication requires that messages be constructed and interpreted in accordance to these maxims, and it is assumed that this is normally the case⁴. In that sense, Grice suggests that the maxims below are implicitly and possibly unconsciously present when humans communicate. The conversational maxims are:

<table>
<thead>
<tr>
<th>Maxim of quantity</th>
<th>&quot;Make your contribution as informative as (but not more informative than) is required!&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maxim of quality</td>
<td>&quot;Do not say what you believe to be false, or that for which you lack adequate evidence!&quot;</td>
</tr>
<tr>
<td>Maxim of relation</td>
<td>&quot;Be relevant!&quot;</td>
</tr>
<tr>
<td>Maxim of manner</td>
<td>&quot;Be perspicuous!&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;Avoid obscurity of expression!&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;Avoid ambiguity!&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;Be brief!&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;Be orderly!&quot;</td>
</tr>
</tbody>
</table>

As noted by Cawsey [37], the five first standards of de Beaugrande & Dressler correspond roughly to Grice’s conversational maxims.

Both de Beaugrande & Dressler and Grice have stated their criteria of textuality as a list of general requirements. More specialized versions of these requirements have been worked out for various types of computer applications [101, 188, 197, 213]. Additionally, Cawsey [38] also argues that texts should be interactive of nature, allowing the receiver to interrupt the sender and ask questions concerning parts of the receiver’s text.

The standards of textuality have proven useful when constructing text generation systems [53]. Although somewhat vague and abstract, they have defined a starting point from where to generate communicative texts. Still, criticism has been raised that concern the psychological reality of these lists. It is not clear, for instance, why the fulfillment of

⁴Actually, is is possible to deviate from these maxims in a very apparent manner. These deviations are called “floutings”, and are used to indicate ironies.
Grice's maxims ensures successful communication, or whether there might be additional or more fundamental maxims. Sperber & Wilson [231] has launched the principle of relevance that tries to explain the psychological matters involved in communication, and that subsumes the results from de Beaugrande & Dressler and Grice. According to their theory, the relevance theory, there is only one unifying principle governing human behavior in communication:

**Principle of relevance**

"Every act of ostensive communication communicates the presumption of its own optimal relevance."

A user will assume that the message communicated be worth the processing of it and that the stimuli used be the most relevant for the communication. The interpretation of the message will be guided by these two assumptions, but if it turns out that the assumptions must be false, the message is deemed unsuitable for the communication.

The relevance theory has lately been applied in discourse analysis [21], and many have pointed out the similarities with Grice's conversational maxims [145]. While the relevance theory is motivated from perceptive and cognitive processes of humans, the maxims seem to be an easily accessible list of the outcome of these processes. Although these two do not always coincide, in many cases the maxims have turned out to be a satisfactory and tractable simplification of Sperber & Wilson's relevance theory. And in educational texts, which is the main focus in this work, little is indeed lost confining ourselves to the guidelines of Grice and de Beaugrande & Dressler [37, 38, 145].

**Graphics and Formulas**

Textual communication, supplemented with illustrative pictures, has been subject to a growing interest in recent years [143, 206]. In explanatory texts pictures have primarily been used to illustrate or highlight parts of the given text [38, 163]. They are not used as an independent communication channel, but form a supportive role in the total communication process. This is exemplified in Figure 5.2, which explains how a light dependent resistor looks like. Correspondingly, when identifying the location of a town in Maybury's system [163], the explanation will typically involve three acts: (1) displaying the region in which the town is located, (2) highlighting the town, and (3) generating a text saying that the highlighted spot on the map is that particular town.

To produce the graphical objects supporting a text, two kinds of visual communicative acts are usually used: Visual display acts, like display-region and depict-image [163], are invoked to display some graphical object, whereas deictic acts, like highlight and indicate-direction [38, 163], point out parts of an already displayed object. Strictly speaking, though, it is not necessary to have a separate set of visual communicative acts. In COMET [166], for instance, an act like directing is used both for textual and graphical parts of the explanation, and two additional features are used to indicate whether this particular element is to be presented as text, graphics, or both.
This component here is a light dependent resistor.

![Diagram of a light dependent resistor](image)

Figure 5.2: Explaining how a light dependent resistor looks like (from [38]).

The dose after adjusting for increased serum calcium was set according to the following formula:

\[ D2 = D1 \times C \]

where:
- \( C = \text{the reduction constant for increased serum calcium (0.8)} \)
- \( D1 = \text{the dose before adjusting for increased serum calcium (0.25)} \)
- \( D2 = \text{the dose after adjusting for increased serum calcium (0.20)} \)

Table 5.1: Mathematical formulas may be used to explain events in XPLAIN [234].

Formal languages may also be useful in communication processes. In many natural language generation systems, they form a supportive role, illustrating or describing selected parts of the message to be conveyed. In the text shown in Table 5.1, for example, a mathematical formula is included to explain an event in the system’s reasoning. Other systems exploit formal languages in the same manner to improve their texts.

Although natural language is usually emphasized in generation systems, the usefulness of formal languages is tacitly accepted. This work will reflect this, and we will use the term "text" for both pure natural language texts and natural language texts including supplements in some formal language. In the latter case, though, we require that the formal language parts are textual and clearly subordinate to the natural language parts.
5.3 The Concept of Explaining

Messages, in general, are generated due to the sender’s intention to change the receiver’s state of knowledge. This intention may be sparked by observing the receiver’s behavior or by receiving some message from her, but still the message is primarily meant to fulfill the intentions of the sender.

Explanations, on the other hand, are intended to satisfy receiver’s needs as well as sender’s intention. They are provided in response to receiver’s lack of knowledge and can be requested by the receiver or just made available by some co-operative sender. Also, explanations are not restricted to purely textual messages. In ordinary conversations texts, non-verbal sounds, gestures and pictures may all be used to explain some phenomenon, although most explanations tend to include a textual kernel. According to [147],

"an explanation is a description that in practice enables one to understand a certain phenomenon."

We will here not try to define the notion of explanation in general, but will try to develop an understanding of the term by characterizing its typical structures, features and properties. In information systems engineering, explanations are generated by the computer to satisfy the needs of its user. The communicative goal of the computer is either provided by the user directly or inferred from her behavior. Multimedia explanations are used, but these are usually restricted to the combination of text and simple graphics. In the rest of this work, we will use the term explanation in the following sense:

An explanation in information systems engineering is a text or a combination of text and graphics that
- describes some phenomenon or aspects of that phenomenon,
- satisfies the textuality requirements,
- is generated by a computer using some well-defined source model,
- is made available to the user of the computer, and
- intends to satisfy the user’s expressed or inferred needs of knowledge.

An interesting discussion of explanations, their structures and characteristics, can be found in [211].

5.4 Types of Explanation

Explanations are today used in a variety of computer-based systems, like expert systems [1, 48, 177, 203, 234], expert critique systems [100], on-line help systems [47, 101, 130, 195,
5.4. Types of Explanation

196], tutorial systems [37, 168, 205], tools for information systems development [232, 233, 55, 68, 113, 124, 202], and on-line report systems [131]. In the following, we will briefly present a classification of the main types of explanation generated by these systems, using the notions of explanation domain and distinctive explanation features. A more elaborate discussion of that classification is found in [85]. Within specific fields, other classifications have been suggested (expert systems [1, 17, 40, 177, 203, 243], online help systems [101], information systems engineering tools [55], and tutorial systems [168]), but these are all subsumed by the one introduced in [85].

Explanation Domains

The explanation’s domain is a characterization of the content of the explanation. Basically, there are at least three major domains from which an explanation’s topic may be drawn: (1) user abilities, (2) system theory and (3) system behavior, which is comprised of both program-independent theories and underlying theories of programs. It can be argued, though, that we should include a forth one which may be called explanation. An explanation’s domain is explanation, if it explains parts of a previously generated explanation. Below, each of these domains are elaborated on and subcategorized into more specific subdomains.

1. **User abilities** This domain is most relevant to help systems, and involves explanations on what the user can do using the program. The explanations encompass goal-directed sequences of user actions as well as listings of possible single actions, and are divided into the subdomains enablement, functionality and orientation. We will not go into these explanations here, but the interested reader can find a good overview of user abilities explanations in [101].

2. **System theory** The explanation component’s system theory is the knowledge represented in the conceptual model (or rule base) and — if explicitly represented — the meta model. The questions associated with this domain are independent of the system state. They concern the general theory knowledge possessed by the explanation component, and their focus is on the theory itself rather than on its exploitation in any program. Hence, the explanations are related to neither user actions nor system actions, but are triggered from questions referring to elements or aspects of the conceptual model or rule base and possibly also the meta model.

   The explanations here include rationale explanations, element justifications, function descriptions, behavior descriptions, and structure descriptions [1, 101, 168, 188].

3. **System behavior** The domain is only relevant if the explanation component keeps a source model of some program. All the explanations related to the execution and purposeful use of the program are associated with this domain. The explanations try to make the system’s computations or ongoing activities more transparent to its users, helping them to understand the program and evaluate its behavior.
(a) **State clarification** Some explanations in this domain only describe the current system state, like answering questions like "What is the system status?" [101] or "What mode am I in?" [101]. Others justify the system's current goals or tasks (e.g. "Why is the valuation of soil quality necessary?" [17]), or explain how and why particular acts are being carried out (e.g. "Why are you using <method> to achieve <goal>?" [174, 188] or "Why are you doing <act>?" [174, 177])

(b) **Input justification** The explanation explains why a particular input is needed in the course of execution, by relating it to some ongoing acts or goals. It is triggered by questions like "Why is this question being asked?" [56, 188]

(c) **Result interpretation** In some cases it may be difficult to know exactly what the system has done. The results may be partially hidden to the user or they are hard to understand. The explanations here can either rephrase a complex result message or give information about results that are not so apparent to the user. An example question is "Has message 2 been deleted?" [101]

(d) **History** The questions here concern the system's execution or problem-solving behavior. They are related to the acts leading to the latest result, and are usually based on some execution trace. These results are not intermediate results to perform higher-level tasks or achieve higher-level goals, but form stable system states and final conclusions. Questions in this domain may be related to the steps in a computation or solutions process [40] (e.g. "Why (not) <conclusion>?" [174, 188] or "How did you achieve <goal>?" [174]), the way acts were carried out (e.g. "How did the system do <action>?" [188]), or the reasons for choosing specific acts, tasks, or goals (e.g. "Why did the system not investigate the impact on air?" [17]).

(e) **Hypothetical explanation** Using hypothetical questions, the user can explore what would have happened if some other user input were given, some other system acts were performed, or some other result should be produced. Example questions are "What if the impact on soil were valued as 'high'?" [17] and "What rules would apply if it could be shown that 'spark is not present' at the cylinders?" [203]

(f) **System abilities** System abilities explanations tell the user — on the basis of current system state — what it can do, how it would do it, and why it would do it in that way. Unlike user abilities and functionality questions, the focus here is on what the system itself can do rather than on what the user can do using the system functions. Questions referring to the abilities of the system can for example be "How does the system do <action>?" [188] and "How do you deduce the following diagnosis?" [1]

4. **Explanation** In this domain it is possible to ask questions related to the previously generated explanation. There may be parts of the explanation that the user did not understand or believe, or she just wants more information about the topic. Subdomains are elaboration and justification.

(a) **Elaboration** An elaboration question asks the system to elaborate on (part of) a previously given explanation. The explanation try to rephrase the original
text or include additional information about the topic. For example, "Huh?" [174] in EES is used to request an explanation elaboration.

(b) Justification In this case the user understood the explanation, but she is not convinced about the validity of it. She asks the system to justify those parts of the explanation that she did not believe in (e.g. "Why use 'next'?" [101]).

The hierarchical structure of explanation domains is shown in Figure 5.3.

In order to generate explanations from a specific domain, the source model must contain the necessary domain information. User abilities explanations may be available if the source model specifies how the system is supposed to be used (as is done, for example, in task analysis and user modeling [118, 220]). System behavior explanations require that the system model be executable and that an execution trace be recorded. Besides, if hypothetical explanations are to be generated, we need mechanisms for backtracking to the previous state after executing some parts of the model. System theory explanations are related to the actual elements of the source model, no matter their interpretation or use, and are supported in some way or another in all explanation generation systems. The structure and expressiveness of the source model, however, greatly affects what kinds of system theory explanations are producible. Whereas descriptions of function, behavior, and structure are found in systems based on traditional modeling languages like ER [55, 124, 168], Petri-net [131], and rule-based languages [39], rationale explanations are harder to relate to model elements. To support this description type, one might introduce design decisions into the source model in line with the works of [142, 194]. In some cases, explanations of previously generated explanations rest on interface routines being able to reason about

---

5Meaning that the user did not understand the explanation.
user's comments to the explanation she did not understand, but usually the follow-up question can be directly interpreted without reference to what has already been generated.

**Distinctive Features of Explanations**

Distinctive explanation features are used to distinguish explanations in the same domain and to describe properties that are shared across the domains. Each has a predefined set of values, and each describes the way a phenomenon is explained rather than the type of phenomenon explained [85].

Using these features, an explanation can be said to be

- **terminological** (describes generic constructs), **intensional** (describes domain properties), or **referential** (describes an execution of a model or an instance of a model element);
- **principled** (explains a topic by referring to some underlying generic principles), **theoretical** (no generic principles, no instances, and no execution trace elements are included in the explanation), or **illustrative** (explains a topic by including instances or executions related to it);
- **comparative** (topic is compared to some other phenomenon) or **centered** (topic is explained without references to comparable phenomena);
- **textual, graphical, or textual with graphics**; and
- **informative** (explanation just provides information about the topic) or **corrective** (explanation corrects a false statement, state, or opinion, like answering a why-not question in expert systems).

Consider the **how** question in traditional expert systems, which is a typical history question. If a system is to explain how a certain conclusion was found, it usually paraphrases the instantiations of the rules used in the deduction. Characterizing the explanation in terms of the features above, we can call it a **referential, principled, centered, textual, informative history explanation**.

### 5.5 Representing Explanations

Consider the process P3 **Process loan requests** in the conceptual model in Appendix A. A simple natural language text describing the function of the process might look like this:

"P3 is a process that processes loan requests. Dynamically, it can receive Requested loan, Customer data, and New customer data, and can generate Recommendation."
This text is comprised of two sentences, and each sentence includes a number of smaller units called clauses. A clause is a textual unit that contains at least one noun (phrase) and at least one agreeing verb (phrase) [59]. A sentence is required to contain at least one such clause. In the first sentence above, we have two clauses, "P3 is a process" and "P3 processes loan requests". Note that the constructions connecting the clauses, like the "that" in the first sentence, are not parts of any clause themselves.

A natural language text can at least be analyzed at four different linguistic levels [82, 114, 200]. At the morphological level, word inflection and word formation are the two central issues. The syntactic level describes how words and phrases are combined to form higher-level phrases and sentence parts. The semantic level deals with the meaning of natural language text, whereas the pragmatic level tries to relate the text to how it is used in a broader communicative setting.

At the sentence level, various types of formalism for describing and analyzing natural language exist (see [214, 252]). We will in this work use a grammar formalism called Functional Grammar, which represents syntactic, semantic, and pragmatic knowledge in the same framework. Using this formalism, we can represent "P3 is a process" as

\[
\text{DECL(PRES } e_1: (\text{process}_N (d[1] x_1: p3)_{\text{Subj Top} \text{Top}})).
\]

Process P3 has the semantic role zero (\(\emptyset\)), the syntactic role subject (Subj), and the pragmatic role topic (Top). In Appendix C, the main features of Functional Grammar are described and illustrated.

At the paragraph level, a text structure, also called a discourse structure, describes how text units are related to each other to achieve some higher-level goal. Essentially, these structures are text-oriented notions, in that the relationships between units are completely determined from the contents of the units alone [21]. A unit, then can be anything from clauses to paragraphs, and can be either complex or atomic. A complex unit is a structure of other units and are referred to as rhetorical relations or rhetorical predicates [168]. Atomic units, which are not decomposable, are usually independent clauses [161], but depending on the objective of the analysis larger units may be employed.
Figure 5.4 shows a text structure comprised of three atomic clause-size units and two complex ones.

From an explanation generation point of view, the research on text structures have given some important results:

- Every coherent text has some sort of structure. The structure ties the units of the text together, so that the text as a whole is perceived as a natural unit [38, 188].

- Although not clear in general, many well-structured discourses tend to be comprised of a strictly hierarchical structure [37]. Rhetorical relations at one level may then be used to realize some higher-level relation, as shown in Figure 5.4.

- The rhetorical relations used to structure a text in a given domain fall into a fixed set of relation types. The number of relation types and the nature of these types may differ from one domain to the other, but within a limited and well-structured domain it has still proved satisfactory to consider a closed set of relations [115].

- In many domains it is possible to characterize the texts in terms of a small number of discourse structures. For example, when analyzing descriptions of data base structures, McKeown [168] found only four types of discourse structures: identification structure, constituency structure, attributive structure, and contrastive structure.

In this presentation of rhetorical relations, we will focus on Rhetorical Structure Theory (RST) by Mann & Thompson [160, 161], in which a number of about 30 relations have been found. Other lists of relations have been used by Grime, Hobbs, Williams and McKeown [55, 168], but they will not be treated here. Their relations differ only slightly from the ones in RST, but their theories have not received the same attention as RST in text generation (see for example [52]).

Rhetorical Structure Theory stems from intensive studies of naturally occurring patterns of text. The theory is both descriptive and constructive of nature, showing promising results in text analysis as well as in text generation. It is based on the assumption that texts be hierarchically structured, and that most coherent pieces of text have a central part and zero or more subsidiary parts linked to the central one by relations.

In RST, a text structure is analyzed as a tree of instantiated schemas. A schema indicates how a particular text unit is decomposed into one or more other units, and is defined in terms of a relation. In such a schema, there is a nucleus, which is the central part of the relation, and a number of satellites that are linked to the nucleus by relations. In most cases there is only one satellite, but as we will see soon there might be several satellites related to one single nucleus. In Figure 5.5 it is shown how these schemas are depicted. We will use the notation in Figure 5.5(b).

A schema is associated with the definition of a relation. These definitions determine when the corresponding schema may be applied and consists of four fields: (1) constraints on the nucleus, (2) constraints on the satellite, (3) constraints on the combination of nucleus and satellite, and (4) the effect.
Consider the instance of the *evidence* relation in Figure 5.4. Informally, the definition of the *evidence* relation may be stated as follows:

1. *Constraints on the nucleus (the claim):* The reader possibly does not already believe the claim.
2. *Constraints on the satellite (the evidence):* The reader either already believes the satellite or will find it credible.
3. *Constraints on the combination of nucleus and satellite:* As the reader understands the evidence, her belief in the claim will increase.
4. *The effect:* The reader will believe the claim.

In order to use the relation to describe the text in Figure 5.4, then, it is assumed that the reader do not fully accept that the car is French. Moreover, she will believe it to be a Renault when told, which will increase her convictions that the car really is French.

An important aspect of RST is its attempt to list all the relation types that might occur in ordinary coherent texts. The number of relations and their corresponding definitions are still subject to changes, but it seems clear that at least the following relations will be included in the theory:

- antithesis
- background
- circumstance
- comparison
- concession
- condition
- contrast
- contribution
- disjunction
- elaboration
- enablement
- evaluation
- evidence
- interpretation
- justify
- means
- motivation
- non-volitional result
- otherwise
- restatement
- sequence
- solutionhood
- summary
- volitional cause
- volitional result
- non-volitional cause
Figure 5.6: RST structure of the functional description of P3.

Usually, a text cannot be fully analyzed using one single schema application. There will be a series of applications, in which the units forming one schema instance themselves are decomposed into a tree of other schema instances. Taking the explanation of process P3 as an example, we get an RST diagram describing its structure as shown in Figure 5.6.

A distinguished feature of RST is the asymmetric nature of relations. When using nuclei and satellites it is assumed that one part of the relation is more prominent than others, and that this part is most essential to the meaning of the text. Without the nucleus of the evidence relation, for example, the satellite is non-sequitur. Also, the satellite part of a relation instance may be more easily replaced than the nucleus part [174]. In addition to this, it is assumed that the prominent part is predictable from the relation itself, so no references to the contents of the parts or the relation’s context may be necessary.

A problem with this asymmetry is the treatment of paratactic constructions. These constructions are complex and involve the coordination of simpler units linked by some junctional devices (e.g. *and, or and but*). It is not always possible to discriminate between parts of paratactic constructions, deeming one part more prominent than the others. The solution chosen in RST, then, is to encode relations involving paratactic constructions as *multi-nuclear* relations, letting all their parts be marked as nuclei. In the work of Mann & Thompson, three such multi-nuclear relations are identified: *sequence, contrast, and list*. Both Scott & Souza and Rösner & Stede have found it useful to include a forth one, *alternative*, which is used for disjunctive constructions.

In text generation, RST relations have mostly been used to encode strategies for forming coherent pieces of text. We will come back to that exploitation elsewhere, but one should also note that attempts have been made to use the theory as part of a general knowledge representation language [161]. In the work of Harrius [100], the domain knowledge of an expert critiquing system AREST is represented as conceptual graphs [230] extended with RST relations.

Among the problematic issues in RST are the following:

- So far, it has been impossible to agree on the ontology of rhetorical relations. When RST is used on a particular domain, only a subset of the relations are needed and these are often specializations of the original RST relations (e.g. [168, 189, 205]).
5.5. Representing Explanations

The minimal unit in RST analyses is usually considered to be a clause. However, as the multilingual project TECHDOC [205] shows, a clause structure in one language can be realized as for example a prepositional phrase in another, and in these cases there is no clear criterion for separating out minimal units of the text.

RST structures seem to be most suitable for microstructures of text. When larger structures are to be constructed (macrostructures), the relationships between them appear to be more governed by schematic standards or communicative goals and effects [115, 205].

An RST structure may fail to capture more complex dependencies among text segments. In Figure 5.7, for example, which is the RST structure for

"[The spark plugs must be securely tightened]_A, but [not over-tightened]_B. [A plug that's too loose]_C [can get very hot]_D and [possibly damage the engine]_E; [one that's too tight]_F [could damage the threads in the cylinder head]_G."

The two contrast relations denote the same phenomenon, but this is impossible to represent in the structure.

The strictly hierarchical structure of RST structures hide the fact that some text segments may be more prominent to a relation than others. In Figure 5.6, for example, the nucleus of interpretation is definitely more central to the dynamics relation than the satellite.

Today, RST is a well established theory for text generation systems [52, 115]. Depending on the scope or the domain of these systems, however, the theory is often domain-tailored and accompanied by other theories for the construction of larger structures of text (e.g. [110, 146, 205]).
Chapter 5. Explanations in Communication

5.6 Two Levels of Explanation

For this work we have found it useful to introduce an intermediate level of explanation, the deep explanation. Although sometimes not stated explicitly, most explanation generation systems have an intermediate representation of explanations that corresponds to our notion of deep explanations. In [166], for example, the term logical form is used for a representation comparable to deep explanations. Other terms in line with deep explanations are ordered message [167], internal and external structure of text [221], text plan [174], text structure [111], paragraph structure (tree) [115, 116], realization specification [131], and tactic structure [29]. Besides, process trace [189], RST structure [205], task hierarchy [101] and augmented conceptual network [46] have been used on a more restricted class of deep explanations.

- The deep explanation is an intermediate representation of an explanation to be presented to some user. The deep explanation includes, on a conceptual basis, the content and structure of the final surface explanation. Each element of the deep explanation is associated with either a graphical realization or a textual realization. Also, the element's function in the communication process may be — either explicitly or implicitly — represented at this level. The deep explanation may represent information that relates it to the standards of textuality, but is otherwise independent of any linguistic considerations. Syntactic and morphological realizations are not represented at this level of explanation.

- The surface explanation is the actual explanation shown at the display and consists of natural language sentences and possibly some pictorial information. Unless the context indicates differently, the term explanation will be used synonymously with surface explanation. Surface explanations are based on some underlying deep explanation and is realized using linguistic and graphical knowledge on the content of these deep explanations.

A deep explanation is a linear or hierarchical structure of deep explanation elements. If the structure is hierarchical, the linear ordering of the explanation content elements is often given implicitly as a depth-first traversal from left to right (but not if it is an RST structure). Alternatively, the ordering is explicitly specified for each non-leaf node, directing in which order its immediate descendants are to be traversed [116, 205].

The non-leaf nodes of the hierarchical structure are called non-terminals. All other nodes of deep explanations are referred to as terminals.

All terminals contain a content element, i.e. a content reference or a content clause. A content reference refers to some clause that is to be included in the surface explanation or to some part of the source model. The content clause, which is often referred to as content proposition [167], is an underlying representation of the NL clause to be included in the surface explanation. Additionally, the terminals may include communicative acts telling what to do with these content elements. The acts are usually represented as predicates, like
5.6. Two Levels of Explanation

Assert(SYSTEM, USER, Location(# <Karl-Marx-Stadt>)) [163] and pointat(device) [38], but features describing communicative acts can also be used [166].

Non-terminals describe how lower-level elements fit together to fulfill some higher-level goal, and they fall into two main categories:

Text-structuring elements These are text-oriented elements specifying how two blocks of texts are related to each other textually, independently of the conversation in which they are used. They help ensure the coherence of the text, and are typically implemented as rhetorical relations (see [52]).

Discourse-structuring elements These are related to communicative acts, goals and effects, and show how the text is used in a wider conversation setting. There are systems that make use of only one type of discourse-structuring elements (e.g. [37, 163]), though many combine acts and goals (e.g. [101]), or acts and effects (e.g. [174]).

In some systems only text structuring elements or only discourse-structuring elements are used, but often there is a combination of text structuring elements and discourse-structuring elements present in deep explanations (for example in [174]). Also, a specific non-terminal may be associated with both a text-structuring part and a discourse-structuring part, as is the case in [116].

The representations of deep explanations can be shown using examples from existing generation systems. In Figure 5.8, there is a deep explanation generated in an expert system framework called EES (Explainable Expert System) [174]. The deep explanation corresponds to the following surface explanation:
"I'm trying to enhance the readability of the program by applying transformations that enhance readability. CAR-to-FIRST is a transformation that enhances readability."

The terminals of the deep explanation here are comprised of primitive linguistic acts and content references. In the terminal "(INFORM SYSTEM USER APPLY-1)", for example, "INFORM" is a non-decomposable communicative act and "APPLY-1" is a content reference. Non-terminals are either text-structuring (like "MOTIVATION") or discourse-structuring (like "PERSUADED") elements.

Another example, illustrated by the deep explanation in Figure 5.9, shows how pictures may be included in deep explanations. The resulting surface explanation is a map displaying the region around Karl-Marx-Stadt, in which the town is highlighted, and the map is accompanied by the following text:

"Karl-Marx-Stadt is a town located at 50.82° latitude 12.88° longitude."

The terminals of the deep explanation are primitive communicative acts and content references. "Display-region" is a visual display act, "highlight" a visual deictic act and "assert" a primitive linguistic act. The non-terminals are discourse-structuring elements related to abstract communicative acts.
Chapter 6

Deep Explanation Generation

*Natural language generation* denotes the computer's production of natural language utterances — comprising both single-sentence generation and the generation of entire discourses [53]. The input to the generation process is usually some structure indicating the desired utterance content, whereas the output is a text or a combination of text and pictures. *Explanation generation* is the production of natural language in an interactive communication process and has recently received a lot of attention in computer applications [167, 190].

In this chapter we examine the techniques used in explanation generation. Our focus is on the deep generator, but also the main tasks involved in surface generation are mentioned. We outline the process of generating explanations and see how and when different types of decisions are involved in the process. Discussing generation on a purely conceptual basis, we will free ourselves from any particular domain in which generation systems have been used. Techniques exploited in one of these domains are included in so far they have a potential in a domain-independent explanation generator. The description below, thus, is not the description of any real system, but rather of a merge of the most well-known generation systems. In fact, no system today includes all the techniques below, but most of them share the same general approach to the generation process.

6.1 The Main Principles of Explanation Generation

The successful generation of explanations rests on a wide range of knowledge sources and considerations. Based on these sources and a request from some user or some other part of the computer system, an explanation is generated and made available to the user.
Essentially, this generation process may be described as a series of choices\(^1\) [167]. The available knowledge pertaining to the phenomenon to be explained is usually quite extensive, and the generator has to decide what information to include and what to discard. After this is done, choices have to be made concerning the arrangement of this information in the explanation. Generally speaking, there are several feasible ways of structuring the information, and in deciding on one of them the generator will have to take into consideration both the intended user and the context in which the explanation is requested. Next, sentence structures or graphical representations must be chosen that realize the various pieces of information to include. Constructing a sentence, for example, one must determine which words to use, how to arrange the words into sentences, and how to use sentence structures and cohesive devices to coordinate a sequence of sentences. What graphical representations are concerned, the choices include size and appearance of pictures, and devices for emphasizing, suppressing or pointing out parts of these pictures.

At all these decision points, the generator needs principled strategies for making a choice, and the assembly of these strategies forms the core of the generator.

**Modularization of the Generation Process**

Looking at the decisions involved in explanation generation, one is tempted to group them into two consecutive stages. In the first stage, all linguistic aspects are ignored and the focus is on the conceptual structure and content. In the second stage, linguistic and graphical considerations are introduced to translate the conceptually oriented representation into the final surface explanation. These two stages of the generation process are called *deep generation* and *surface generation*, respectively [167].

**Deep generation** At this stage the explanation's content and structure are determined. If several media or languages are available, the elements of the deep explanation are also associated with a particular medium and a particular language. On the basis of an explanation request, thus, the deep generator plans a deep explanation that ideally meet all the seven standards of textuality. The process is often referred to as *text planning*, and the corresponding system component as the *conceptual* [221] or *strategic* [131, 168, 181] component.

**Surface generation** Accepting a deep explanation as input, the surface generator's responsibility is to produce a surface explanation consistent with the deep one. In this component, called *linguistic* [221] or *tactical* [131, 168, 181] component, decisions concerning lexical items and sentence organization as well as graphical realizations are made. The complexities of the component may vary, ranging from full natural language production to simply mapping of deep explanation structures onto canned sentences or templates. Another common term for surface generation is *text realization* [114].

---

\(^1\)The generation process is sometimes referred to as "decision-making under multiple constraints" [190].
6.1. The Main Principles of Explanation Generation

In the deep generation component, we determine "what-to-say", in the surface generation component the "how-to-say-it" part is added [198].

Ideally, the whole generation process could be divided into these two consecutive and independent subprocesses, using deep explanations as the only interface between the two. Deep generation could then take place in its entirety before surface generation starts, reducing the complexities of the decisions involved in each subprocess considerably. There are tasks, though, that cannot be isolated to either of these subprocesses, but that rather spread themselves out all over the generation process. One of them is user-tailoring, the task of tailoring the explanation to a particular user or class of users, which affects practically all tasks within both deep generation and surface generation. It may influence what to include in the explanation, how to organize the content, and what words and sentence structures to use.

On this basis, it has been argued that the division of the process into deep generation and surface generation is unnatural, and that the process should be treated as an integrated whole [7, 111, 114]. Viewing the generation task as a continuous process, all kinds of constraints and decisions could be treated in the same way. In KAMP [6], this is realized using a hierarchical expansion planner. There are doubts, however, whether it is feasible or at least efficient to use this integrated approach (see [114]).

Others have accepted the use of a strategic and tactical component, but have insisted on a more complex interface between the two components. Hovy [116], for example, describes a method called limited-commitment planning, in which deep generation and surface generation are interleaved. In his system, called PAULINE, deep generation decisions are deferred until necessitated by the surface generator, and there is a two-way communication between the two components. The bidirectional interaction between the components is also found in a natural language generation system called POPEL [198].

In this work, we adopt the notions of deep generation and surface generation. We do not require that the deep generation process and the surface generation process are two strictly consecutive stages of the explanation generation process. To find a suitable structure for the presentation, though, we assume that most conceptually motivated generation decisions can be explained separately from the linguistically ones. We also define a deep generator core that is restricted to those conceptually oriented generation decisions that are not affected by any linguistic considerations or that can be used without conferring with linguistic knowledge. Deep generation decisions outside the core interacts with surface generation decisions, but are still related to the conceptual content of explanations.

In the presentation of the deep generator below, most of the discussion will concern its core. Those deep generation decisions that may be affected by surface generation decisions will be mentioned, but not treated in detail. This is done, since the focus of this work is on the deep explanation core rather than the total explanation generation process.
Knowledge Needed to Generate Explanations

Explaining properties of some phenomenon, the generator needs knowledge of the following categories:

- The **request** identifies the user’s need for an explanation. It is usually assumed that the request itself determines what to explain; that is, it is not necessary to consult any user model to interpret the explanation request.

- A **source model** is a combination of models and is the generator’s main source of information about the phenomenon. It may be any combination of the models below:
  
  - A **meta model** defines those aspects of the constructs used in the tool’s modeling language pertaining to the generation of the desired explanations.
  
  - A **conceptual model** can be a **domain model**, a **computer system model**, or a **database model**. In information systems engineering, a **domain model** is used to develop a mutual understanding between developers and domain experts about the properties of some domain. A **computer system model** is typically used in help systems. It keeps information about how to use a computer program, and this information is used to produce guiding instructions to the user. The model could also be an executable information system model, in which case the explanations are based on the execution unit itself. A **database model** is used to explain the structures of a database. The model is static, and the explanations generated are supposed to assist the users when performing operations on the database.
  
  - **Instances** and **traces** may be available if the conceptual model include a database model and/or is executable. **Database records** are instantiations of database structures. An **execution trace**, resulting from the execution of a program, is a record of the behavior of the program.

Domain models, computer system models and database models are in this work referred to as just **conceptual models**.

- A **user model** describes properties of the user and may contain information about user’s type, preferences, goals, or knowledge. All these four user aspects may be implemented as views of the source model. The user knowledge view marks all the elements that are known to the user, the user preference view all elements that the user would want the system to use in its explanations, the user goals view all elements that are relevant to her goals, and the user type view all elements that are associated with that particular class of users.

- **Explanation context** characterizes the situation in which the explanation is requested, and may influence the content of the explanation. Exercising an executable model, for example, one may be interested in shorter explanations than the ones requested when documenting that particular model. The dialogue history between users and system is part of the explanation context.
6.2. Deep Generation

- The already planned parts of the explanation may be needed to avoid repeating information in the explanation.

- Discourse strategies are primarily used to structure the explanation, but are also useful in selecting the appropriate explanation content.

- Linguistic knowledge is used by the surface generator and includes a lexicon indicating how concepts in the deep explanation can be mapped onto lexical items in the surface explanation, and a grammar guiding the organization of sentence constituents into sentences. There may also be additional knowledge linking pragmatic and syntactic information to syntactic realization.

- Model presentation knowledge includes routines for displaying views of the system model. Since many modeling languages are graphical, the routines will typically involve graphical knowledge as well as knowledge about how views of system models are described.

In the following we will go more into the details of some of these sources and see how they are used to generate a complete surface explanation.

6.2 Deep Generation

Deep generation is naturally divided into at least four subtasks, each of them vital to ensure a satisfactory deep explanation result. A number of techniques are available in the strategic
component, and many of them are relevant to several subtasks. User-tailoring, for instance, penetrates most subtasks both in deep generation and surface generation. Also, in many systems, several of these subtasks are integrated to produce a more effective generator. Still, we are here going to present the subtasks one by one, giving the following list in deep generation:

1. handling explanation request,
2. selecting explanation content,
3. structuring deep explanation, and
4. optimizing deep explanation.

Each of these subtasks involves the use of certain techniques and representations, and are explained below.

After the presentation of subtasks, we will briefly compare the strategies and techniques used in today's systems. The explanation strategies will be discussed with respect to efficiency, flexibility, internal structures and assumptions about their environments, whereas the techniques will be related to the seven standards of textuality.

**Handling Explanation Request**

Some very simple explanation generation systems do not need any request at all. Both the content and structure of their explanations are fixed, and the user's contribution is only to initiate the generation (e.g. the Gist paraphraser [232, 233] and Kalita's execution report system [131]).

In most systems, though, the explanation component is requested to explain a specific phenomenon. The communication between user and system has been depending on one of the following four methods [101]:

- She may formulate the request using *natural language* or a *subset of natural language* (see for example [18, 46, 91]). Although this is a very flexible way of communicating with the computer, few explanation generation systems have employed a natural language interface yet. The computational complexities of interpreting natural language questions may often get unreasonably high [112, 207, 243], and many users tend to prefer more compact request strategies than full natural language sentences [101].

- In some systems, like TEXT [168] and EES [174], questions are posted using stylized *command languages*. 
6.2. Deep Generation

- **Context-sensitive menus** are often used to ask questions to the computer (e.g. [1]). The problem of using this method is to organize the menus according to user's context-depending intentions.

- **Hierarchies of menus** are easier to implement than context-sensitive menus, but are usually not that effective to use. If the system is simple, however, perhaps only one menu is needed (e.g. [166]). Of course, it is also possible to combine context-sensitive menus with access to a whole menu hierarchy [101] or even to combine natural language with the use of menus [243].

Most explanation generation systems interpret the requests independently of user properties and explanation context. It is assumed that the user ask the "correct" question, or that a more sophisticated interface be added at a later stage. There are systems, though, that let the context or user model decide how to understand a question from the user. In EES [174], for example, three heuristics are used to interpret a why question. Basically, these heuristics say that the question should be interpreted in such a way that the conversation topic is maintained, and the corresponding explanation does no contain things that the user knows or that have already been said. In UC, the inferred user goals determine how to interpret an explanation request. Using the technique of goal inference, the utterance "Do you know how to delete a file?" is usually interpreted as the request "How do I delete a file?" [130].

Most requests are initiated by the user and concern the behavior of a program or the properties of a domain or a system. But there are also cases in which the request is coming from the system itself. In active help systems like ROMPER [165], an explanation is generated when the system thinks the user possesses some misconception about the domain. And there are also a few systems where the user is allowed to ask about unclear arts of already given explanations [37, 101, 174].

**Selecting Explanation Content**

After receiving the explanation request, most systems start planning the discourse structure right away. There is no separate content selection phase, and content elements are included according to how they fit into the planned explanation structure.

Still, it is possible to simplify discourse planning by constraining the relevant parts of the source model. On the basis of user's request and inferred user goal, McKeown et al. [170] partition off a subset of the knowledge base determined to be relevant to the request, and call this subset the **relevant knowledge pool**. When describing an object, the system generates goal-related texts by including only properties from this knowledge pool [168]. The technique is illustrated in Figure 6.2, in which a simple entity relationship model with generalization is used to represent properties of university courses. If user's goal is to relate the courses **discrete mathematics** and **data structures** to when the courses are usually taken in the process, only the **state model hierarchy** is made visible to the generation process. If, on the other hand, the user wants to know how courses tie in with requirement sequencing, only the **requirements hierarchy** is used to generate explanations.
A similar approach is taken in ROMPER [165], in which a domain view determines what objects and attributes may be included in the explanation. Every object and attribute in ROMPER's data model is given a salience value between 0 and 1. These values indicate the relative importance of the model elements, and when generating the explanations elements with high values are preferred to elements with low values.

In UC [46] explanations are generated to help the user use UNIX commands. To make the explanations as informative as possible, those parts of the source model that are known to the user are deleted before the explanations are generated.

**Structuring Text**

Generating explanation structures, various types of discourse strategies may be used. Basically, these strategies fall into two main categories:

- *discourse schemas* and
- *plan operators*.

There is also a third category that involves complicated reasoning about user's belief and knowledge. The systems of this category, though, have so far only been used for single-sentence planning, and it is not sure whether or not the approach is feasible for generating paragraph-length texts. Well-known examples of systems using this highly formal approach are KAMP, BERTRAND, and SPIRIT (see discussion in [167]). Concentrating on the generation of paragraph-length explanations, we will not be dealing with the techniques employed in these systems.
((deny (possess OBJECT MIS-ATTRIBUTE))
(state (possess OBJECT REAL-ATTRIBUTE))
(offer (confused-object/bad-analogy SIM-OBJECT))
(offer (similarity OBJECT
  SIM-OBJECT
  (share-attributes OBJECT
    SIM-OBJECT
    ATTRIBUTES))))

Table 6.1: Discourse schema for misattribution in ROMPER.

Most systems of the two categories above do not assume an initial content selection phase. The content is selected according to how it fits into the structures produced by the generator. The nature of the strategies, however, do not rule the content selection phase out, and it would often be quite easy to supplement the strategies with a content-constraining initial phase.

Below, it is shown how discourse schemas and plan operators are used to construct deep explanations.

**Discourse Schemas** A discourse schema encodes complete patterns of discourse structure. It may be a linear generic deep explanation structure or an algorithm for traversing the relevant parts of the source model. In both cases, instantiated schemas are complete deep explanations. Only one schema is used to generate a single multi-sentenced explanation, and the generation system will usually have a set of predefined schemas to be able to generate various types of explanations. All the deep explanations generated are linearly structured.

Discourse schemas may be implicitly [232, 233, 234] or explicitly represented in the system. If explicitly represented, they may either form a separate discourse strategy set or be equivalent to parts of the source model. An example of an explicitly, separately represented discourse strategy is shown in Table 6.1. The strategy is used in ROMPER when the user has given an object an incorrect attribute. The explanation generated denies that the object has this attribute, reveals the real attributes of the object, and tries to find another object that the user might have been thinking of. Discourse strategies that are equivalent to parts of the source model are usually found in expert systems. The rules, and even the traces of instantiated rules, may be used directly as discourse schemas to explain the behavior of the system.

A discourse schema may be either *declarative* or *procedural*. When a declarative strategy is employed, the structure of the deep explanation mirrors the pattern represented in the schema. Instantiating the schema, the deep generator picks pieces of information from the source model and puts them into the schema structure. A prominent example of this is
given in McKeown's TEXT system [168]. TEXT is used to describe and compare entities in a database. It has a number of augmented transition networks (ATNs) specifying possible explanation structures, and one of them is rendered in Figure 6.3. Each arc of the network (except the JUMP and POP arcs) is labeled by a rhetorical predicate that specifies a piece of information to be included in the explanation. To construct the deep explanation, the network is traversed from the CONSTIT node to the CONSTIT/End node. When the CONSTIT/Intro node is active, the system has two possible ways of continuing the traversal. To decide what arc to follow, the system uses a set of topic rules to choose the path that maximizes the coherence of the explanation. For each predicate-labeled arc that is passed, the corresponding predicate is instantiated and added to the deep explanation. Instantiating the constituency predicate, for instance, we may get something like

\[(\text{constituency SKIP (AIRCRAFT-CARRIER FRIGATE OCEAN-ESCORT, CRUISER, DESTROYER)})\]

Similar declarative schemas are found in COMET [166], MPA [40], ROMPER [165], TAILOR [189], and UC [46].

A procedural schema produces a deep explanation by traversing the source model. The schema dictates how to traverse the model, and the resulting deep explanation reflects the structures of the source model rather than the structures of the discourse schema. A schema of this type is implemented in TAILOR to generate process descriptions of physical devices. The schema, which is called a process trace, is an ATN and is depicted in Figure 6.4. Unlike the labels of the arcs of TEXT's ATNs, most labels here dictate how to chain through cause-effect links in the source model (NEXT-MAIN-LINK, SUBSTEPS, SIDE-LINK). On its way through the source model, the system may include attributive information about the device (ATTRIBUTIVE), or information about the substeps of a cause-
effect link (SUBSTEPS). Side links (SIDE-LINK) are included to get the preconditions for the events on the cause-effect links (NEXT-MAIN-LINK).

In the work of Kalita [131], an execution trace is traversed in very much the same way as in TAILOR, although the system does not have an explicit representation of the strategy. COMET uses the process trace schema from TAILOR to generate action plans.

**Plan Operators**  A plan operator associates a generic structuring element with one or more other generic structuring elements or (generic) content elements. The element is said to be realized by or decomposed into these other elements, and the operator is combined with other lower-level operators to plan a hierarchical deep explanation. Most plan operators have at least the following components:

- A **generic structuring element** that is to be realized instantiating the operator. The element is called the operator head or the operator’s discourse goal.

- A set of **generic structuring elements** or **content elements** that are used to realize the operator head. These elements are often called subgoals, and fulfilling the subgoals means that the operator’s goal is achieved.

- **Preconditions** governing the use of the operator. These preconditions may refer to properties of the source model, but may also refer to the user model, the request structure, the explanation context, or already planned parts of the explanation.

- An **effect** describing the communicative effects of using the operator. If the head is an effect-oriented structuring element, the operator’s goal and effect are the same.

Examples of plan operators in which the components are explicitly marked are given in Figure 6.3, 6.4 and 6.5. In Table 6.2 the plan operator is simplified and implemented as a rewrite rule in a Definite Clause Grammar [191].

An operator-based explanation generation system must possess a vast set of plan operators. Some of these operators are quite abstract and are associated with various types of
exemplification(\ldots(\text{elaboration}(E) \& \text{parallel}(P))) \Rightarrow \text{elaboration}(e,E), \text{parallel}(i,P). \{\text{not ocur}(E,P),!\} 

Table 6.2: Plan operator in Dalianis's system [55]. An entity is explained by giving an instance as an example.

explanation request. Others are more concrete and refer directly to elements of the source model. In order to plan an explanation, the system picks an abstract plan operator whose discourse goal can satisfy that particular type of request. The operator's components are instantiated, and if the preconditions are evaluated to true, the planning process proceeds by finding other plan operators matching the operator's subgoals. By combining plan operators in this way, the planner constructs the hierarchical structure of a deep explanation. The planning terminates when there are no more structuring elements to realize or decompose. If the preconditions are false or there are no plan operators matching the subgoals, the system must start backtracking by trying other instantiations or abandoning already included operators.

The planning process is perhaps best understood using an example. In the EES framework, an expert system called Program Enhancement Advisor (PEA) has been implemented. Given that the system has just recommended a replacement act to enhance the readability of the user's program, the system can construct an explanation that persuades her to do the act. The operators used are given in Table 6.3. Instantiating the PERSUADED operator, we get the following initial discourse goal:

\[(\text{PERSUADED} \ \text{USER} \ (\text{GOAL USER} \ (\text{DO USER REPLACE-1}))))\]

The preconditions of the operator are given in the constraints field. The system tries to find bindings in the knowledge base which satisfy all of the predicates in the constraint list. If there are constraints referring to the user (like (GOAL USER ENHANCE-READABILITY)), and there is no information about this in the user model, the system may assume the constraints to be satisfied. In our example it is possible to find only one binding that satisfies the constraints, and the nucleus subgoal

\[(\text{MOTIVATION REPLACE-1 ENHANCE-READABILITY})\]

is posted. Using the MOTIVATION predicate, this goal is decomposed into the subgoals

\[(\text{INFORM SYSTEM USER ENHANCE-READABILITY}) \ \text{and} \ \ (\text{MEANS REPLACE-1 ENHANCE-READABILITY})\].

The INFORM goal is a primitive speech act and a content element. Using the MEANS predicate to expand MOTIVATION's satellite, we get another content element and a new discourse subgoal:
### 6.2. Deep Generation

<table>
<thead>
<tr>
<th>Effect</th>
<th>Constraints</th>
<th>Nucleus</th>
<th>Satellites</th>
</tr>
</thead>
</table>

Table 6.3: Plan operators in EES. Effect is the operator’s discourse goal, constraints its preconditions and nucleus and satellites its subgoals.
(INFORM SYSTEM USER APPLY-1) and
(BEL USER (STEP REPLACE-1 APPLY-1)).

The planning proceeds until all leaves of the explanation’s goal structure are content elements. The deep explanation in Figure 5.8 shows the result of the process above.

Although the general principles of operator-based deep generators are the same, there are some important differences. Among these are the choice of structuring elements as discourse goals and the expressive power of the system’s plan operators.

**Structuring Elements** The types of structuring elements used as operator heads vary from one system to the other. In EES, a combination of effect-oriented elements (like PERSUADED) and text-structuring elements (like MOTIVATION) are used. The text-structuring plan operators are formalizations of Mann’s RST-based rhetorical relations, which explains the use of nucleus and satellite to specify subgoals. RST-based plan operators are also found in Hovy’s system, and in both systems the relations are formalized in terms of their effects. In some other systems only the text-structural properties of RST-relations are formalized [100, 221]. Characteristic to RST-based operators is that the structural relation between the subgoals of an operator is explicitly represented in the operator. Text-structuring elements of a different kind are used in Cawsey’s EDGE system. Except for the precondition field\(^2\), these text-structuring elements do not reveal the subgoals’ interrelationships as clearly as RST-based elements do.

The effect-oriented elements of EES specify the intentional effects of using these operators to build the explanation. They can later be used to replan parts of the explanation if the expected effects were not achieved. In Maybury’s formalization of plan operators, the structuring elements are communicative acts (see Table 6.5).

**Operator Flexibility** Often, there are several ways of attaining the discourse goal of an operator, and each way yields a slightly different explanation. Often, these variations are subsets or supersets of each other, and it can be an advantage — both conceptually and implementationally — to cluster them into one single operator. In many of these cases, it is a question of including something extra in the explanation — or removing some small parts of it — due to user’s previous knowledge of the topic. In other cases, the quality of an already acceptable explanation may be improved by adding some small pieces of extra information. The goal of the operator is the same, but the decompositions are slightly different from one explanation to the other.

So far, there have been two approaches for increasing the expressive power of plan operators. In EES a satellite may be marked as optional. This means that the user model and the explanation context\(^3\) are used to determine whether this subgoal should be included in the explanation or not. Basically, an optional satellite

---

\(^2\)The **precondition** field corresponds to the RST **background** relation saying that the information in **precondition** is the background for the information specified in **subgoals**.

\(^3\)To be precise, only a small part of the explanation context, the **dialogue history**, is used in this process.
is expanded and included if the user is a novice or the dialogue history shows that the information provided by the satellite is usually requested afterwards anyway. Otherwise, the satellite is left out to make the explanation more compact.

Name: SEQUENCE
Results:

((BMB SPEAKER HEARER (POSITION-OF ?PART ?NEXT)))

Nucleus+Satellite requirements/subgoals:

((BMB SPEAKER HEARER (NEXT-ACTION.R ?PART ?NEXT)))

Nucleus requirements/subgoals:

((BMB SPEAKER HEARER (TOPIC ?PART)))

Nucleus growth points:

((BMB SPEAKER HEARER (CIRCUMSTANCE-OF ?PART ?CIR))
(BMB SPEAKER HEARER (ATTRIBUTE-OF ?PART ?VAL))
(BMB SPEAKER HEARER (PURPOSE-OF ?PART ?PURP)))

Satellite requirements/subgoals:

((BMB SPEAKER HEARER (TOPIC ?NEXT)))

Satellite growth points:

((BMB SPEAKER HEARER (ATTRIBUTE-OF ?NEXT ?VAL))
(BMB SPEAKER HEARER (DETAILS-OF ?NEXT ?DETS))
(BMB SPEAKER HEARER (POSITION-OF ?NEXT ?FOLL)))

Order: (NUCLEUS SATELLITE)
Relation-phrases: ("" then" "next")

Activation-question:

"Could "A be presented as start-point, mid-point, or end-point of some succession of items along some dimension?"

that is, should the hearer know that "A is part of a sequence?"

Table 6.4: The RST-relation sequence formalized as a plan operator in [116].

Hovy has introduced two sets of nucleus growth points and satellite growth points in his RST-based plan operators. The purpose of these points is to add information to the explanation if this additional information happens to be available in the source model. Consider the plan operator in Table 6.4, which is a formalization of the RST relation sequence. Achieving the operator’s goal,

((BMB SPEAKER HEARER (POSITION-OF ?PART ?NEXT))),

the system expands the nucleus requirements/subgoals and the satellite requirements/subgoals. But in addition to these subgoals, it is possible to include more information by instantiating one or several of the six growth points. For Hovy’s system to work, though, it is assumed that the explanation content has already been selected and that the operators is supposed to include as much as possible of the selected source model parts into the explanation. The only restriction is that the explanation should not repeat information; that is, the same instantiated goal must not be included several times. In Figure 6.5 we have shown two different explanations based on the same overall discourse goal. In (a) none of the growth points have
Figure 6.5: Two partial deep explanations generated by Hovy's structurer. (a) "Know is C4. It will arrive on 4/24." (b) "Know, which is C4, is heading SSW. It will arrive on 4/24."

been included, whereas in (b) the nucleus growth point CIRCUMSTANCE has been expanded and added to the explanation structure.

Optimizing Deep Explanation

At this stage, the deep generator tries to optimize the explanation structure. The optimization task may be integrated with the structure planning task, but is also often interwined with decisions from the surface generator. The most used technique is to amalgamate similar pieces of information from the deep explanation [115, 131]. Optimization rules are introduced in some systems, but many of these rules are domain-dependent or language-dependent, and it is usually very difficult to clearly define rules of that kind.

The optimization process could equally well take place in the surface generator. The technique resembles techniques used in surface generation (like pronominalization), and it is also partly dependent on what kind of natural language constructions that are available for these compressed explanation structures. In any case, optimization of deep explanations cannot be considered a part of the deep generator core.

Comparison of Strategies and Techniques

In this presentation we have focused on two different discourse strategies, discourse schemas and plan operators. Related to these strategies are a wide range of techniques
that help us improve the quality of the generated deep explanation. A strategic component comprises both an explanation strategy and a number of techniques.

We will here make a few comments on the similarities and differences between the two discourse strategies. Most of the techniques used are related to the standards of textuality in some way or another. Instead of comparing the techniques with each other, we will see how they may enhance the textuality of the explanation. In that discussion, the standards will be treated separately, and with each standard we will associate techniques that are meant to ensure the explanation’s compliance with that particular standard. Included in the presentation of standards is also the generation of multimedia explanations at the deep explanation level.

**Discourse Strategies**  The discourse strategies in today’s multi-paragraph strategic components are represented either as schemas or as plan operators. Discussing the differences between these two approaches, we will below concentrate on efficiency, flexibility, reasoning capabilities, and theoretical basis.

**Efficiency**  Each schema forms a complete generic deep explanation. Although many schemas may be instantiated differently, depending on user properties or coherence considerations, a schema will typically specify one type of explanations. In operator-based systems a planned deep explanation generally includes a structure of several instantiated plan operators. An operator is more primitive than a schema, and the strategic component has to search for and combine operators to construct a complete deep explanation.

**Flexibility**  Flexibility in explanation generation concerns the range of explanation types supported by the component. In schema-based approaches, there is usually only one type of explanation associated with each schema. Even though the instantiations of a schema may differ, the overall structure and content of an instantiated schema is fixed. In operator-based approaches, however, the same operator may be included in a whole range of different explanation types. Since operators only express some formal relationship between text units, and an explanation is considered a hierarchical structure of such units, a set of operators will typically produce a large number of explanation types by combining them in meaningful ways. Hence, operator-based generators can generate considerably more explanation types than schema-based generators can [114].

**Reasoning Capabilities**  The deep explanation generated by an operator-based system tends to be hierarchical, in which the structuring elements relate text units to each other [115, 174]. If these elements are text-oriented, like in [37, 55, 116], the elements are useful when the corresponding surface text parts are to be coordinated. The coherence between units are given by these elements, and the cohesive constructions of the surface explanation may be directly derived from them. Using a text-structuring element like “example” to relate structure α to β, for example, could suggest the use of the cohesive construction “for example” to initiate the surface

---

4The plan operator is more primitive in the sense that it only includes two text units.
text for structure $\beta$. In some systems, effect-oriented or goal-oriented structuring elements are used. These make it possible to reason about the effects of various parts of an explanation. If the user fails to grasp the whole content of an explanation, the system would be able to detect and replan those parts not understandable to her.

The same reasoning capabilities are not available in schema-based generation systems. The deep explanation is a linear structure of content elements, and there is usually no way of knowing how one element is structurally or communicatively related to another element.

**Theoretical Basis** Basically, the design of explanation schemas and plan operators stems from two different traditions. Plan operators are intended to formalize the linguistic notion of textuality, and their design reflects general theories from text linguistics and pragmatics. In principle, the operators are domain-independent and only provide mechanisms for combining text units to more complex communicative text units. RST-based plan operators, for example, make explicit the coherence relation between text units and draw on theories about the structural and hierarchical relationships between elements of meaningful texts.

Schemas, on the other hand, are usually domain-specific. They are constructed on the bases of intensive studies of how experts explain phenomena in one particular domain. They do not claim general validity, although there have been some attempts to form domain-independent generic text structures by generalizing schemas [166].

The constructs of the source model's language are referred to in the discourse strategies, and this makes the explanation component highly dependent on the modeling language. Consequently, the component has to possess knowledge about the language, and this restricts the application of it to that particular language environment. If the modeling language is modified or simply replaced with another language, the schemas and plan operators need to be changed correspondingly.

An assumption underlying schema-based approaches concerns the quality of these models. A schema forms a complete deep explanation structure in itself, and assumes the model to be equally complete. If this is not the case, it may be difficult to instantiate the whole schema and there will be no explanation. It would be feasible to handle incomplete models by designing schemas that are subsets of other schemas, but this would violate the basic principles of schema-based explanation generation. If such subschemas were used, one might just as well abandon the superschemas all together since they are derivable from the subschemas. Using the same decompositional approach recursively to the subschemas, one would eventually end up with a large set of undecomposable subschemas called **plan operators**.

Operator-based systems are usually more flexible with respect to incomplete system models. If some elements are missing, the system will try other operators that refer to elements present in the model. And even if the system model is fragmentary or stated on a very abstract level, it will often be possible to generate at least some superficial deep explanations. The use of planning algorithms in operator-based approaches, thus, yields a more robust and adaptive explanation generation component.
6.2. Deep Generation

Techniques and Standards of Textuality  The standards of textuality affect both the design and instantiations of discourse schemas and plan operators. We will now see how the most usual techniques in deep generation relate to the seven standards. Many of the techniques are used in the subtasks of content selection, structure planning or explanation optimization and were used as illustrations in the previous sections, so we will only draw the main lines here. In addition to the standards of textuality, there will be an assessment of how various types of medium are combined to improve the quality of explanations.

Cohesion  Cohesion is associated with the actual wording of explanations and is mostly handled by the surface generator. However, in deep generation there have been three approaches to help the surface generator meet the standard.

First, it may be possible to include keywords that associates explanation structures with cohesive devices. This is done in Hovy’s structurer [116] and Harrius’s AREST system [100], in which every RST relation includes a set of cohesive devices that can be used to connect its realized nucleus and satellite. Relating two propositions by means of motivation, for example, suggests that the corresponding sentences be connected using the cohesive devices “because” or “as” [100].

Another way of improving cohesion is to determine the topic of every content element. Kalita uses four rules to do this, and each sentence topic is later realized as the subject of the produced natural language sentence\(^5\) [131]. In TEXT the same set of rules are used to decide what information to include in the explanation [168]. This means that only cohesion-preserving information can be added to TEXT’s descriptions. And in EES the rules are used to find cohesion-preserving interpretations of vaguely articulated follow-up questions\(^6\) [174].

Coherence  In schema-based deep generators, the schemas are usually based on intensive studies of large bodies of coherent texts. The design of these schemas, and the methods of instantiating them, are supposed to ensure that the text be coherent [168, 165, 189, 166].

In operator-based generators, the design of the operators may reflect some theory of text coherence. This is the case for text-structuring plan operators like RST relations, in which a unit can only be included into the explanation if it is structurally related to some other unit [174, 188, 116, 221, 37]. A theory of coherence is also given by McKeown et al. [169], and this theory can be used both to instantiate schemas and to make RST relations more flexible in text generation [116].

Intentionality  The intentionality of a piece of text expresses the system’s intention of sending that particular piece of information. Most systems do not explicitly represent this aspect of communication, but rather assume the intentionality to be given from the content of that piece of text. Since that is not always the case (see for instance [37, 174]), some recent systems have tried to extend the machinery to model intentionality on a well-founded basis.

Looking at the systems that do represent the intentionality of texts, we can find two ways of doing this: (1) The intentionality is marked using intentional discourse

---

\(^5\) Topic is called "focus" by Kalita, and the four rules are referred to as "focus rules".

\(^6\) Making sure that the standard of cohesion is met usually indicates that the text is also coherent.
goals [174, 188] like "PERSUADED"; (2) intentionality is marked representing higher-level communicative acts like "deny" [165] or "explain" [37, 163].

**Acceptability** The acceptability of explanations is particularly important in intelligent help systems. In these systems, explanation type is chosen on the basis of inferred user goals (e.g. directly in UC and EUROHELP and indirectly in TEXT) or inferred user misbeliefs (e.g. ROMPER and Aqua) [130, 165, 170]. The same goal-oriented approach to acceptability has also been applied in some expert systems (e.g. [18]).

Also, acceptability concerns the content of explanation, and in TEXT and ROMPER only information relevant to user’s goals can later be included in explanations. This is done defining a relevant knowledge pool and a domain view, respectively. When explaining the behavior of executable Gist models, the system keeps a record of events that may surprise the users⁷. To make the behavior explanation as relevant as possible, all surprising situations are highlighted in the text [233].

**Informativity** Adapting an explanation to user’s preferences, skills, or level of expertise, the systems have found several strategies useful:

- Techniques are used that prevent information from being repeated in the explanation [91, 116, 174, 221, 233, 234].
- Similar or resembling text segments are grouped to make the final explanation more compact [131, 115].
- The generator tries to use familiar concepts to explain more difficult phenomena [48, 233].
- Facts already known to the user are omitted if they are not necessary to understand the rest of the explanation [46, 71, 174, 197].

In general, user knowledge and user preferences are used to discriminate between discourse strategies [37, 91, 163, 188, 189, 221]. They influence the choice of discourse schema or plan operator, and are in that sense affecting both the content and structure of explanations. The motivation for this is to use terms, concepts and structures that the user would understand, and find a level of abstraction that she would feel comfortable with. If there is no separate user model in the system, there may be static policies encoded that suppress too much details from the explanations [234].

Another technique for user-tailoring explanations is to use discourse features [114]. These features determines the explanation’s style as well as content and structure, and will typically depend on user type.

**Situationality** The situationality of texts is addressed in Hovy’s feature-oriented text generator [114]. These features characterize the conversation setting, making it possible to let the situation affect both the content and structure of explanations.

If the explanation generator is integrated with an executable program (like in expert systems or help systems), the status of the program may affect the interpretation of requests as well as the generation of appropriate explanations. To take an example,

---

⁷Unfortunately, there is no user model in the system. Surprising events are directly encoded into the generator and cannot be changed according to the properties of the user.
6.3. Surface Generation

Table 6.5: Plan operator for multimedia explanations [163]. Header is the operator’s goal, constraints and preconditions its preconditions, and decomposition its subgoals.

when a user asks the system in EUROHELP what she can do now, the system will only mention commands that are available in the system’s current state [101].

Intertextuality In EES, EDGE, and EUROHELP it is possible to ask follow-up questions or interrupt an ongoing explanation process to ask more specific questions. These questions are interpreted in the context of the previous explanation, and the new explanations are constructed using the old one as a basis and point of departure.

All explanation generation systems today are basically text generation systems. Their main task has been to generate coordinated natural language sentences, and they have paid little attention to the supplementary use of graphics.

If act-oriented structuring elements are used, graphical information has been included using operators with visual acts as operator heads. The operator in Table 6.5 indicates how the act indicate-deictically may be instantiated to add pictorial information about the location of a place. A similar technique is used in EDGE. An alternative to the technique of visual acts would be to add features to the operator specifying the medium and the communicative act, but this has so far only been tried in schema-based generation systems (COMET).

6.3 Surface Generation

The surface generator bridges the gap between content elements and units of text or graphics. Taking a deep explanation as input, it uses its linguistic and graphical knowledge to produce a surface explanation consistent with the deep one. Surface generation is not the focus of this work, and we will here only scratch the surface of this very complicated process. The interested reader can consult [85] for a more thorough presentation and evaluation of surface generation systems.

Realizing a surface explanation, the tactical component may need routines for producing both graphical and textual elements.
Graphical elements of explanations are produced from special display commands, like the Display-Region(#{<Karl-Marx-Stadt}>) and Highlight(#{<Karl-Marx-Stadt}>) elements in Figure 5.9. The components responsible for graphical realizations are clearly separated from the textual ones, which may often complicate the coordination of text and graphics. Some graphical components show simple views of an underlying picture or model (e.g. [38]), some generate new pictures on the basis of certain input data and graphical primitives (e.g. [74, 206]), and some highlight parts of the picture for the coordination of text and graphics (e.g. [38, 143, 163]). Still, most systems today — and this includes all text generation systems in information systems engineering — are not able to supplement their textual descriptions with graphical presentations.

Textual elements are realized in a two-stage process, inter-clause realization and intra-clause realization [8, 126]. Both depend on linguistic knowledge, and both make use of default values when the deep explanation is incomplete. The responsibilities of these two processes are as follows:

Inter-clause realization In this process, those parts of the explanation that depend on the deep explanation structure are taken care of. The tasks include structure optimization, theme control, sentence scoping, and clause coordination [160].

Optimizing deep explanation structures, one exploits the same kinds of techniques as in the optimization stage of deep generation, like pronominalization and amalgamation of explanation parts. In theme control, hierarchical deep explanations are linearized (see [8, 205]), and the text's topic and focus development — if not already indicated — is determined (see for example [131]). If RST structures are used, linearization simply means to decide the ordering of nuclei and satellites. Deciding the sentence scope, one checks which elements should be combined to form complete sentences [205]. Clause coordination is relevant if the clauses of the deep explanation are related by means of rhetorical relations. To signal the textual relationships between text units, it is often useful to realize the rhetorical relations as cohesive constructions, though the choice of constructions can sometimes be pretty difficult to make [8, 205, 213].

Intra-clause realization This process deals with the realization of the deep explanation's content elements as natural language clauses. According to Hovy [117], there are four types of surface generation approaches: canned texts, templates, phrases, and features. The first two are simple, but rather inflexible, whereas the last two rest on extensive linguistic knowledge and are referred to as grammar-based approaches to surface generation [85]. On the basis of lexicons and grammars, these two divide the realization into the subprocesses of choosing lexical items, choosing grammatical constructions, and making morphological adjustments.

Good introductions to surface generation systems are given in [117, 171], and some newer issues related to grammar-based systems are addressed in [190, 258]. Of particular interest to this work are also the experimental surface generators founced on Functional Grammar representations [10, 126, 210].
Chapter 7

Requirements to Explanation Generation in Conceptual Modeling

In information systems engineering, there are parties involved that have different backgrounds, skills, and responsibilities. These people are taking part in most of the development activities, starting from the construction of information system model views to the validation of complete, executable information system models. An explanation component that is to support the work in conceptual modeling must support all the main activities as well as all the main parties involved. In this chapter we work out the requirements to an explanation component. These requirements concern the component's explanatory power and its use in CASE environments:

A complete explanation component is not the aim of this project, so after the establishment of requirements, we will make the necessary simplifications and restrictions to reach a manageable level of complexity.

7.1 Explanation Needs

In conceptual modeling, explanations are useful in at least two areas:

- construction of models, and
- validation of models.
These two activities are common to most contemporary environments for conceptual modeling. The construction and validation of conceptual models are both vital to the process of developing information systems, and their results both depend on humans' understanding of some modeling language and model representation.

In principle, we would like all kinds of explanations to be available in the environment, but this attitude would interfere with the fact that some explanations are more important and basic than others. Whereas some explanations are absolutely necessary for their exploitation in conceptual modeling environments, others are more supplementary of nature and can be compensated for generating other explanations or using other techniques. In this project, thus, we only look into the most basic explanation needs in conceptual modeling, and a list of necessary explanation types are given below.

Also, as far as possible we will avoid explanation types that requires special routines or representations not usually included in CASE environments. Comparative explanations, for example, will not be considered here, since they require routines for comparing model elements and can be compensated for using several centered explanations. The same applies to for example corrective system behavior explanations. CASE environments do not usually have models of the information system's use, and we will consequently not assess the usefulness of user abilities explanations either. The explanation needs assessed here, thus, are restricted to what is supportable — without the addition of extra functionality — within current CASE technology.

**Construction of Models**

Effective model construction requires that the analyst understand the semantics of the modeling language and know how constructs of the language are combined with each other, and that the results of tool-supported verification checks are made available to her. The explanation component should provide individualized descriptions of semantic and syntactic properties of the modeling language and present messages from verifying the model in terms understandable to her.

**Language semantics** Explanations of the semantics of modeling languages are terminological system theory explanations. To the extent that the information is available in the source model, behavior explanations, functionality explanations, and rationale explanations should be offered.

**Language syntax** Explaining syntactic properties of the modeling language, the environment must be able to generate terminological structure explanations. Explaining structural properties of concepts, the explanation component must be able to make use of principled, theoretical and illustrative explanation parts.

**Verification checks** The explanations here are terminological, illustrative, corrective descriptions of illegal model constructions. They should reject the illegal model construction, which is also included in the explanation as an illustration, and explain why the construction is not allowed.
As seen from Section 5.4, generating the explanation types above, the system needs explicit representations of both meta model and conceptual model, and of course a verification component that can compare elements of these two models. Otherwise, no additional routines or representations are needed.

Validation of Models

For model validation to be successful, all stakeholders should fully understand the conceptual model and know the consequences of using it as a design and implementation goal. The explanation component should offer a front-end to the conceptual model and the recorded modeling deliberations, such that the presentations' complexity, focus and notation are tailored to the context and the people receiving it. Similarly, executing conceptual models, the environment should offer explanations that help the users understand what is going on and that justify the model's behavior. The specific explanation types needed for model validation are summarized below.

**Model inspection** In this group, we need intensional system theory explanations. The explanations must cover the whole conceptual model, they should have access to both underlying generic principles and illustrative model instances, and they should be able to include views of the conceptual model as illustrations.

**Modeling deliberations** As a means for presenting modeling deliberations, the component must provide rationale explanations and element justifications.

**Model execution** Explaining and guiding the execution of a conceptual model, the component must at least offer input justifications and history explanations. Other types of system behavior explanations may also be useful, but these require either extra information in the conceptual model and/or extra routines for generating their content. The explanations must be able to relate executions to model elements — that is, they need access to generic principles and graphical views of the conceptual model.

The explanation types above requires that conceptual models and execution traces be available. Moreover, their quality is improved if views of the conceptual models can supplement their textual parts.

### 7.2 Integration Requirements

CASE environments are quite different from systems in which most current explanation generators are used. Their complexity, generality and flexibility have made them powerful instruments in information systems engineering, but the same properties complicate the introduction of explanation facilities into the systems.
The most important difference between these systems and traditional systems possessing explanation facilities concerns the source model. Characteristic to CASE environments is the unstability of the source model, in which both the conceptual model and sometimes also the meta model may be modified or substituted at any time. This means that the major source of knowledge to the generator cannot be assumed to be stable, and the generator has to be very flexible to adopt to the changes around it.

Going into some more detail, we get four integration requirements, and these concern the generator's applicability, its flexibility, its isolation of generation decisions, and its tailoring of explanations. Below, each of these requirements are discussed.

Generator Applicability

CASE environments are used to develop computerized systems within the information systems domain. Various languages may be used in a project, and the content of the models constructed cannot be known prior to the modeling process.

The explanation process must therefore be applicable for several modeling languages and independent of the content of the source model constructed. Specifically, it is not possible to assume any particular elements to be present in the source model.

Generation Flexibility

In an increasing number of CASE environments, there are several modeling languages available. These languages may be used in different projects or phases, as is the case for CASE tools with meta modeling capabilities, or they may be used simultaneously and integrated to reach a more precise specification of the system.

In any case, it is impractical and tiresome to specify the possible explanations for each of these languages. The explanation generation component should have a fixed language-independent kernel to produce the most basic explanations. When a new language is introduced, we should be able to use the kernel directly — without any additional explanation modeling — to generate explanations based on the language.

However, since the expressive power of languages differ, we can hardly ever expect to have a generator kernel that is able to use every construction of every language in its explanations. Hence, it is important that the explanation generator can easily be extended to work on the more rare constructions of a particular language.
7.3. Requirements to Explanation Component

Isolation of Generation Decisions

It is the nature of conceptual modeling that models are subject to frequent modifications. In some recent tools, the modifications can even be done on the modeling language itself, making it more suited for the domain, the involved parties, the implementation environments, etc. Usually, the whole language is substituted, but it is also imaginable just to make some slight alterations to the language in use.

Of course, to generate explanations related to these modifications, the generator needs access to the modifications being made. We cannot expect the analysts to know the internal working of the generator, though, so we cannot assume the modifications to be stated in terms of their effects on the explanation generator. The source model must be assumed to be represented using terms from the modeling world alone.

Still, the explanation generator should be able to use these representations without any additional information about how they fit into explanations. *All generation decisions should be isolated to the generator, such that the source model does not need to refer to elements outside the modeling world!*

Tailoring of Explanations

Different types of users are not only different in level of domain expertise and language acquaintance, they have also different reasons for being involved in the project and are interested in different aspects of the source model.

Successful development of information systems usually requires that all the involved parties be contributing to the result. Various types of knowledge are involved to build a satisfactory system, and most of these types are reflected in the conceptual model. Since no single person can be assumed to know the complete model in advance, and every person's contribution might be vital to the result of the project, it is important that all the parties understand their relevant model representations.

Adapting explanations to user characteristics is crucial in conceptual modeling. *The generator must be able to tailor its explanations to improve user's comprehension and commitment.*

7.3 Requirements to Explanation Component

A full explanation component, as alluded to in the previous sections, would be ready for inclusion in most current CASE environments. It would be connected to the languages and models used in the environment and could automatically produce *system theory explanations*. If the models were executable, it could also provide *system behavior explanations*. 
A good generator would meet the following overall requirements:

- The architecture of the component complies with the four integration requirements from Section 7.2.
- Provided that the source model contains the relevant knowledge, all the explanations needs from Section 7.1 are satisfied.
- The explanations generated meet all the seven standards of textuality and can include graphical elements to improve their presentation (see Section 5.2).

Building such a component is a very ambitious goal. It involves intimate knowledge of information systems, modeling languages, planning methods, and linguistics. Previous explanation generation in conceptual modeling has been restricted to intelligent language-dependent paraphrasing of model and trace (e.g. [232, 233]). There have been no attempts to construct a full explanation component, so the project would only to a limited degree be able to draw from previous research. The technology of explanation generation is still immature, making it difficult and perhaps even undesirable to produce a full-fledged explanation generator in one step.

In our opinion, a step by step approach would be the sensible way of constructing the component. Formulating a first step, then, there are two main concerns: (1) The completion of the step should make it easier to proceed the project, and (2) the result from the step should be amenable to a proper evaluation. Our first step towards an explanation generator in CASE environments is then described as follows:

- We will concentrate only on the strategic component core of the generator. The output generated by the component, the deep explanation, must be represented in a language that is well suited for surface generation.
- The focus is on representational issues. Our aim is to examine how the generator could be represented in order to generate the necessary explanation types. Specifically, it must be possible to represent and use the information affecting the textuality of the explanations. Also, within the expressive power of the modeling languages, the needed explanation types from Section 7.1 must be representable in the same language.
- The strategic component is to meet the integration requirements from Section 7.2.

Using this approach, we are able to divide the construction of the explanation generator into at least three independent tasks: (1) construction of deep generator, (2) construction of surface generator, and (3) text studies of how actual explanation structures are related to the standards of textuality. Each of these parts can be partly evaluated without the access to the other ones. The objective of this work is to carry out the first step of introducing explanation generators in conceptual modeling. The focus of the step is the representation of knowledge and explanation strategies in the strategic component of the generator.
Part III

The Deep Explanation Generation Approach

In Chapter 8 through 11 a strategic component core for conceptual modeling is gradually developed. The chapters are as follows:

**Chapter 8** A general CASE architecture including an explanation component is described. In this architecture, explanation generation is integrated with model verification, model execution, and view generation, and the explanations are used in the construction as well as in the validation of conceptual models.

**Chapter 9** Our principles for generating deep explanations in conceptual modeling environments are presented.

**Chapter 10** A language *EML* for representing all explanation-relevant knowledge and realizing the principles in Chapter 9 is defined.

**Chapter 11** A simple strategic component core is developed for the system described in Appendix A. The generic part of the core — that is, its strategies for generating explanations — is used to build an explanation component for PPP in Chapter 12.

Both the architecture and the explanation component are intended to be compatible with the architecture of contemporary CASE environments.
Chapter 8

A CASE Environment with Explanation Facilities

CASE environments today do not have any explanation facilities. They can help the users to build models and are able to perform a number of analyses or transformations on these models, but extensive explanations on how to use the modeling language or how to interpret the models are not available. However, some environments make it possible to alter the tool’s modeling language or to execute models of the future information system.

In this chapter we use the architecture of contemporary environments as a basis for including explanation facilities. By extending the relevant parts of these existing architectures, we can construct an explanation component that supports the construction and validation of information system models. We will first assume a very flexible CASE architecture with meta modeling capabilities, but at the end of the chapter we also discuss some more restrictive architectures.

After describing the scope of the architecture, the main components and model representations are laid out. The internal structures of the explanation component are presented, and its use in constructing and validating conceptual models is indicated.

8.1 Architectural Basis

Our goal here is not to outline a complete CASE architecture, but rather to explain how an explanation component would interact and co-operate with other components of existing environments. Components of CASE environments irrelevant to the explanation component are neither discussed nor included in the architecture. As such, it comprises, on the one hand, a subset of contemporary CASE architectures, and, on the other hand, an explanation component that is integrated into this subset architecture.
Our CASE architecture concentrates on the conceptual modeling part of information systems engineering. It is intended to be compatible with existing CASE environments in the sense that it should show how the explanation component could be introduced in already well-known environments. In the presentation to follow, then, we make the following assumptions about CASE architectures:

- The same model representations can be accessed and used by many components of the environment (data integration [238]).

- A well-defined graphical modeling language is used to build models of the information system. The language is defined in an explicit meta model, and we assume that this meta model can be modified to alter or replace the environment's modeling language.

- There is an executable language included that can be used to exercise conceptual models and build traces from these executions. We assume that neither this language nor its trace is used directly by the users. The conceptual model constructed by the analysts is translated to an executable representation, whereas the trace is afterwards made available presented in terms of instantiations of conceptual model elements.

This is not to say that environments that do not satisfy the assumptions are not suitable for explanation generation. Explanation generation may still be both feasible and desirable, but the architecture will then either be simpler and less powerful, or more loosely coupled. At the end of the chapter, we will return to the assumptions above, discussing cases in which one or several of the assumptions are not met.

### 8.2 Representations and Components

At a superficial level, CASE environments encompass model representations and some major components that work on these representations. There are interfaces between models, components and users, but the complexities of most of them are not important to the discussion to come. The architecture suggested here is depicted in Figure 8.1. There are five interrelated components, and they make use of meta models, conceptual models, traces, and model instances. At the meta model level, various kinds of properties of the environment's modeling language are stated. The conceptual model level is used by the users to model domain properties and information systems, and at the lowest level, traces are built and databases filled as the conceptual model is being tested.

The model representations of the environment are described as follows:

**Modeling meta model** The modeling language has a syntax and a semantics defined. The syntactic and semantic requirements to the language are used by the editor and the verification component to help the users in the modeling process. They ensure that
only concepts of the modeling language are included in the model, and that these concepts are used in the correct manner. Consistency checks and completeness checks depend on this information about the modeling language.

All syntactic and semantic requirements to the language are kept in a \textit{modeling meta model}. It is used by all the five components of the architecture, and the elements of the conceptual model are instantiations of meta model elements.

We leave open the possibilities of changing the modeling meta model, so that the environment's conceptual modeling language may be modified or replaced with another one.

\textbf{Conceptual model} The conceptual model is constructed using the modeling language defined by the modeling meta model, but it is not necessarily a directly executable language. We assume, though, that the language's expressiveness makes it feasible to translate its models to formalisms that are executable.

At least some of the users should be able to understand and use the language. Since expressive power and formality are important here, we cannot expect users unfamiliar with the modeling language to understand all facets of the language and its models.

\textbf{Instances and traces} If there is a database submodel of the conceptual model, database records may be created to test the quality of the database. The records are instances of structures of the database submodel and are later inspected by users to see whether the system will be able to represent the necessary data efficiently.

We assumed that the conceptual modeling language — or a sublanguage of it — be translatable to an executable representation. When executing this representation, the environment builds traces that document the model's dynamic behavior. Repre-
sentationally, elements of these traces are viewed as instantiations of the conceptual model.

The user has access to a syntax-directed editor to build conceptual models. Syntactic and semantic knowledge from the modeling meta model governs the modeling process, and a priori verification rules are enforced throughout the process. The constructs of the modeling language are made directly available to the user, and the model is constructed by picking out and instantiating the available constructs.

A verification component makes sure that the conceptual model comply with formal syntactic and semantic rules of the modeling language and that specification parts be consistent with each other. Both a priori and a posteriori verification rules are defined, and these are utilized by the editor and the user, respectively. The verification component formulates requests to the explanation component when violations of a posteriori rules are detected.

A view generation component displays views of the conceptual model, using both horizontal and vertical abstractions (see Section 4.1). Several kinds of views of the model are defined, and these views correspond to what model elements are considered relevant to a particular context or a particular user. Rather than operating on a full model, the user is presented a view of the model that include only the parts that are of interest in that situation. Since not all of the original model is used, it will sometimes be advantageous to restructure the view elements to improve its layout. In sum, then, the component provides means for suppressing irrelevant details of the conceptual models and highlight relevant details.

The views defined for the component must be shared by the explanation component. Graphical explanations can then be generated by invoking the proper view generation routine. In addition, of course, the users can exploit the component to produce their own views of the conceptual model during the modeling process.

The execution component includes some executable representation language that can be used to represent and execute information conceptual models. This language can be a programming language (e.g. Ada [84]) or a modeling language with an operational semantics (e.g. StateCharts [96]). When a conceptual model is to be validated, the corresponding executable representation is constructed and executed. The users can interact with the system in the execution, and the execution history is recorded as a trace related to elements of the conceptual model.

The executable representation language is a fixed language and is not changed as the user’s modeling language is modified or replaced. It must meet the following requirements:

- It must have the necessary expressive power to represent all the kinds of operationality in information systems that are exposed and validated at the modeling stage of the development project.
- There are no requirements to the user-friendliness of these executable representations. The representations are hidden to the users and are only used internally by the
environment to explore the behavior or conceptual models.

- During execution, the component must be able to effectively build traces referring to elements of the conceptual models.

Since the execution component’s representation is different from the environment’s modeling language, we need translation rules between the two formalisms. If the modeling language is modified or replaced — that is, the modeling meta model is altered — these changes must be reflected in the translation rules.

In the explanation component, explanations are generated to help the users understand the modeling language or the properties of the conceptual model. During execution, user inputs and trace results can be justified and explained. The component accepts explanation requests from the users and the verification component, and the explanations are combinations of texts and views of graphical parts of the conceptual model.

In a real CASE environment, there may also be components for versioning, project control, simulation, code generation, meta modeling, etc. All these components are useful, but since we are concentrating purely on the inclusion of explanation facilities, we will only discuss the five components emphasized above. The verification component and the execution component are necessary for generating the inputs to the explanation component, the editor helps us understand how the other three components work together.

8.3 Internal Structures of Explanation Component

In general, the conceptual modeling language is not suited for explanation generation. Instead of working directly on the conceptual models, we are in the explanation component using a language called EML to represent model information relevant to the generation of explanations. This language is specifically designed for the planning and production of explanations, and is explained in detail in Chapter 10. The conceptual model represented in EML is referred to as the explanation conceptual model.

The meta model specified in EML describes concepts of the modeling language at a level of detail that is appropriate for natural language paraphrasing. Details of the real meta model are omitted if they are deemed too peripheral to the explanations needed. The EML meta model, which we call the explanation meta model, is not a complete definition of the conceptual modeling language, but suffices to generate explanations of the most important features of the language. The explanation conceptual model is formed from instantiating elements of the explanation meta model, whereas the explanation trace contains instances of the explanation conceptual model. The meta model, the conceptual model and the traces — all represented in EML — are referred to as the explanation component’s source model.

The explanation component rests on a taxonomy of possible model structures in the domain and the definition of conceptual model views. It is subdivided into two connected subcomponents, the strategic component and the tactical component.
Strategic component In the strategic component the source model and some internal deep generation knowledge sources are used to generate deep explanations that are passed to the tactical component for realization.

Tactical component In this component, textual and graphical surface explanation parts are generated. In addition to deep explanations, the component uses some internal surface generation knowledge sources (like a lexicon) and graphical routines for presenting conceptual model views. The graphical routines are shared with the editor of the environment.

In the strategic component, there are tasks that are independent of the tactical component. The implementation of these processes is referred to as the strategic component core. As indicated in the figure, results from the tactical component may affect deep generation processes outside the strategic component core (the dotted arrow). These results may impose changes to the initial deep explanation generated by the core or add details to an incomplete deep explanation.

If the user wants to use another conceptual modeling language, the explanation meta model must be revised to reflect the new modeling language. This would also influence the rules used to translate conceptual models, model instances, and traces to EML.
8.4 Generality of Architecture

In the general architecture outlined above, there is an explicit modeling meta model and a separate executable language in the environment. Even though this generality complicates the structure and perhaps even hamper the effectiveness of the environment, the architecture has several advantages to more restricted ones.

- The environment is not restricted to any single conceptual modeling language.
- Having an executable representation language hidden to the users, we need not consider the understandability of the language. Instead, we can fully concentrate on developing an executable language that is effective and efficient to execute, has the expressive power to model all relevant operational properties of information systems, and is suitable for trace generation. A language like this is suggested in [249].
- The explanation component requires that an explanation meta model be constructed. Since the whole environment is based on meta modeling, this would neither necessitate a set of additional meta modeling functions nor a new consciousness about meta levels on the users' part.

Two deviations from this architecture are often found in CASE environments of today: (1) There is no explicit modeling meta model, and (2) the conceptual model is executable in itself:

No explicit modeling meta model In most CASE environments today there are no explicit modeling meta model. The syntactic and semantic requirements to the environment's modeling language are just added as code fragments to the appropriate components of the environment.

If the explanation component outlined here is to be integrated into one of these environments, the potential of the component will not be fully exploited and the integration will require some more work:

- The explanation component is designed to work on arbitrary modeling languages, and in order to do so an explanation meta model is built as part of the integration process. If there is no explicit modeling meta model, the environment cannot alter its modeling language, and it does not need a language-independent explanation component.
- Since no modeling meta model is used, the environment will lack meta modeling facilities. Prior to the specification of an explanation meta model, then, we must add functions for the modeling and representation of modeling languages.

Still, we would be able to generate the same types of explanation and the explanations would be of the same quality.
No separate execution language If the conceptual modeling language is executable in itself, we can use this directly instead of a separate executable representation language. This solution yields a simpler architecture, since an additional representation language is no longer needed in the system. However, the executable part of the conceptual modeling language is now fixed, and we cannot freely change the modeling language to better suit the users or the domain. Besides, it is an open question whether it is possible to find a language that is suitable both for the modeling and the execution of information systems. While a modeling language should be comprehensible and easy to use, the two main concerns of the executable language here are effectiveness of execution and feasibility of trace construction.

We will here not choose between the general and the two more restricted architectures. The explanation component is applicable in all three, although it is conceptually most oriented towards the general one.

The other assumptions from Section 8.1, however, are more vital to produce satisfactory and useful explanations. The whole concept of the architecture assumes the components and modules to be tightly integrated. And if there are no graphical symbols in the modeling language, the desired graphical explanations cannot be generated. If there are no executable language in the environment, we cannot exercise models of the information system and explain the actual behavior of these models.

8.5 The Conceptual Modeling Cycle

In this environment, conceptual models of the future information system are iteratively constructed and validated. A business area — or rather some stakeholders' perception of a business area — feeds the initial construction of the model. Since the model is meant to capture and clarify domain properties and system requirements, it is vital that the necessary business knowledge is brought into the modeling. Verification checks make sure that the conceptual model be formally correct as the validation step is initiated. Validating the conceptual model, the stakeholders assess its qualities as a rendition of the relevant aspects of the business area or the requirements at hand. The detection of incompleteness between model and stakeholder's perception or inadequacies of model's precision and scope will spur new cycles of model modifications and validation. The conceptual model is then iteratively modified and validated until it is deemed a satisfactory representation of the existing or future information system. In Figure 8.3, the overall modeling cycle is illustrated.

The CASE environment supports both the construction and validation of conceptual models. The techniques made available to the users are tightly integrated with the model and are used actively throughout the conceptual modeling process.
Construction Support

In the construction step, the environment's focus is on how the language is utilized to build the conceptual model. Whereas the task of the users is to structure and make explicit properties of some information system, the CASE environment provides functions that help them work out proper representations in an efficient manner. In doing that, the environment can inform the users about formal qualities of their models and explain how the modeling language is to be used:

- If the user attempts to violate an *a priori* verification rule, the environment interrupts the modeling. This function is naturally integrated into the editor explained earlier in the chapter.

- Checks on *a posteriori* verification rules are performed on requests from users. If illegal constructions are detected, error messages are formulated as graphical explanations and presented to the user. The model view incorporated into the explanation shows the illegal part of the model.
At any stage of the modeling process, the user may ask for explanations that clarify syntactic and semantic properties of the modeling language. Views of the conceptual model may be included in the explanation, and these will then illustrate the use of language constructs.

The explanation component, thus, forms a front-end to user-initiated verification checks and language help facilities. The people involved in the modeling do not need an intimate knowledge of the modeling language, but can of course take advantage of the functions to acquire a deeper understanding of it.

Validation Support

Validation can start when a verified conceptual model is available. The techniques implemented in the environment help expose properties or aspects of the model that can be hard to detect only from reading the model. Moreover, due to the integrated nature of the environment, techniques can be combined to further enhance their usefulness.

In the validation step, the stakeholders may make use of the following techniques supported by the environment:

- Views of the conceptual model may be generated and presented to the stakeholders. Irrelevant model details are hidden and relevant details highlighted to simplify the interpretation of the model.

- Having reached a certain level of detail, the conceptual model can be translated to a complete executable program. Exercising the program, the stakeholders can explore the model’s observable dynamic properties and reveal any discrepancies with the expected behavior. Also, with the help of a tracing mechanism, the internal behavior of the model can be recorded and later inspected by the people involved.

- The user can ask the environment to explain properties or aspects of the conceptual model. Being focused on just the relevant parts of the model and presented in terms and structures tailored to the particular user and context, the explanations are assumed to be easier to understand than the model itself. Executing (a translation of) a conceptual model, the environment can also generate explanations that justify computed outputs and requested inputs. The component integrates the view generation technique and the execution technique, so that both model views and information from execution traces can be incorporated into the explanations.

These validation techniques address properties of conceptual models that makes them hard to grasp for unskilled users. In particular, the models’ complexity is reduced, they are paraphrased in natural language, and their deeper semantics is given a very tangible presentation through execution mechanisms. Involving various kinds of stakeholders in the validation process will be easier, since they are no longer required to deal only with the conceptual model representations.
Chapter 9

Principles for Deep Generation in Conceptual Modeling

The strategic component core takes care of those parts of the deep generation process that are independent of surface generation, and is the focus of this work. In this chapter, we explain the principles and structures of a core for conceptual modeling. Examples from the PrM, model in Appendix A are used to illustrate the generation approach.

9.1 Basic Principles

The generation of deep explanations is initiated by an explanation request. This request asks for information related to the modeling language, the conceptual model, the execution trace, or the recorded model instances — that is, information in the source model — and can be posted by the user or the CASE environment itself.

A number of discourse strategies is used in the explanation generation process. For each type of request handled by the component, there is at least one strategy defined. A strategy determines which source model elements to include in the explanation, and how these elements should be structured textually. It can either select the elements itself or it can invoke substrategies that each generate parts of the total explanation. The substrategies, in turn, may also invoke new strategies, thereby forming a hierarchy of instantiated strategies.

Generating a deep explanation, the strategic component analyzes the request and select an appropriate strategy. The strategy is instantiated and executed, which means that it collects elements from the source model and/or invokes substrategies for the generation of parts
of the explanation. This process continues until there are no more substrategies left, and the hierarchical structure of instantiated strategies and collected source model elements form the resulting deep explanation. Throughout the generation process, there are tests that prevent the same source model element from being repeated in the explanation.

To enhance the quality and flexibility of the explanation component, the following improvements have been made:

**Formulation of discourse strategies** The discourse strategies form decompositions of discourse goals into simpler discourse subgoals. A requirement for defining a strategy is that there is a clear rhetorical relationship between its different subgoals (between nucleus and satellite), so that the explanation generated is coherent.

**Tailoring of explanations** User's knowledge of the source model, user's preferences, and the explanation context form three separate views of the model. All together, these views define what parts of the source model are relevant — or can be assumed to be
relevant — in the generation of a particular explanation. The rule is then that only relevant parts can be included in the explanation. Similarly, the views decide what discourse strategies can be used in the generation process.

**Inclusion of graphical elements** A visual display act can be used by the strategies to show graphical views of the source model as part of the explanation. This act corresponds to a routine from the view generation component.

**Generality of component** All elements of the source model are given references to a fixed generalized meta model. The discourse strategies specify these references instead of the elements themselves, which means that the strategies can be used on several meta levels and on several modeling languages.

The main principles of the deep generation process is illustrated in Figure 9.1. In practice, though, there is no separate stage in which the relevant discourse strategies and the relevant part of the source model are sorted out. As will be shown in the rest of this chapter, the whole generation process is administered as one integrated planning process, where the relevance of strategies and model elements is determined on the way.

### 9.2 Architecture

The architecture is comprised of a deep generation process and various sources of knowledge. The sources are used as inputs to the generation process and reflect properties of the source model, the users, the explanation context, the explanation itself and the discourse strategies. In addition, we assume the existence of a domain structure model and the availability of routines for showing predefined views of the system model. The representations exploited in the architecture are given in Figure 9.2 and are briefly explained below.

**View definitions** A view is defined by a name and a list of source model elements (or element types) included in that particular view. When a new element is instantiated from a generic element, the instance will — as a default — be associated with the same views as its type. For example, the generic construct “process” in the modeling meta model may be included in a view called process perspective. As long as nothing else is stated, all processes created in the system modeling process are also added to this view.

**View generation component** This component is drawn in dotted lines, which indicates that it is not a part of the strategic component core. Its purpose is to display graphical views of the conceptual model being constructed, and the name of these views are defined centrally in the environment. A routine of this kind, which we will call a view generation routine, is given in the form

\[
\text{display}(<\text{list of elements}>):<\text{list of views}>
\]
meaning that <list of elements> must be included in the view, and all the elements displayed must be available from the view specification in <list of views> [250]. The routine, thus, is supposed to include additional elements to form a coherent unit of information. For example, executing the routine display([p1,p2,p3],[BEHAVIORAL yes] (see Figure A.2), the system would make the behavioral perspective more apparent if also the control flows connecting them are shown.

**Domain structure model** The domain structure model is a generalization of meta models of conceptual modeling languages. A specific explanation meta model is defined by selecting and specializing elements from the domain structure model.

**User model** The user model encompasses two submodels, the *user knowledge* submodel and the *user characterization* submodel. The user knowledge submodel is an overlay model to the source model, in which all the source model elements known to the user are marked. The user characterization submodel associates the user with a number of views. The views correspond to a role, a category, a background, a goal,
9.2. Architecture

a task, or any other user property that deems some parts of the source model relevant
and others irrelevant. The user model is supposed to be individual and dynamic, but
we will her just assume the existence of that model without going into the details of
how the model is to be created, modified, or stored.

Explanation context Apart from being indirectly derived from the execution trace, the
context is specified as a set of views. More precisely, a context is defined in terms
of what source model elements are considered relevant to explanations generated at
this particular point of time, and this is encoded as views. In the structure context,
for example, we are only interested in structural properties of the model, and only
information pertaining to its structure should be included in the explanation.

Explanation request This is the representation of user’s explanation need as well as ini-
tial information necessary to construct an appropriate explanation. The request in-
corporates the views encoding the user characterization submodel and the expla-
nation context, since these affect the content of the explanation. Furthermore, it
includes a strategy for how the knowledge of the user should influence the determi-
nation of explanation content.

Discourse strategies A number of discourse strategies are defined and represented as
modified plan operators. Normally, a strategy includes a discourse goal, a rhetorical
relation, a number of subgoals, and conditions for using the strategy. The subgoals
form a decomposition of the discourse goal and may refer to the discourse goals of
other strategies, elements of the source model or displayed views of the conceptual
model. We distinguish between two classes of subgoals, called nucleus and satellite.
Both of them may be present in a strategy and may specify a single subgoal, a list of
necessary subgoals, a choice among subgoals, or even a list of subgoals from which
any number may be used. The notions of nucleus and satellite stem from the Rhetor-
ical Structure Theory [161], and the specified rhetorical relation makes explicit the
relationship between these two classes of goals. In the relationship, the nucleus is
given the most salient position; that is, the satellite tends to depend on the nucleus
and say something about it, whereas the nucleus can be interpreted independently
of the satellite. In discourse strategy properties in Table 9.3, for example, the de-
tonation of an element is assumed to provide an interpretation of it. In general, the
discourse strategies glue together units of information that can form coherent texts
and indicate the corresponding cohesive constructions.

There are two types of conditions, a precondition and a feature condition, and both
govern the use of the strategy. The precondition checks that certain properties of
the source model hold and binds the variables used in the discourse goal and the
subgoals. The views associated with a discourse strategy, and the user’s knowledge
of it, are specified in the feature condition. As opposed to the precondition, the
feature condition not only decide whether this particular strategy should be used,
but it also affects the inclusion of other strategies or content elements subordinate to
it. Unless later strategies revise the feature condition, the expansions of the subgoals
down to content elements cannot include strategies or content elements violating the
feature condition.

Extended source model The extended source model is a source model that includes the
view definitions and the user knowledge submodel. The source model itself is repre-

presented in a special explanation-oriented language and includes an explanation meta model, an explanation conceptual model, explanation model instances, and explanation traces. All the elements of the source model are classified by referring to elements of the domain structure model used here. For example, the construct process refers to the more general element dynamic_concept of the domain structure model. The submodels of the source model do not necessarily express exactly the same information as their counterparts in the modeling environment. Their contents are motivated from what is relevant to explanation generation, which means that some details may be left out while others may be given a more prominent place in the model.

Deep explanation log This is just a list of the discourse goals and content elements included so far in the explanation. The log is updated and consulted throughout the generation process to prevent the explanations from repeating information.

The architecture above also indicates how the strategic component core is supposed to function: A number of information sources are made available to the generation process. The generation itself involves the activities of selecting and structuring pieces of information from these sources.

9.3 Deep Generation Process

The deep generation process is initiated by an explanation request specifying the subject to be explained and possibly the current context and the relevant properties of the user. The result of the generation is a deep explanation that can explain the subject while meeting the standards of textuality. Most importantly, the deep explanation specifies the explanation's content and structure, but also the use of textual versus graphical elements and the elements' communicative functions are paid attention to. The content, structure, and media can be tailored to both the knowledge and the preferences of the user, and to the explanation context. The discourse strategies used to generate the deep explanation makes sure that the text be coherent and that no information be repeated.

In the following, the deep generation process will be informally discussed with respect to the tasks of

- handling explanation request,
- selecting explanation content, and
- structuring deep explanation.

There are no attempts to optimize the generated deep explanation in our current version of the component.
### 9.3. Deep Generation Process

<table>
<thead>
<tr>
<th>EXPLANATION REQUEST</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>subject:</strong></td>
</tr>
<tr>
<td><strong>context:</strong></td>
</tr>
<tr>
<td><strong>user preferences:</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>user knowledge strategy:</strong></td>
</tr>
<tr>
<td><strong>linguistic features:</strong></td>
</tr>
</tbody>
</table>

Table 9.1: Content of a typical explanation request.

**Handling Explanation Request**

An explanation request is made in the form of a command to the deep generator. The content of a request may include:

- a subject, which indicates the overall content of the requested explanation;
- various linguistic features, like tense and pragmatic topic, that are known at the time the request is made and help in the realization of the deep explanation;
- an explanation context as a set of views;
- user’s preferences as a set of views; and
- a user knowledge strategy saying whether the explanation should be affected by user’s knowledge, and if this is the case, whether the explanation should include elements that are known or unknown to the user.

Subject and initial linguistic features must always accompany an explanation request. Information about context, user preferences and user knowledge strategy may be left out, in which case the explanation will not be tailored to the context or the user.

In Table 9.1, we have sketched an informal request to the strategic component core, related to the model in Appendix A. The explanation should describe the structural properties of P3, the context is external, and user’s preferences are given by the visible computational view and the hidden behavioral view. User’s knowledge is not considered relevant to the generation process, whereas the linguistic features indicate an explanation in the present tense with the global topic P3.

**Selecting Explanation Content**

The request’s explanation context and user preferences define a set of views of the source model. Amalgamated, the views form what we can call the initial explanation view, and
Table 9.2: Informal declaration of an element of the extended source model.

this view makes some part of the source model accessible and other parts inaccessible as the deep generation process starts. Accessible parts, then, are elements that do not belong to any view specified as inaccessible in the request, but can at least be assumed to belong to the accessible views. When generating the deep explanation, only accessible source model elements can be included. In this way, the content is restricted by what is relevant according to the context and the preferences of the user.

Additionally, the source model elements may be marked as known or unknown to the user. As specified in the request and possibly further determined in the generation process, parts of the deep explanation can be required to be or assumedly be known or unknown to the user. In some cases, one would only include elements that are unknown or can be assumed to be unknown to the user, whereas in other cases one would like to include known elements to shed light on unknown ones.

In Table 9.2, the informal declaration of P3 can receive Requested loan, Customer data, and New customer data is visible in the view computational and hidden in the restrictive view, and is also unknown to the user. If not revised in the planning process, the request in Table 9.1 could lead to the declaration’s inclusion into the explanation, since it is included in the initial explanation view and the request’s user knowledge strategy is to accept both unknown and known elements. On the other side, if the declaration were included in the restrictive view and not in the computational view, it would not be visible to the request and could therefore not be put into the deep explanation. It must be added, though, and this will be further clarified in the next section, that the explanation view and the user knowledge strategy may be changed as the deep explanation is generated.

Structuring Deep Explanation

A number of discourse strategies are used to select and structure elements from the source model to form a deep explanation. The strategies specify how a given subject can be explained in terms of other strategies, elements from the source model, and displayed views of the model. Implementationally, they are represented as extended plan operators.

For each type of subject that should be explainable, we have defined at least one discourse strategy which has a discourse goal corresponding to the subject. When the explanation request is made, the request’s topic is interpreted as an instantiated discourse goal. A
strategy with the same discourse goal is picked, and if all conditions evaluate to true, the operator is used to generate the deep explanation. Evaluating the conditions, the predicates in the precondition are first checked. They refer either directly to elements of the source model or to rules analyzing properties of the source model. In any case, the evaluation of the precondition depends on the source model and — if found to be satisfiable — binds the variables of discourse goal and subgoals. The feature condition, on the other hand, specifies in what views the strategy is visible, and if this condition is unifiable with the explanation view, the operator can be applied in the generation. If there are several strategies to choose among, or several possible formulations of the same strategy, they are tried in sequence until one succeeds. In Table 9.3, three strategies are shown, of which the dynamics strategy has three formulations ((c)-(e)).

When a strategy is chosen, it's rhetorical relation (if not nil) is put on top of an empty explanation structure and the generation proceeds by expanding its subgoals. Each of the two subgoal classes contains a logical expression of simple subgoals specifying what combinations of them can be used to attain the overall discourse goal. Possible combinations are all of a group, one of a group, or as many as possible of a group, where the elements of the group are either simple subgoals or new combinations of subgoals. A simple subgoal is a new discourse goal, a reference to a source model element, or a call to a view generation routine, each requiring a separate generation strategy as explained below.

When discourse subgoals are expanded they will first inherit the explanation view, the user knowledge strategy, and the linguistic features from the instantiated discourse strategy. A new strategy corresponding to the subgoal is then searched for, and provided that a strategy is found and its conditions are satisfiable, its rhetorical relation is put into the explanation structure as a subordinate to the previously inserted relation, and the planning process continues with the instantiated subgoals of that strategy.

For subgoals referring to elements of the source model, the requirements for including the element is that it be visible in the explanation view and that user's knowledge of it be unifiable with the user knowledge strategy. For example, if the user knowledge strategy says that only unknown elements can be included, the element must be marked as unknown or must simply lack a user knowledge marking. When a source model element meeting the requirements is found, it inherits the linguistic features from the discourse strategy and is put into the explanation structure as a subordinate to the rhetorical relation just inserted. That particular subgoal has then been attained.

Subgoals referring to view generation routines are fulfilled by inclusion and are just added to the explanation structure as subordinates to the rhetorical relation.

To improve the quality of the generation, three additional generation features have been introduced.

- A discourse strategy cannot be used the subgoals turn out to be impossible to attain, for example because some source model element is missing. The system will then backtrack and try other strategies and content elements, and this backtracking process can continue all the way back to the request.
<table>
<thead>
<tr>
<th>DISCOURSE STRATEGY structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>discourse goal: structures of E</td>
</tr>
<tr>
<td>rhetorical relation: dynamics</td>
</tr>
<tr>
<td>nucleus: properties of E</td>
</tr>
<tr>
<td>satellite: dynamics of E</td>
</tr>
<tr>
<td>precondition: E is dynamic_concept</td>
</tr>
<tr>
<td>feature condition: <code>&lt;behavioral view = yes, computational view = yes&gt;</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DISCOURSE STRATEGY properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>discourse goal: properties of E</td>
</tr>
<tr>
<td>rhetorical relation: interpretation</td>
</tr>
<tr>
<td>nucleus: declaration of concept E</td>
</tr>
<tr>
<td>satellite: denotation of E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DISCOURSE STRATEGY dynamics</th>
</tr>
</thead>
<tbody>
<tr>
<td>discourse goal: dynamics of E</td>
</tr>
<tr>
<td>rhetorical relation: condition</td>
</tr>
<tr>
<td>nucleus: dynamics of E</td>
</tr>
<tr>
<td>satellite: all triggers of E</td>
</tr>
<tr>
<td>feature condition: behavioral view = yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DISCOURSE STRATEGY dynamics</th>
</tr>
</thead>
<tbody>
<tr>
<td>discourse goal: dynamics of E</td>
</tr>
<tr>
<td>rhetorical relation: enablement</td>
</tr>
<tr>
<td>nucleus: dynamics of E</td>
</tr>
</tbody>
</table>
| satellite: \{ all flows received by E: \(d_{\text{modality}} = \text{perm}\) \}
\{ all preconditions for E \} |
| feature condition: computational view = yes |

<table>
<thead>
<tr>
<th>DISCOURSE STRATEGY dynamics</th>
</tr>
</thead>
<tbody>
<tr>
<td>discourse goal: dynamics of E</td>
</tr>
<tr>
<td>rhetorical relation: nil</td>
</tr>
</tbody>
</table>
| nucleus: \{ all flows generated by E: \(d_{\text{modality}} = \text{perm}\) \}
\{ all postconditions for E \} |
| feature condition: computational view = yes |

Table 9.3: Informal discourse strategies. `<list of goals>` means that only one of the goals should be used, `{list of goals}` that as many as possible should be used.
9.3. Deep Generation Process

- Sometimes, a discourse strategy will change the explanation view, the user knowledge strategy, or the linguistic features that it has inherited. For example, if a result given to the user should be explained, the request could specify that the explanation should consist of elements that are unknown to the user. The discourse strategy used, however, could modify the strategy, saying that those parts of the explanation that elaborate on the result itself may be known to the user, whereas the underlying computation should be explained in terms of unknown elements. Similarly, the satellite in strategy in Table 9.3(d) specifies that the linguistic feature \( d_{\text{modality}} = \text{perm} \) is to be used when the conceptual model element corresponding to "all flows received by \( E \)" is realized in natural language.

- Throughout the whole generation process, included instantiated discourse strategies and source model elements are recorded in an explanation log. The log is consulted every time a new strategy instantiation or source model element is under consideration, so that already included elements will not be put into the explanation structure a second time.

The explanation structure built during the generation process is the deep explanation. It is a hierarchical structure, of which the non-leaf nodes are rhetorical relations from the discourse strategies and the leaf nodes are calls to view generation routines or elements from the source model extended with linguistic features.

Using the informal request in Table 9.1 and the discourse strategies in Table 9.3 as examples, we can now go through the generation of a deep explanation. First, the structures strategy is chosen and instantiated to equal the request topic. The precondition evaluates to true since \( P3 \) is a dynamic concept, and the process continues by checking that the feature condition is satisfiable. The strategy can at least be used in the computational and the behavioral view, and since the computational view is included in the initial explanation view, the strategy's rhetorical relation is put into the explanation structure and we start expanding its subgoals. The nucleus subgoal is simply attained by using the condition-less operator properties, whereas the satellite is a bit more complicated. The satellite is specified as dynamics of \( E \), which means that any of the three formulations of dynamics in Table 9.3 may be used. The first one, though, is rejected, since it can only be used in the behavioral view. The second one is then tried, and this time the feature condition is compatible with what was specified in the request. Its rhetorical relation, enablement, is put into the hierarchical explanation structure, and its subgoals are expanded. The nucleus is dynamics of \( E \), and since the first formulation of dynamics is unavailable and the explanation log says that the second one is already included, the third formulation is now instantiated to attain the subgoal. The satellite is \{ all flows received by \( E \): \( d_{\text{modality}} = \text{perm} \), all preconditions for \( E \) \}, which means that zero, one, or both of these simple subgoals have to be fulfilled (preferably, as many as possible). There is no precondition for \( P3 \) in the conceptual model, however, so only all flows received by \( E \) is selected from the model and included as content elements in the explanation structure. \( D_{\text{modality}} = \text{perm} \) is added to the conceptual model elements to help in the natural language realization. When the third formulation of dynamics is used, no rhetorical relation is put into the explanation structure (since it is set to nil), but the information about \( P3 \)'s output flows is included. The resulting explanation structure, which is called the deep explanation, is depicted in Figure 9.3.
In the deep explanation, the features of content clauses govern the morphological and syntactical realization of them. Topic tends to mark the element realized as subject in the final sentence, whereas tense indicates the proper form of the finite verb. There are of course other linguistic features too, for example d_modality, and these we will come back to in Chapter 10. The text-structuring rhetorical elements building the hierarchical structure help in the coordination of sentences or sentence parts to form coherent paragraphs of text.

A surface generator could realize the deep explanation in Figure 9.3 as

"P3 is a process that processes loan requests. Dynamically, it can receive Requested loan, Customer data, and New customer data, and can generate Recommendation."

The element P3 can generate Requested loan, Customer data, and New customer data, could be included since the request's user knowledge strategy did not exclude unknown elements and the element was visible in the computational view. If the restrictive view were used instead of the computational one, another element must have been included instead of this one.

### 9.4 Generality of Component

The architecture outlined above and the set of discourse strategies and list of views can be reused for several conceptual modeling languages. When a new modeling language is to be used, new explanations can be generated as soon as an explanation meta model has been constructed and the proper translation rules are working. Since the explanation meta model is a specialization of the predefined domain structure model and does not include any knowledge of explanation generation, familiarity with the representations and generation process of the strategic component core is not necessary for the introduction of new languages.
9.4. Generality of Component

<table>
<thead>
<tr>
<th>types of DSM element</th>
<th>DSM elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>generic concepts</td>
<td>modeling_concept</td>
</tr>
<tr>
<td></td>
<td>flow</td>
</tr>
<tr>
<td></td>
<td>dynamic_concept</td>
</tr>
<tr>
<td></td>
<td>text</td>
</tr>
<tr>
<td>generic relations</td>
<td>generate(dynamic_concept,flow)</td>
</tr>
<tr>
<td></td>
<td>receive(dynamic_concept,flow)</td>
</tr>
<tr>
<td></td>
<td>denotation(text,concept)</td>
</tr>
</tbody>
</table>

Table 9.4: Some elements of the domain structure model.

This core generality is possible because the domain structure model form an interface between extended source model and discourse strategies. To explain the way the representations are connected, we need to go into some more detail on

- the content of the domain structure model,
- the way source model elements refer to elements of the domain structure model, and
- the way plan operators are defined in terms of domain structure elements.

The domain structure model contains two kinds of information, generic concepts and generic relations between terms. In Table 9.4, examples of both kinds are shown, and in Figure 9.4 the relationships between source model, domain structure model and discourse strategies are illustrated. The domain structure model, DSM for short, is supposed to capture structural concepts used in today’s graphical conceptual modeling languages. The concepts and relations included are generalizations of the structures found in these languages, and the whole model may be considered a generalized meta model.

The explanation meta model is constructed by selecting and specializing concepts from DSM. Associated with each such specialization are structural information (like “processes generate data flows”) and constraint information (like “every process generates at least one data flow”). To maintain the relationship between meta model and DSM, every meta model element is declared as the structure of, or the constraint on, some DSM element specialization. Additionally, since conceptual models, traces and model instances are formed by instantiating meta model elements, we declare these as instances of specializations of DSM elements. All the elements of the source model, then, are related to DSM, and the relationships will be referred to as the source model element’s reference. These specify the identifiers of the elements as well as the related DSM elements, and whether the element is a structure or a constraint. For example, both process and P3 will be referred to as structures related to the DSM element dynamic_concept.

The discourse strategies specify how specializations and/or instances of DSM elements can be combined to explain some model-related phenomenon. They are defined in terms of uninstantiated references — that is, a reference with an unbounded source model element
that can be instantiated to be used in a deep generation process. To take an example, the nucleus of the properties strategy (see Figure 9.3) is an uninstantiated reference saying that E is a structure related to the general DSM element concept.

The technique of using a domain structure model and references to implement discourse strategies has made the strategies independent of both model level and modeling language.

**Level independence** The same discourse strategies may be applied to explain properties of explanation meta model, conceptual model, trace, and model instances. As an example, the strategy structures in Table 9.3 can also be used to describe the dynamic properties of the process concept in general as well as of a specific execution of process P3.

**Language independence** The discourse strategies do not refer directly to any element of a particular modeling language. As long as a language is definable by specializing elements from DSM, though, the strategies will work on models of that language as if they were actually formulated in terms of it.

Of course, the approach rests on two crucial assumptions. First, the explanation meta model must be definable in terms of specializations of DSM elements. If DSM does not possess the necessary generality, parts of the explanation meta model would be lost and the quality of the explanations would suffer. Secondly, it is assumed that the granularity of DSM is appropriate for the formulation of discourse strategies. All specializations of a specific DSM element must be explainable from the same set of strategies, since no additional strategies at the level of the specializations are formulated.
Chapter 10

A Formalism for Deep Generation

The principles from the previous chapter are now ready to be formalized. In this chapter, we define a simple language for representing the information relevant to explanation generation, and introduce an algorithm for generating deep explanations.

10.1 An Explanation Modeling Language EML

The explanation modeling language EML is the only representational language used in the strategic component. It is a linguistically motivated language, though it represents deep explanations, explanation requests, discourse strategies, explanation meta models, explanation conceptual models, conceptual model instances, explanation execution traces, and user models. The language itself is more like a structuring mechanism than a traditional representation language, and at the lower levels it can include and relate information represented in other more specialized languages.

Linguistic Foundation

The structures of EML are modifications and extensions of the functional structures found in feature-based grammars like Lexical Functional Grammar [28] (also called feature structures or attribute-value structures [193, 218]). These functional structures, called f-structures for short, encode linguistic information and has been used both to parse and generate natural language. In Weigand [246] the same structures are applied in the formalization of Functional Grammar [67], which has been used as a linguistically oriented knowledge representation language.
Generally speaking, f-structures are simply sets of attribute-value pairs. The values in an f-structure, written on the right-hand side of the structure, may be expressions or other f-structures, of which the expressions are usually symbols or semantic forms. Formally, f-structures represent functions, from attributes to values. In Table 10.1(a) the general structure of f-structures is depicted. As Table 10.1(b) shows, the values can be f-structures themselves, forming a highly recursive representation language. In the table, \( v_1 \) is the value of attribute \( a_1 \). Attribute \( a_2 \)'s value is a new f-structure that contains the attribute-value pairs \([a_{21} \ v_{21}]\) and \([a_{22} \ v_{22}]\).

In EML we have used the ideas of f-structures at a general knowledge representation level. We have increased the expressive power of the structures, but retained most of their original properties. Below, the basic syntactic and semantic properties of EML are described.

**Basic syntax**

The basic structuring unit in EML is the EML structure. It includes a reference, which forms a simple interface to the structure, and an AND structure specifying the unit's content. In an EML structure, we distinguish between structure elements and expression elements. A structure element is either a simple attribute value pair or a combination of other structure elements, while an expression element is a simple expression, a new EML structure, or a combination of expression elements. In both cases, the combinations allowed are possibly nested lists of AND-ed, OR-ed, or XOR-ed elements or the same kind.

With respect to the grammar in Table 10.2, we can briefly describe the basic syntactic features of EML structures. The EML structure is written as `reference:AND.structure`, of which `reference` is a predicate terminal. The AND structure is a list of structure elements, and each of these elements may be another AND structure, an OR structure, and XOR structure, or an attribute value pair. An attribute value pair is written as 'attribute value', where attribute is a terminal and value is an AND structure, an OR structure, an XOR structure, or an expression element. At last, the expression element denotes a simple expression, an EML structure, an AND expression, an OR expression, or an XOR expression. In Table 10.2, \( \alpha \) marks that \( \alpha \) may be repeated zero or more times, while \( (\alpha|\beta) \) means that either \( \alpha \) or \( \beta \) is included. Terminals are written in bold face and are underlined.

The terminals used in the grammar are as follows:
### 10.1. An Explanation Modeling Language EML

#### basic syntax of EML

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EML_structure</code></td>
<td><code>reference : AND_structure</code></td>
</tr>
<tr>
<td><code>AND_structure</code></td>
<td>`{structure_element</td>
</tr>
<tr>
<td><code>OR_structure</code></td>
<td>`{structure_element</td>
</tr>
<tr>
<td><code>XOR_structure</code></td>
<td>`{structure_element</td>
</tr>
<tr>
<td><code>structure_element</code></td>
<td>`(AND_structure</td>
</tr>
<tr>
<td><code>att.value.pair</code></td>
<td><code>attribute value</code></td>
</tr>
<tr>
<td><code>value</code></td>
<td>`(AND_structure</td>
</tr>
<tr>
<td><code>AND_expression</code></td>
<td>`{expression_element</td>
</tr>
<tr>
<td><code>OR_expression</code></td>
<td>`{expression_element</td>
</tr>
<tr>
<td><code>XOR_expression</code></td>
<td>`{expression_element</td>
</tr>
<tr>
<td><code>expression_element</code></td>
<td>`(AND_expression</td>
</tr>
</tbody>
</table>

Table 10.2: A grammar for EML.

- **reference** is simply a predicate describing or classifying the content of the EML structure. A reference does not need to be unique.

- **attribute** is taken from a predefined set of attribute names. It describes the nature of its value, in that it characterizes the value's role in the structure.

- **simple_expression** can be a well-formed predicate formula, an arithmetic calculation, a symbol, or a representation in another language.

Using EML we can represent that process P3 can receive the flow Requested Loan as follows:

```
receive(p3,requested_loan) : [P_OPERATORS [TENSE pres, D_MODALITY perm], HEAD receive, AGENS [T_OPERATORS [DEF d], HEAD p3], GOAL [T_OPERATORS [DEF d], HEAD requested_loan]]
```

This notation is not very perspicuous, so we will rather use the tabular structure shown in Table 10.3. The structure itself is quite simple and only involves the nesting of AND structures. For example, the attribute P_OPERATORS has a structural value which is an AND structure of attribute value pairs, and hence all these pairs are required to apply. In Table 10.4, however, which formalizes two discourse strategies introduced in Chapter 9 (see Table 9.3), both OR structures and XOR structures are needed. The views of the structures strategy is an XOR structure of two required views, which means that the strategy can be used if at least one of them are available. In the dynamics strategy, the satellite is
Table 10.3: EML structure for “P3 can receive Requested_loan”.

Table 10.4: Discourse strategies structures and dynamics represented in EML (the strategies will be made more complicated in Chapter 12).

an OR structure of two subgoals. The OR structure states that both, one, or none of the subgoals must be attained (preferably, as many as possible).

Going into the details of EML structures, we will adopt the following conventions for variables:

\[ \alpha : \text{an attribute} \]
\[ \beta : \text{a value} \]
\[ \Sigma : \text{a structure (AND, OR, or XOR)} \]
\[ \gamma : \text{a simple expression} \]
\[ \Gamma : \text{an EML structure} \]
\[ \rho : \text{a reference} \]

For example, the generic structure \( \Gamma_1 = \rho_1 : [\alpha_1 \beta_1] \) denotes a simple EML structure containing one single attribute value pair.
### 10.1. An Explanation Modeling Language EML

#### Table 10.5: A generic EML structure and its interpretation.

<table>
<thead>
<tr>
<th>EML representation</th>
<th>semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rho_i$</td>
<td>$\rho_i$ specifies a relation $\mathcal{R}$ such that</td>
</tr>
<tr>
<td>$\alpha_1 \gamma_1$</td>
<td>1. $\alpha_1 \mathcal{R} \gamma_1$ applies;</td>
</tr>
<tr>
<td>$\alpha_2 \begin{bmatrix} \gamma_2^1 \ \gamma_2^2 \end{bmatrix}$</td>
<td>2. $\alpha_2 \mathcal{R} \gamma_2^1$ and $\alpha_2 \mathcal{R} \gamma_2^2$ apply;</td>
</tr>
<tr>
<td>$\alpha_3 \begin{bmatrix} \gamma_3^1 \ \gamma_3^2 \end{bmatrix}$</td>
<td>3. $\alpha_3 \mathcal{R} \gamma_3^1$ or $\alpha_3 \mathcal{R} \gamma_3^2$ or none of them applies;</td>
</tr>
<tr>
<td>$\alpha_4 \begin{bmatrix} \gamma_4^1 \ \gamma_4^2 \end{bmatrix}$</td>
<td>4. $\alpha_4 \mathcal{R} \gamma_4^1$ or $\alpha_4 \mathcal{R} \gamma_4^2$, but not both of them, applies;</td>
</tr>
<tr>
<td>$\alpha_5 \begin{bmatrix} \alpha_5^1 \gamma_5^1 \ \alpha_5^2 \gamma_5^2 \end{bmatrix}$</td>
<td>5. $\alpha_5^1 \mathcal{R} \gamma_5^1$ and $\alpha_5^2 \mathcal{R} \gamma_5^2$ apply;</td>
</tr>
<tr>
<td>$\alpha_6 \begin{bmatrix} \alpha_6^1 \gamma_6^1 \ \alpha_6^2 \gamma_6^2 \end{bmatrix}$</td>
<td>6. $\alpha_6^1 \mathcal{R} \gamma_6^1$ or $\alpha_6^2 \mathcal{R} \gamma_6^2$ or none of them applies;</td>
</tr>
<tr>
<td>$\alpha_7 \begin{bmatrix} \alpha_7^1 \gamma_7^1 \ \alpha_7^2 \gamma_7^2 \end{bmatrix}$</td>
<td>7. $\alpha_7^1 \mathcal{R} \gamma_7^1$ or $\alpha_7^2 \mathcal{R} \gamma_7^2$, but not both of them, applies;</td>
</tr>
<tr>
<td>$\alpha_8 \begin{bmatrix} \alpha_8^1 \ \alpha_8^2 \end{bmatrix} \gamma_8^1$</td>
<td>8. $\alpha_8^1 \mathcal{R} \gamma_8^1$ and $\alpha_8^2 \mathcal{R} \gamma_8^2$ apply;</td>
</tr>
<tr>
<td>$\alpha_9 \begin{bmatrix} \alpha_9^1 \ \alpha_9^2 \end{bmatrix} \gamma_9^1$</td>
<td>9. $\alpha_9^1 \mathcal{R} \gamma_9^1$ or $\alpha_9^2 \mathcal{R} \gamma_9^2$ or none of them applies;</td>
</tr>
<tr>
<td>$\alpha_10 \begin{bmatrix} \alpha_{10}^1 \ \alpha_{10}^2 \end{bmatrix} \gamma_{10}^1$</td>
<td>and</td>
</tr>
<tr>
<td>$\alpha_{10} \begin{bmatrix} \alpha_{10}^1 \gamma_{10}^1 \ \alpha_{10}^2 \gamma_{10}^2 \end{bmatrix}$</td>
<td>10. $\alpha_{10}^1 \mathcal{R} \gamma_{10}^1$ or $\alpha_{10}^2 \mathcal{R} \gamma_{10}^2$, but not both of them, applies;</td>
</tr>
</tbody>
</table>

### Basic semantics

An EML structure specifies the relationships between attributes and values. We will here just indicate the semantics of these relationships, since their interpretations should be quite obvious. More formal discussions of this topic are found in for example [2, 127, 218].

In Table 10.4, dynamics is the value of an attribute called HEAD in the structures strategy. The structure specifies a relationship, let us call it $\mathcal{R}_2$, between the attribute and the value, which we may write $\mathcal{R}_2(\text{HEAD, dynamics})$ or $\text{HEAD} \mathcal{R}_2 \text{dynamics}$. $\mathcal{R}_2$ is a relation and not a function since — as the value of strategy dynamics's SATELLITE illustrates — several possible values can be stated for the same attribute. In our terminology, we will say that in the EML structure for the structures strategy in Table 10.4, the relation HEAD $\mathcal{R}_2$ dynamics applies.

Not all the specified attribute value pairs of an EML structure has to apply. As the value of dynamics's SATELLITE in Table 10.4 is specified, for example, there are four alternative interpretations:
(1) \text{SATELLITE}_R \forall \text{receive}(E,:)[\text{P.OPERATORS [D.MODALITY perm]}] \text{ and } \text{SATELLITE}_R \forall \text{precondition}(E,:) \text{ applies.}

(2) only \text{SATELLITE}_R \forall \text{receive}(E,:)[\text{P.OPERATORS [D.MODALITY perm]}] \text{ applies.}

(3) only \text{SATELLITE}_R \forall \text{precondition}(E,:) \text{ applies.}

(4) none of them applies.

To decide on an interpretation, an additional strategy of making as many relations as possible to apply is adopted by the system.

When structures are nested, the relationships between attributes and values get more complex. In the value of \text{P.OPERATORS} in Table 10.3, for example, two relationships are required to apply, of which the first of them is that the value of \text{P.OPERATORS}'s TENSE is pres. Notationally, we will write this as \text{<P.OPERATORS,TENSE>} \text{ R}_i \text{pres}. In Table 10.5, we have indicated the interpretations of the various structures in \text{EML}. \text{ R} is the relation specified by the content of \text{EML} structure \rho_i.

Terminology

Consider the \text{EML} structure in Table 10.3. The \text{EML} structure has four attributes (\text{HEAD}, \text{P.OPERATORS}, \text{AGENS}, and \text{GOAL}), and we will refer to them as the \text{structure's attributes} or the \text{attributes of the structure}. The four corresponding values are called the \text{structure's attribute values} or just the \text{values of the attributes}. Receive, for instance, is the value of \text{HEAD} and an attribute value of the whole \text{EML} structure. Attribute value pairs may be grouped together as \text{AND} structures, \text{OR} structures, or \text{XOR} structures, as shown in Table 10.5. The \text{OR} structure in Table 10.5, which is the one made up of attributes \alpha^1_9 and \alpha^2_9, will be called an \text{OR part structure} of the \text{EML} structure \rho_i, whereas the structure including \alpha^1_2 and \alpha^2_2 and the structure including \alpha^1_{10} and \alpha^2_{10} are called \text{AND part structure} and \text{XOR part structure}, respectively.

We also need a terminology for describing hierarchical relationships among attributes and structures. As mentioned before, an attribute value may be an expression or a new \text{EML} structure. With reference to Table 10.3, we will call \text{DEF} an \text{immediate subattribute} of \text{T.OPERATORS} and a \text{distant subattribute} of \text{AGENS}, whereas \text{AGENS} is an \text{immediate superattribute} of \text{T.OPERATORS} and a \text{distant superattribute} of \text{DEF}. \text{AGENS} and \text{GOAL} are referred to as \text{sister attributes}, and the set of all such sister attributes is referred to as a \text{sister attribute set}. Correspondingly, the \text{EML} structure is the \text{immediate superstructure} of \text{AGENS}'s structural value and a \text{distant superstructure} of \text{P.OPERATORS'} value. The interpretation of \text{immediate substructures} and \text{distant substructures} should be obvious.

In addition, we need to define the \text{EML-resolution} of a an attribute value structure. If a structure is an \text{XOR} structure or an \text{OR} structure — or contain substructures or part structures of these kinds — we can construct new structures, called \text{EML-resolutions}, that reflect the way these structures are handled. For each \text{XOR} structure and \text{OR} structure included in the original structure, we make a choice as to which of its elements are to apply (following
10.2 Predefined Attributes

Table 10.6: The five bottom structures are EML-resolutions of the upper structure.

The rules in Table 10.5. Including only these chosen elements in a new possibly nested AND structure, we get an EML-resolution of the original structure. Different choices lead to different EML-resolutions, but there will always be a definite number of them of a given structure. In Table 10.6, for example, the upper structure has five EML-resolutions, each of them the result of a particular interpretation of the original structure.

10.2 Predefined Attributes

The attributes used in this work are given special meanings due to their links to other representations and are briefly described in Table 10.7 and Table 10.8. They correspond to constructs or constructions of these other representations, and we have only defined them by describing these correspondences. In this sense, our list of attributes is not necessarily exhaustive, though it suffices for the representations currently used in the strategic component:

- **HEAD**, **NUCLEUS**, **SATELLITE**, and **PRECONDITION** are taken from existing discourse strategies (see Section 6.2).
- **USER** and **VIEWS** represent user models and explanation contexts.
- The rest of the predefined attributes correspond to operators, roles, and functions in Functional Grammar, and a more thorough explanation of them can be found in Appendix C or in Dik’s introduction to Functional Grammar [67].

There are certain hierarchical relationships between these attributes. In Table 10.3, for example, **P.OPERATORS** have two subattributes, **TENSE** and **D.MODALITY**, but it would not make any sense for **TENSE** to have **P.OPERATORS** as a subattribute. On the basis of an analysis of meaningful hierarchical patterns of attributes, we can restrict the set of allowable attribute structures by defining an attribute hierarchy. In the hierarchy in Figure 10.1,
Figure 10.1: The attribute hierarchy.

\( \alpha \rightarrow \beta \) means that \( \beta \) can be an immediate subattribute of \( \alpha \). If \( \alpha \rightarrow \beta \) and \( \alpha \rightarrow \gamma \), though, this does not necessarily mean that \( \beta \) and \( \gamma \) are sister attributes. It might be the case that the use of one of them excludes the use of the other. \( <EML\ structure> \) marks the top of the hierarchy and is not an attribute.

An attribute in an \( EML \) structure must be identifiable from the outside. This means that any attribute name must be unique within its sister attribute set, though the attribute values need not of course be determined or unique.

Two simplification rules, shown in Table 10.9, make \( EML \) structures simpler and easier to read. As indicated by the first rule, default values are implicit when information about a specific attribute is missing. Defaults for simple expressions are given in Table 10.7 and Table 10.8. When it comes to attributes with possibly structural values, like \( \text{P\_OPERATORS} \) and \( \text{T\_OPERATORS} \), the default values are always the empty structure [] . The latter rule allows the attribute name \( \text{HEAD} \) to be deleted without any loss of information. Both rules are illustrated in Table 10.10, where the structural value of \( \text{GOAL} \) is gradually simplified to just \text{process}.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEAD</td>
<td>In structures representing source model elements, HEAD represents clause predicates or term heads. When discourse strategies are represented, HEAD specifies the rhetorical relation describing the textual relationship between NUCLEUS and SATELLITE. If the value of HEAD must be unique, they can be given an index as a subscript (e.g. offer\textsubscript{recommendation}).</td>
</tr>
<tr>
<td>NUCLEUS &amp; SATELLITE</td>
<td>These two attributes denote the subgoals of discourse strategies. In some rare cases, the values of these attributes may be a whole new EML structure, but usually the value is just a reference.</td>
</tr>
<tr>
<td>PRECONDITION</td>
<td>This attribute is used in formalizations of discourse strategies, and it contains predicates that affect when and how the strategy can be used. The predicates check properties of the source model and can also bind free variables in other parts of the strategy.</td>
</tr>
<tr>
<td>SALIENCE</td>
<td>This attribute, with subattributes TOPIC, FOCUS, and SUPPRESSED, specifies salience values of a predicate’s terms.</td>
</tr>
<tr>
<td>TOPIC</td>
<td>The attribute’s value specifies the pragmatic topic of the clause.</td>
</tr>
<tr>
<td>FOCUS</td>
<td>The attribute’s value is the pragmatic focus of a clause.</td>
</tr>
<tr>
<td>SUPPRESSED</td>
<td>Using this attribute, one may specify that an argument to a clause should be suppressed when the clause is presented in natural language. The value is the argument to be suppressed.</td>
</tr>
<tr>
<td>P_OPERATORS</td>
<td>The subattributes of this attribute correspond to operators at the predicate level in Functional Grammar and is used for the representation of source model elements.</td>
</tr>
<tr>
<td>POLARITY</td>
<td>The value is either pos (for positive) or neg (for negative), of which the default value is pos.</td>
</tr>
<tr>
<td>PERFECTIVITY</td>
<td>This attribute specifies the perfectivity aspect of a clause. Values are impf (for imperfective) and pf (for perfective).</td>
</tr>
<tr>
<td>INT_PHASE</td>
<td>The attribute keeps the internal phasal aspect of a clause, as defined in FG. Values are ingr (for ingressive), progr (for progressive), and egr (for egressive).</td>
</tr>
<tr>
<td>TENSE</td>
<td>This is the tense of a clause, and the values are past, pres (for present), and fut (for future).</td>
</tr>
<tr>
<td>EXT_PHASE</td>
<td>In FG, the external phasal aspect of a clause is specified as prosp (for prospective) or perf (for perfect).</td>
</tr>
<tr>
<td>E_MODALITY</td>
<td>This attribute is used for the epistemological objective modality of a clause, as defined in FG. A wide range of values could be imagined, but for our purposes cert (for certain) and pos (for possible) are sufficient.</td>
</tr>
<tr>
<td>D_MODALITY</td>
<td>The deontic objective modality of a clause, as used in FG, is specified here. The values needed in our system are obl (for obligatory) and perm (for permissible).</td>
</tr>
<tr>
<td>Restrictors</td>
<td>A restrictor is a lemmat that modifies a clause or a term. In FG terminology, it is a satellite that is not realized as an argument, but rather as adjectives, adverbs, and so on.</td>
</tr>
</tbody>
</table>

Table 10.7: Attributes used in EML.
<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;SEM. ROLE&gt;</td>
<td>The names of these attributes are taken from the semantic roles defined in FG, e.g. AGENS, GOAL and INSTR (see Appendix C). The values correspond to obligatory or optional arguments of a clause, and can normally involve the subattributes ROLE, T.OPERATORS, HEAD, POSSESSOR, RESTRICTORS, and &lt;SEM. ROLE&gt; (which is a restriction given a semantic role). However, as a whole clause may serve as a semantic role in another clause, the value may also be a new EML structure.</td>
</tr>
<tr>
<td>ROLES</td>
<td>Subattributes of ROLES are SYNTAX and PRAGMATICS.</td>
</tr>
<tr>
<td>SYNTAX &amp; PRAGMATICS</td>
<td>These two immediate subattributes of ROLES are used to give the constituent syntactic and pragmatic roles. Possible values of SYNTAX are subj (for subject) and obj (for object), and for PRAGMATICS the values are top (for focus) and foc (for focus).</td>
</tr>
<tr>
<td>T.OPERATORS</td>
<td>This attribute specifies term operators associated with the arguments of source model elements.</td>
</tr>
<tr>
<td>DEF</td>
<td>The attribute specifies the definiteness of the constituent corresponding to a semantic role. Values are d (meaning definite), i (meaning indefinite), and * (meaning universal quantification, as in &quot;every man&quot; or &quot;all men&quot;). As a default, i is used.</td>
</tr>
<tr>
<td>NUM</td>
<td>The attribute specifies the quantity of the constituent corresponding to a semantic role. Values are specified as either [q] or [q₁,q₂], following the notation in [247]. q is a given quantity, and can be a number like 1 or a more vaguely specified number like many or all. q₁ and q₂ are the boundaries of a specified amount and can be a quantity like q or a character indicating indefiniteness in upwards or downwards direction. To take some examples, [1,+] means &quot;one or more&quot;, [-10] means &quot;ten or less&quot;, while [s] and [p] means &quot;singular&quot; and &quot;plural&quot;, respectively. A default value of [p] is assumed when nothing else is stated.</td>
</tr>
<tr>
<td>SET</td>
<td>This attribute is included when a quantity or a constituent is described as a part of some other quantity or constituent (e.g. &quot;two of four&quot;). SET will then represent the larger set (quantity or constituent), of which the specified one is a subset.</td>
</tr>
<tr>
<td>POSSESSOR</td>
<td>This attribute specifies the possessor of head, and the corresponding value may itself be a full argument involving the subattributes T.OPERATORS, HEAD, POSSESSOR, and RESTRICTORS.</td>
</tr>
<tr>
<td>USER</td>
<td>User's knowledge of source model elements is recorded using this attribute. If the value is yes, the user is assumed to be familiar with that particular element. No means that the element is unknown to the user.</td>
</tr>
<tr>
<td>VIEWS</td>
<td>The attribute lists the views in which the element corresponding to the EML representation is visible or hidden. Immediate subattributes are names of predefined views.</td>
</tr>
</tbody>
</table>

Table 10.8: Attributes used in EML.
10.3 Representing Explanation-Relevant Knowledge

Table 10.9: Simplification rules in EML.

\[
\begin{align*}
\text{GOAL} & \quad \text{T-OPERATORS} & \quad \text{DEF} & \quad i \\
\text{HEAD} & \quad \text{NUM} & \quad [p] \\
\text{} & \quad \text{process}
\end{align*}
\]

\[\uparrow\text{ simplification rule 1}\]

\[
\begin{align*}
\text{GOAL} & \quad \text{HEAD} & \quad \text{process}
\end{align*}
\]

\[\uparrow\text{ simplification rule 2}\]

\[
\begin{align*}
\text{GOAL} & \quad \text{process}
\end{align*}
\]

Table 10.10: Example showing how the simplification rules work.

### 10.3 Representing Explanation-Relevant Knowledge

EML may be viewed as a structural mechanism for manipulating structures from other languages. In a strategic component there are well-defined links between EML and all the original formalisms used to represent explanation-relevant knowledge. The detailed semantics of EML is deduced from these links and the semantics of the various formalisms. Since EML is only a way of structuring representations from other formalisms, its interpretation depends on how the links are established.

The source model represented in EML is not necessarily equal to the representations found in the rest of the environment. Only the explanation-relevant parts are included in this model, and since it is not manipulated or used by other components, its formal properties are also a bit relaxed. Verification checks and model executions, for example, are performed on other kinds of model representations and are only indirectly related to our
dynamic_concept(process) : \[
\begin{bmatrix}
\text{HEAD} & \text{dynamic_concept} \\
\text{ZERO} & \text{process}
\end{bmatrix}
\]

Table 10.11: Process is declared as a dynamic concept.

explanation conceptual model (see Section 8.2). However, a well-defined mapping from these other representations to the source model is assumed.

Contrary to most representations in the strategic component, the deep explanation itself is not transformed from another language. It is generated by the component, and there are no direct links between deep explanation elements and structures from another language. Still, since many of the basic structures of EML are taken from formalisms for representing linguistic information, it is possible to establish transformation from EML to some linguistically motivated language. The semantics of the natural language parts of EML deep explanations, thus, is given by transformations from EML to FG and the formal semantics of FG.

**Explanation Meta Model**

There are various types of elements in an explanation meta model, and these we will come back to in Chapter 11. Here, we will indicate the declaration of structure elements in the meta model using the example element in Table 10.11. A meta model element has a reference that classifies it according to a generalized meta model implicit in the system, and the element structure is a predication that describes the nature of the element. In Table 10.11, the generic concept process is declared as a specialization of dynamic_concept, which belong to this underlying generalized meta model.

The other types of meta model elements, constraint elements and model constraint elements, are used to specify syntactic constraints on the modeling language and to declare concepts for specifying constraints in the conceptual model, respectively.

**Explanation Conceptual Model**

The explanation conceptual model contains instances of the explanation meta model elements. References are on the same form as for the meta model, as illustrated in Figure 10.2. In the figure, an EML declaration corresponding to the creation of process P3 is given in (a), and in (b) the attachment of flow Requested_Joan to P3 is shown.

The EML structures do not intend to capture all the semantics of the original source models. As the aim of the explanation conceptual model is just to provide a conceptual description that is suitable for explanation generation, features of the original conceptual model that are easiest understood using the environment’s modeling language, are included directly.
10.3. Representing Explanation-Relevant Knowledge

\[ \text{P3} \rightarrow \text{Requested loan} \]

\[
\begin{align*}
\text{dynamic\_concept(p3)} & : \\
\begin{bmatrix}
\text{HEAD} & \text{process} \\
\text{ZERO} & \text{p3}
\end{bmatrix} & \\
\text{receive(p3,requested\_loan)} & : \\
\begin{bmatrix}
\text{HEAD} & \text{receive} \\
\text{AGENS} & \text{p3} \\
\text{GOAL} & \text{requested\_loan}
\end{bmatrix}
\end{align*}
\]

(a) \hspace{5cm} (b)

Figure 10.2: (a) Declaration of process P3. (b) Process P3 can receive the flow Requested\_loan.

postcondition(r321.formula(offer_{recommendation = maximum_{loan\_limit}})):

\[
\begin{bmatrix}
\text{HEAD} & \text{result} \\
\text{AGENS} & r3.2.1 \\
\text{GOAL} & \text{formula(offer}_{recommendation = maximum_{loan\_limit}})
\end{bmatrix}
\]

Table 10.12: Including formal language expressions in EML using formula(...).

using the structure formula(...). In Table 10.12, the calculation of rule r3.2.1 is specified as the rule's postcondition. The formula itself is not converted to an EML structure, since it is assumed to be easier to understand in its original mathematical notation.

Traces and Model Instances

At the conceptual level, we assume traces and model instances to be just instances of elements from the explanation conceptual model. The EML representations are constructed just as the elements of the conceptual model and the meta model, which means that the source model as a whole is a collection of connected submodels at the meta model level, the conceptual model level, and the instance level.

For example, in Table 10.13 both a conceptual model element and a trace element in shown. In (a), offer_{recommendation} is declared as an item in the conceptual model, whereas offer_{recommendation} is given the value 80 in (b). The subscripts of the values are necessary for the elements to be uniquely identified. Similarly, the execution of statements and dynamic objects is viewed as an instantiation process, even though there are no values involved in these executions. To take an example here, executing process P3 could be
static_concept(offer\textsubscript{recommendation}): \[ \begin{bmatrix} \text{HEAD} & \text{item} \\ \text{ZERO} & \text{offer}_{\text{recommendation}} \end{bmatrix} \]

static_concept(80\textsubscript{i1}): \[ \begin{bmatrix} \text{HEAD} & \text{offer}_{\text{recommendation}} \\ \text{ZERO} & 80\textsubscript{i1} \end{bmatrix} \]

(a) \hspace{1cm} (b)

Table 10.13: (a) Offer\textsubscript{recommendation} declared as an instance of item. (b) 80\textsubscript{i1} declared as an instance of offer\textsubscript{recommendation}.

dynamic_concept(p3): \[ \begin{bmatrix} \text{HEAD} & \text{process} \\ \text{ZERO} & p3 \\ \text{VIEWS} & \begin{bmatrix} \text{COMPUTATIONAL} & \text{yes} \\ \text{RESTRICTIVE} & \text{no} \\ \text{USER} & \text{yes} \end{bmatrix} \end{bmatrix} \]

Table 10.14: Annotating source model elements with view information.

recorded as the creation of trace element P3\textsubscript{14}, which is of type P3.

**View Definitions**

A view is defined as a subset of the source model. Defining a view, one might list all the elements of the source model that belong to that particular view. Alternatively, though, one might annotate source model elements with the names of the views to which they belong, and this is the strategy chosen in our system.

In Table 10.14, we have stated that the conceptual model element is included in the COMPUTATIONAL view and hidden in the RESTRICTIVE views. If an element's visibility in a particular view is not indicated, the system is free to assume what suits it best.

Similarly, discourse strategies may be associated with a number of views, and the strategies can then only be used if one of these views applies.

**User model**

The user model is divided into two submodels:

- the *user characterization submodel*, which characterizes the user in terms of which elements of the source model are accessible to him, and
10.3. Representing Explanation-Relevant Knowledge

\[ v_i : \begin{bmatrix} \text{VIEWS} & \begin{bmatrix} \text{COMPUTATIONAL} & \text{yes} \\ \text{RESTRICTIVE} & \text{no} \end{bmatrix} \end{bmatrix} \]

Table 10.15: A user characterization submodel specifying the views associated with a user in some context.

- the user knowledge submodel, which specifies the parts of the source model that are known to the user.

User characterization submodels are represented as sets of views of the source model and may look like the one in Table 10.15. In this submodel, the user is assumed to need or be interested in those source model elements that are included in the computational view and to be uninterested in elements of the restrictive view. One could also imagine a view like end-user, which would decide what parts of the source model are relevant when generating explanations to end-users of the modeled system. In general, a user may be characterized by means of an arbitrary number of views, but all these views must be defined and specified in the extended source model.

User’s knowledge of source model elements are modeled as overlay models to the source models. Implementationally, an attribute called user is added to the elements of the source model, and its value indicate whether the elements are known to the user. In Table 10.14, the user is assumed to know that p3 is an instance of process. If the attribute value pair is missing, the system may make its own assumptions.

Our user model is assumed to be dynamic and individual. Both the user characterization submodel and the user knowledge submodel should be determined on the basis of user’s background and behavior and should be dynamically updated by the system. Though, stereotypical user models can also be defined, and these will keep typical properties of classes of users. When a new user is introduced, she could be assigned an appropriate stereotypical user model, and as the system records and analyzes her behavior, the user model is gradually modified to an individual model. However, since user modeling is a whole discipline in itself and depends on extensive analyses of user behavior and background, we will here just assume the availability of these user models.

Explanation Context

There are two ways of describing the explanation context:

- The explanation context may be characterized as a combination of view specifications, referred to as the context specification.

- During the execution of a conceptual model, the trace being built can serve as a continually changing explanation context. However, we consider this part to be
<strategy>:
[ HEAD : nil ]
[ NUCLEUS : <subgoal_expression> ]
[ PRECONDITION : <conditions_for_use> ]
[ VIEWS : <availability_in_views> ]

<strategy>:
[ HEAD : <rhetorical_relation> ]
[ NUCLEUS : <subgoal_expression> ]
[ SATELLITE : <subgoal_expression> ]
[ PRECONDITION : <conditions_for_use> ]
[ VIEWS : <availability_in_views> ]

Table 10.16: (a) General structure of schema-based discourse strategy. (b) General structure of operator-based strategy.

included in the source model, and we will not refer to the trace as an explanation context.

Views, thus, are used both to characterize the explanation context and the properties of users. A context specified in terms of views looks just like a user characterization sub-model, which was illustrated in Table 10.15.

Discourse Strategies

Both schema-based and operator-based discourse strategies can be defined in our EML formalism. The general structures of these discourse strategies are shown in Table 10.16, which also illustrates the main difference between the two types: When discourse schemas are constructed, there is no rhetorical relation that characterizes the relationship between its subgoals (HEAD is set to nil), and all subgoals are given the same status. For plan operators, a rhetorical relation must be added as head and the subgoals are classified as nucleus or satellite depending on their involvement in the rhetorical relation. Both strategies, however, are text-structuring.

Two operator-based discourse strategies are shown in Table 10.4. In the structures strategy, the rhetorical relation dynamics characterizes the textual relationship between the subgoals properties(\(E\)) and dynamics(\(E\)). PRECONDITION, which specifies a condition for using the strategy, requires that element \(E\) is a dynamic concept. Moreover, as seen from VIEWS, the strategy can only be used if the behavioral or the computational view can be assumed to apply.

The subgoals specified in NUCLEUS and SATELLITE are of three kinds: (1) Some subgoals may refer to new strategies, as shown in the structures strategy. (2) Some subgoals may refer to elements of the source model, and this is done in the satellite of dynamics in Table 10.4. (3) Some subgoals refer to view-displaying routines and are specified as display(\(E\)), which means that element \(E\) is to be displayed on the screen. To construct the
10.3. Representing Explanation-Relevant Knowledge

```
request: [HEAD nil]
        [SALIENCE [TOPIC p3]]
        [P_OPERATORS [TENSE pres]
          [structures(p3)]]
        [NUCLEUS]
        [VIEWS [COMPUTATIONAL yes]
          [EXTERNAL yes]
          [BEHAVIORAL no]
          [INTERNAL no]
          [ILLUSTRATIVE no]]
```

Table 10.17: Explanation request represented in EML.

actual call to the view generation component, all the deep explanation’s display references and their associated views are collected to form a call on the form

```
display(<list_of_elements>):<list_of_views>.
```

<list_of_elements> is the list of elements to be displayed, <list_of_views> the views characterizing the situations in which the references were included.

Two special operators, all and list, are defined to increase the expressive power of the strategies. All(Goal) means that the system should search for all solutions to Goal and return an AND structure with all these solutions. Provided that Var is an argument variable in Goal and Set is a list of possible values of Var, list(Set,Var,Goal) is interpreted such that Goal is attempted fulfilled for each element of Set used as Var in Goal, and the solution to the whole list goal is an AND structure of these part solutions.

Explanation Requests

The request includes a topic, a view specification (which represents the context and user’s preferences), a user knowledge strategy, and various linguistic features that are to govern the realization of content elements. In Table 10.17, the topic of the request is given in NUCLEUS as structures(p3), and linguistic features are specified as values of P_OPERATORS and SALIENCE. The view specification in VIEWS states that the computational and the external views apply and the behavioral view, the internal view and the illustrative view are hidden. No user knowledge strategy is indicated in this case. The request in Table 10.17 is a formalization of the request depicted in Table 9.1, except for the view specification which is a bit extended.
Deep Explanations

A deep explanation consists of content elements and hierarchical relations between explanation parts. The relations correspond to the ones used in RST and describe satellites’ rhetorical relationships to nuclei. Content elements are of three kinds: (1) references to view generation routines, (2) formulas given in some formal language, and (3) content clauses denoting clauses in natural language, which can all be translated to predications in FG using the algorithm indicated in Appendix D. Moreover, the deep explanations contain not only semantic information about their surface explanations. They also contain conceptual and pragmatic information that — although not truth-functional — is helpful when generating natural language sentences. Examples of this kind of information are relations like interpretation and enablement and the pragmatic roles topic and focus [139, 213].

Representationally, a deep explanation is a rather complex EML structure that includes instantiated discourse strategies and source model elements extended with additional linguistic information. In Table 10.18, a deep explanation describing the dynamics of process P3 is depicted. It can be given a realization as follows:

"P3 is a process that processes loan requests. Dynamically, it can receive Requested loan, Customer data, and New customer data, and can generate Recommendation."

Ignoring the coordination of clauses in the surface explanation above, we see that the satellite of enablement can be realized as "P3 can generate Recommendation". This corresponds to the FG message

\[
\text{DECL}(\text{PERM PRES e: (generate}_V \quad (d[1]x_1: p_{3\text{AgTop}}) \\
(d[1]x_2: \text{recommendation}_G\text{Foc}))).
\]

which can be produced using the algorithm in Appendix D.

10.4 Explanation Generation Algorithm

Unification and Inheritance of EML Structures

The process of unification can be defined on a strictly formal basis (see for example [78, 137, 219]). Here, we will not go into these details, but rather illustrate the process of unifying attribute value structures from a linguistic point of view. A more detailed treatment of structure unification can be found in for example [193, 214, 218].

An attribute value structure specifies some state of affairs. If one structure $\Sigma_1$ is an extension of another structure $\Sigma_2$, i.e. $\Sigma_1$ is at least as informative as $\Sigma_2$, we can write this as
Table 10.18: Example of a deep explanation.
\[ \text{P\_OPERATORS} \quad \text{TENSE} \quad \text{pres} \quad \text{HEAD} \quad \text{receive} \quad \text{AGENS} \quad p3 \quad \text{GOAL} \quad \text{recommendation} \]

\[ \cup \]

\[ \text{SALIENCE} \quad \text{TENSE} \quad \text{p3} \]

\[ \text{P\_OPERATORS} \quad \text{D\_MODALITY} \quad \text{perm} \]

\[ = \]

\[ \text{SALIENCE} \quad \text{TENSE} \quad \text{p3} \]

\[ \text{P\_OPERATORS} \quad \text{D\_MODALITY} \quad \text{perm} \]

\[ \text{HEAD} \quad \text{receive} \]

\[ \text{AGENS} \quad p3 \]

\[ \text{GOAL} \quad \text{recommendation} \]

Table 10.19: Unification of attribute value structures.

\( \Sigma_2 \subseteq \Sigma_1 \). \( \Sigma_2 \) is said to subsume \( \Sigma_1 \), since a less specified state of affairs naturally captures more phenomena than a detailed specification. An empty attribute value structure \( \top = [] \) embraces all real and imaginary state of affairs and will therefore subsume all other structures. The relation \( \subseteq \) is a reflexive partial ordering.

The unification of two attribute value structures is defined as a new structure that is at least as informative as either of the two others. To put in another way, the unification of structures \( \Sigma_1 \) and \( \Sigma_2 \), written

\[ \Sigma_1 \cup \Sigma_2, \]

is the most general attribute value structure \( \Sigma_{unified} \) such that \( \Sigma_1 \subseteq \Sigma_{unified} \) and \( \Sigma_2 \subseteq \Sigma_{unified} \). In Table 10.19, the unified structure is the greatest attribute value structure which is subsumed by both the other two structures. If two structures contain inconsistent information, e.g. like if the value of tense in the upper right structure in Figure 10.19 were past, the structures cannot be unified and we say that unification fails.

*Inheritance* between two structures means that a structure, the inheritor, receives those parts of the other structure that are unifiable with it. Notationally, if structure \( \Sigma_1 \) inherits \( \Sigma_2 \), the resulting structure \( \Sigma_{extended} \) is written as
\[ \Sigma_{\text{extended}} = \Sigma_1 \triangleright \Sigma_2 . \]

Inheritance is implemented as a modified unification strategy. Basically, \( \Sigma_1 \) and \( \Sigma_2 \) are attempted unified, but every time unification fails, \( \Sigma_1 \)'s properties are chosen and unification continues until all properties of \( \Sigma_1 \) and \( \Sigma_2 \) have been considered.

If the upper structure in Table 10.19 were to inherit the lower structure, the resulting structure would be equal to the unified one. Moreover, if tense of the lower structure were past instead of pres, the resulting structure would still be the same, since inconsistencies are solved by favoring the inheritor's properties. If the inheritor's attribute value is changed, however, this would imply a similar change of the resulting structure. Inheritance is an operation that never fails.

**Predicates for Generating Deep Explanations**

The generation of deep explanations starts with an explanation request. The request includes one or several communicative goals to achieve (attribute nucleus) and a number of constraints (attributes precondition, user, and views) that are to govern the generation process. The goals are attained by searching for elements in the extended source model or strategies in the discourse strategy pool that unify with the specification of the goals. If a new discourse strategy is used, its own subgoals have to be attained, and the generation process continues until there are no more subgoals to consider. The resulting structure, the *deep explanation*, is a hierarchical structure, in which non-leaf nodes are rhetorical relations and leaf-nodes are view generation calls or source model elements extended with linguistic information provided by the request or the discourse strategies used.

For a discourse strategy to be used to attain an explanation goal, its precondition must first be evaluated to true. Precondition contains a structure of predicates that inspects the source model and can return information back to the generation process.

For both discourse strategies and content elements, their inclusion in deep explanations to achieve goals is further governed by the attributes user and views. The elements must adhere to the user knowledge strategy and the available views indicated at the level of the explanation goal. For this, and for the propagation of linguistic information from request and discourse strategy to content elements, the processes of unification and inheritance are vital. Unification is used to match an explanation goal with an uninstantiated strategy from the discourse strategy pool or an element from the extended source model. Inheritance enables the transmittance of linguistic information (attributes salience and P.operators) and generation constraints (attributes user and views) from a discourse strategy's goal to its subgoals so that the same information can be used at the next level of the generation process. However, a discourse strategy — if included — can change some of the generation constraints by giving its subgoals a new user knowledge strategy and/or a new view specification. When these subgoals are to be attained, they inherit the constraints
of their strategy's goal before discourse strategy pool and source model are searched for matching structures. If the inheritance unifies with their own user and views specification, they will continue to influence the generation process, but if they are in conflict, they are replaced by the subgoals' own specifications. Similarly, linguistic information given in the explanation request is normally inherited by the content elements of the deep explanations. Some discourse strategies, though, can change this information such that the content elements subordinate to these strategies will inherit other kinds of linguistic information.

Central to the deep generation process is the concept of backtracking. Every time the generation process fails, it can reconsider earlier decisions and continue the generation in other directions. Specifically, backtracking can alter the results of evaluating predicates, searching in discourse strategy pool and source model, splitting structures in substructures, unifying structures, inheriting structures, and separating out EML-resolutions of structures. Inherent in text generation are all the difficult choices that have to be made throughout the process [167, 171], and backtracking has proven its value when choices are wrong and the process must roll back and reevaluate its decisions.

In Table 10.20 and Table 10.21, the informal predicates generate_deep_explanation and expand_subgoal show how the deep explanation process is realized. Σ is a structure (AND, OR, or XOR) variable, Γ is an EML structure variable or a reference variable, and ρ is a reference variable. The expression Σ(α₁, ..., αₙ) is a function that returns a structure that contains only the attribute value pairs indicated by α₁, ..., αₙ. Producing this new structure, one would sometimes need to use EML-resolution. The dot notation for Σ, Σ.α, returns the value of Σ's attribute α. Used on an EML structure Γ, it returns either the reference (Γ.reference) or the associated structure (Γ.structure). If Γ is a reference, Γ.reference = Γ and Γ.structure = [].

The actual implementation of the deep generation process in Prolog is a bit more complicated due to efficiency considerations (see [86]).

Generating a Deep Explanation

Consider the explanation request in Table 10.17. The response to request₁ is specified as an explanation describing the structures of element P3 (P3 is process P3 in Figure A.2 in Appendix A). The explanation's content is to be visible in the computational and the external view and hidden in the behavioral view, the internal view, and the illustrative view. External can be a view characterizing the explanation context, whereas computational and behavioral describe the user as more interested in data flows than control flows. No user knowledge strategy is indicated in the request.

The deep explanation Σ_{ret,0} is to be generated calling the predicate

\[
generate\_deep\_explanation(request₁, Σ_{ret,0}).
\]
Table 10.20: The predicate generate.deep.explanation for generating deep explanations.

The structure of request, is split such that

\[
\begin{align*}
\Sigma_{0, \text{precondition}} &= \emptyset, \\
\Sigma_{0, \text{inheritance}} &= \begin{bmatrix}
\text{P.OP\_OPERATORS} & \text{TENSE} & \text{pres} \\
\text{SALIENCE} & \text{TOPIC} & \text{p3} \\
\text{VIEWS} & & \\
\text{EXTERNAL} & \text{yes} \\
\text{COMPUTATIONAL} & \text{yes} \\
\text{BEHAVIORAL} & \text{no} \\
\text{INTERNAL} & \text{no} \\
\text{ILLUSTRATIVE} & \text{no}
\end{bmatrix}, \\
\Sigma_{0, \text{nucleus}} &= \text{NUCLEUS structures(p3)}, \\
\Sigma_{0, \text{satellite}} &= \emptyset, \text{ and} \\
\Sigma_{0, \text{head}} &= \text{HEAD nil}.
\end{align*}
\]

Since there is no precondition in the request and nucleus is the name of a discourse strategy.
expand_subgoal($\rho : \Sigma_{goal}, \Sigma_{explanation}$)

IF $\rho : \Sigma_{match}$ IN DISCOURSE STRATEGY POOL THEN
  IF $\rho : \Sigma_{match}$ NOT IN EXPLANATION LOG THEN
    IF $\Sigma_{unified} := \Sigma_{goal} \cup \Sigma_{match}$ SUCCEEDS THEN
      REGISTER $\rho : \Sigma_{match}$ IN EXPLANATION LOG
      generate_deep_explanation($\Sigma_{unified}, \Sigma_{explanation}$)
    ELSE
      BACKTRACK
  ELSE
    BACKTRACK
ELSE
  BACKTRACK
ELSE
  IF $\rho : \Sigma_{match}$ IN SOURCE MODEL THEN
    IF $\rho : \Sigma_{match}$ NOT IN EXPLANATION LOG THEN
      IF $\Sigma_{unified} := \Sigma_{goal} \cup \Sigma_{match}$ SUCCEEDS THEN
        REGISTER $\rho : \Sigma_{match}$ IN EXPLANATION LOG
        SPLIT $\Sigma_{unified}$ INTO $\Sigma_{out} := \Sigma_{unified}(user, views)$ AND
        $\Sigma_{explanation}$
      ELSE
        BACKTRACK
    ELSE
      BACKTRACK
  ELSE
    BACKTRACK
ELSE
  IF $\rho = \text{display}(E)$ THEN
    $\Sigma_{explanation} := \text{display}(E) : \Sigma_{goal}(views)$
  ELSE
    BACKTRACK
\end{verbatim}

Table 10.21: The predicate expand_subgoal for generating deep explanations.

the subgoal

$$\text{expand_subgoal(structures(p3):}\Sigma_{0,\text{inheritance}}, \Sigma_{ret,1})$$

is posted. The accompanying AND structure $\Sigma_{0,\text{inheritance}}$ is inherited from $\text{request}_t$, and makes sure that the necessary linguistic features are propagated to the explanation's content elements and that the views and user knowledge strategy are governing the whole generation process.

As seen from Table 10.4, the strategy structures is visible in the computational view. Its structure is unified with the subgoal's structure, the use of this formulation of the strategy is registered in the explanation log, and the generation proceeds with the call

$$\text{generate_deep_explanation(}\Sigma_1, \Sigma_{ret,2}),$$
where

\[
\Sigma_1 = \begin{bmatrix}
\text{HEAD} \\
\text{P\_OPERATORS} \\
\text{SALIENCE} \\
\text{NUCLEUS} \\
\text{SATELLITE} \\
\text{PRECONDITION} \\
\text{VIEWS}
\end{bmatrix}
\begin{bmatrix}
dynamics \\
\text{TENSE} \\
\text{pres} \\
\text{TOPIC} \\
p3 \\
\text{properties(p3)} \\
\text{dynamics(p3)} \\
\text{dynamic\_concept(p3)} \\
\text{EXTERNAL} \\
\text{yes} \\
\text{COMPUTATIONAL} \\
\text{yes} \\
\text{BEHAVIORAL} \\
\text{no} \\
\text{INTERNAL} \\
\text{no} \\
\text{ILLUSTRATIVE} \\
\text{no}
\end{bmatrix}
\]

The precondition of structures, \text{dynamic\_concept(p3)}, evaluates to true, which means that the strategy is applicable from the source model’s point of view. The structure to be inherited by nucleus and satellite is computed, and the discourse strategy in nucleus leads to the call

\[
\text{expand\_subgoal(properties(p3)\_}\Sigma_2, \Sigma_{ret,3}),
\]

where

\[
\Sigma_2 = \begin{bmatrix}
\text{P\_OPERATORS} \\
\text{SALIENCE} \\
\text{VIEWS}
\end{bmatrix}
\begin{bmatrix}
\text{TENSE} \\
\text{pres} \\
\text{TOPIC} \\
p3
\end{bmatrix}
\begin{bmatrix}
\text{EXTERNAL} \\
\text{yes} \\
\text{COMPUTATIONAL} \\
\text{yes} \\
\text{BEHAVIORAL} \\
\text{no} \\
\text{INTERNAL} \\
\text{no} \\
\text{ILLUSTRATIVE} \\
\text{no}
\end{bmatrix}
\]

Without elaborating more on this part, we conclude that the generation here will be successful, and the returned structure will be

\[
\Sigma_{ret,3} = \begin{bmatrix}
\text{HEAD} \\
\text{NUCLEUS} \\
\text{SATELLITE}
\end{bmatrix}
\begin{bmatrix}
\text{P\_OPERATORS} \\
\text{SALIENCE} \\
\text{HEAD} \\
\text{ZERO} \\
\text{P\_OPERATORS} \\
\text{SALIENCE} \\
\text{HEAD} \\
\text{AGENS} \\
\text{GOAL}
\end{bmatrix}
\begin{bmatrix}
\text{interpretation} \\
\text{TENSE} \\
\text{pres} \\
\text{TOPIC} \\
p3 \\
\text{ process} \\
p3 \\
\text{ process} \\
p3 \\
\text{ loan\_request}
\end{bmatrix}
\]
The satellite of structures leads to the call

\[
\text{expand}_{\text{subgoal}}(\text{dynamics}(p3);\Sigma_2,\Sigma_{\text{ret},4}).
\]

In Table 9.3, there are three formulations of the strategy dynamics. The one in (c) is tried first, but has to be rejected since it requires the \text{BEHAVIORAL} view to apply. The formulation in (d), which is formalized in Table 10.4, is then tried, and this time the unification with \text{dynamics}(p3):\Sigma_1 succeeds, the use of the strategy is registered, and a new call is made:

\[
\text{generate}_{\text{deep explanation}}(\Sigma_3,\Sigma_{\text{ret},5}),
\]

where

\[
\Sigma_3 = \begin{bmatrix}
\text{HEAD} & \text{enablement} \\
\text{P OPERATORS} & \text{TENSE} \quad \text{pres} \\
\text{NUCLEUS} & \text{dynamics}(p3) \\
\text{SATELLITE} & \text{all}(\text{generate}(p3..)):
\{ \text{P OPERATORS} \quad \text{D MODALITY} \quad \text{perm} \}
\{ \text{all}(\text{postcondition}(p3..)) \}
\{ \text{EXTERNAL} \quad \text{yes} \}
\{ \text{COMPUTATIONAL} \quad \text{yes} \}
\{ \text{BEHAVIORAL} \quad \text{no} \}
\{ \text{INTERNAL} \quad \text{no} \}
\{ \text{ILLUSTRATIVE} \quad \text{no} \}
\end{bmatrix}.
\]

The discourse goal in nucleus here, \text{dynamics}(p3), is attained using the strategy formulation in Table 9.3(e), since the one in (c) is not available in \Sigma_2’s views and the one in (d) is already registered in the explanation log. The resulting explanation part from this nucleus is

\[
\Sigma_{\text{ret},6} = \begin{bmatrix}
\text{P OPERATORS} & \text{TENSE} \quad \text{pres} \\
\text{HEAD} & \text{D MODALITY} \quad \text{perm} \\
\text{AGENS} & \text{receive} \quad p3 \\
\text{GOAL} & \text{requested,loan}
\end{bmatrix}.
\]

The satellite is an OR structure of discourse strategies and all the strategies are attempted used as a first strategy:

\[
\{ \text{expand}_{\text{subgoal}}(\text{all}(\text{generate}(p3..));\Sigma_6,\Sigma_{\text{ret},7}) \}
\{ \text{expand}_{\text{subgoal}}(\text{all}(\text{postcondition}(p3..));\Sigma_6,\Sigma_{\text{ret},8}) \}.
\]
where

\[
\Sigma_5 = \left[ \begin{array}{c}
\text{P OPERATORS} \\
\text{SALIENCE} \\
\text{VIEWS}
\end{array} \right]
\left[ \begin{array}{c}
\text{TENSE} \quad \text{pres} \\
\text{D MODALITY} \quad \text{perm} \\
\text{EXTERNAL} \quad \text{yes} \\
\text{COMPUTATIONAL} \quad \text{yes} \\
\text{BEHAVIORAL} \quad \text{no} \\
\text{INTERNAL} \quad \text{no} \\
\text{ILLUSTRATIVE} \quad \text{no}
\end{array} \right]
\]

and \( \Sigma_6 \) is \( \Sigma_5 \) without the D MODALITY attribute. Note that D MODALITY's value was included from the satellite itself and would have overruled any value of this attribute already present in \( \Sigma_2 \).

However, there is no postcondition of P3 recorded in the conceptual model, and the second subgoal can then not be attained. Handling the first subgoal, the system selects all conceptual model elements with references generate(p3,..), checks that they are not registered in the log, and unify them with \( \Sigma_5 \). This unification succeeds, the elements are registered, and the resulting structure \( \Sigma_{ret,7} \) is an AND structure of AND structures (due to the all operator):

\[
\Sigma_{ret,7} = \left[ \begin{array}{c}
\text{SALIENCE} \\
\text{P OPERATORS} \\
\text{HEAD} \\
\text{AGENS} \\
\text{GOAL} \\
\text{SALIENCE} \\
\text{P OPERATORS} \\
\text{HEAD} \\
\text{AGENS} \\
\text{GOAL}
\end{array} \right]
\left[ \begin{array}{c}
\text{TENSE} \quad \text{pres} \\
\text{D MODALITY} \quad \text{perm} \\
\text{generate} \\
\text{p3} \\
\text{requested_loan} \\
\text{TENSE} \quad \text{pres} \\
\text{D MODALITY} \quad \text{perm} \\
\text{generate} \\
\text{p3} \\
\text{customer_data} \\
\text{TENSE} \quad \text{pres} \\
\text{D MODALITY} \quad \text{perm} \\
\text{generate} \\
\text{p3} \\
\text{new_customer_data}
\end{array} \right]
\]

There are now no more subgoals to attain and the content elements constructed above can be combined to form the deep explanation:
\[ \Sigma_{ret,4} = \Sigma_{ret,5} = [\text{HEAD enablement}] \cup [\text{NUCLEUS } \Sigma_{ret,6}] \cup [\text{SATELLITE } \Sigma_{ret,7}] \]
\[ \Sigma_{ret,1} = \Sigma_{ret,2} = [\text{HEAD dynamics}] \cup [\text{NUCLEUS } \Sigma_{ret,3}] \cup [\text{SATELLITE } \Sigma_{ret,4}] \]
\[ \Sigma_{0,ret} = \Sigma_{1,ret} \]

\( \Sigma_{ret,0} \), then, is equal to the deep explanation depicted in Table 10.18.
Chapter 11

Building a Strategic Component Core

In this chapter we use the formalism from the previous chapter to show how a generic strategic component core for conceptual modeling is constructed. The core is restricted to the domain of information systems, but is otherwise applicable for several modeling languages. Rather than describing a complete strategic component core, we will in this chapter focus on the way the core is constructed, the steps involved, and the kinds of representations that have to be included. As in the last chapter, we use the modeling language $PrM$, and the conceptual model from Appendix A as an illustration.

The strategic component core generates deep explanations on the basis of information in the extended source model and is constructed in three steps: (1) define a formalism for deep explanation generation, (2) construct a generic strategic component core, and (3) build an actual strategic component core from the generic core and source model information. The formalism only specifies the necessary means for representing the strategic component, whereas step (2) and (3) concern the knowledge available in the component.

1. The strategic component’s formalism was outlined in the previous chapter. It comprises an explanation modeling language and an explanation construction algorithm.

2. The generic strategic component core constructed here is restricted to the information systems domain, but does not depend on any particular modeling language. The core makes assumptions about domain structures found in the relevant modeling languages and the strategies that should explain these structures. Views of the domain — and thereby also of the source model — are defined at this stage of the construction process.

3. The complete strategic component core is built from the generic core and is the complete runnable strategic component core. The core is constructed by making the extended source model — represented in $EML$ — available to the generic core. Also,
the generic core may here be extended with additional domain-structuring concepts, discourse strategies or views to make it more suitable in its CASE environment.

The generic strategic component core is discussed in Section 11.1, whereas the construction of a complete strategic component core is indicated in Section 11.2 and Section 11.3. In Section 11.4, we see how a tactical component can be added to the system.

11.1 The Construction of a Generic Component Core

The generic core is constructed in three steps. Since the core is intended to be independent of the CASE environment's modeling language, it must rest on some other description of the features involved in the explanation generation process. In the first step, one constructs a model that describes general structural concepts for modeling information systems, and this model will be referred to as a domain structure model.

In the second step, one defines the views that should be used in the system. A view, then, specifies which parts of the source model are visible and reflects either properties of the user (e.g. an end-user will not be bothered by details like ports in the $PrM_e$ model in Appendix A) or of the context (e.g. when exercising an information system model, we do not care about generic properties of processes).

In the last step, a set of discourse strategies is defined. The strategies are formulated in terms of elements of the domain structure model from the first step, and the use of the strategies is governed by references to views, user's knowledge, source model and already planned parts of the explanation.

Below, we will show how the domain structure model, the views, and the discourse strategies are defined and used in the component.

Model of Explanation-Relevant Domain Structures

Our model of explanation-relevant domain structures links modeling languages to discourse strategies. It represents structures that are both common to most modeling languages in the domain and relevant to the generation of explanations. The model's level of detail depends on the types of explanations needed. Trying to describe the component construction process rather than the details of a complete component, we will here adopt a very simple view of the information system domain.

The domain structure model, called DSM for short, contains two kinds of elements, concepts and relations between concepts.

The seven concepts defined in DSM are as follows:
11.1. The Construction of a Generic Component Core

![Diagram of domain structure model]

<table>
<thead>
<tr>
<th>nr</th>
<th>relation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>equivalence ((E₁,E₂))</td>
<td>(E₁) is equivalent to (E₂)</td>
</tr>
<tr>
<td>r2</td>
<td>possession ((E₁,E₂))</td>
<td>(E₁) can be said to possess (E₂) from some point of view.</td>
</tr>
<tr>
<td>r3</td>
<td>extension ((E₁,E₂))</td>
<td>(E₂) belongs to (E₁)'s extension.</td>
</tr>
<tr>
<td>r4</td>
<td>generalization ((E₁,E₂))</td>
<td>(E₁) is a generalization of (E₂).</td>
</tr>
<tr>
<td>r5</td>
<td>aggregation ((E₁,E₂))</td>
<td>(E₁) is an aggregation that includes (E₂).</td>
</tr>
<tr>
<td>r6</td>
<td>association ((E₁,E₂))</td>
<td>(E₂) is an element of (E₁).</td>
</tr>
<tr>
<td>r7</td>
<td>relationship ((E₁,E₂))</td>
<td>(E₂) is involved in relationship (E₁).</td>
</tr>
<tr>
<td>r8</td>
<td>characterization ((E₁,E₂))</td>
<td>(E₁) has a textual characterization (E₂).</td>
</tr>
<tr>
<td>r9</td>
<td>denotation ((E₁,E₂))</td>
<td>(E₁)'s purpose is described textually as (E₂).</td>
</tr>
<tr>
<td>r10</td>
<td>justification ((E₁,E₂))</td>
<td>The use of (E₁) is justified by (E₂).</td>
</tr>
<tr>
<td>r11</td>
<td>rationale ((E₁,E₂))</td>
<td>The existence of (E₁) is justified by (E₂).</td>
</tr>
<tr>
<td>r12</td>
<td>syntax ((E₁,E₂))</td>
<td>Constraint concept (E₁) has syntactic description (E₂).</td>
</tr>
<tr>
<td>r13</td>
<td>trigger ((E₁,E₂))</td>
<td>Dynamic concept (E₁) activates concept (E₂).</td>
</tr>
<tr>
<td>r14</td>
<td>terminate ((E₁,E₂))</td>
<td>Dynamic concept (E₁) terminates concept (E₂).</td>
</tr>
<tr>
<td>r15</td>
<td>succeed ((E₁,E₂))</td>
<td>The termination of dynamic concept (E₁) initiates the execution of concept (E₂).</td>
</tr>
<tr>
<td>r16</td>
<td>receive ((E₁,E₂))</td>
<td>Dynamic concept (E₁) receives flow (E₂).</td>
</tr>
<tr>
<td>r17</td>
<td>generate ((E₁,E₂))</td>
<td>Dynamic concept (E₁) generates flow (E₂).</td>
</tr>
<tr>
<td>r18</td>
<td>get ((E₁,E₂))</td>
<td>Static concept (E₁) gets flow (E₂).</td>
</tr>
<tr>
<td>r19</td>
<td>provide ((E₁,E₂))</td>
<td>Static concept (E₁) provides flow (E₂).</td>
</tr>
<tr>
<td>r20</td>
<td>precondition ((E₁,E₂))</td>
<td>(E₂) is dynamic concept (E₁)'s precondition.</td>
</tr>
<tr>
<td>r21</td>
<td>postcondition ((E₁,E₂))</td>
<td>(E₂) is dynamic concept (E₁)'s postcondition.</td>
</tr>
<tr>
<td>r22</td>
<td>duration ((E₁,E₂))</td>
<td>Dynamic concept (E₁) has duration given by (E₂).</td>
</tr>
<tr>
<td>r23</td>
<td>act ((E₁,E₂))</td>
<td>Dynamic concept (E₁) performs (E₂).</td>
</tr>
</tbody>
</table>

Figure 11.1: The structures of the domain structure model.
Modeling_concept (E) This is the most general concept in DSM. It represents any concept E — or aspect of concept — found in modeling languages in the information systems domain. Specializations of the concept are dynamic_concept, static_concept, flow, and constraint_concept.

Text (E) E is a text in some natural language.

Dynamic_concept (E) A dynamic concept E has some kind of behavior. It can be triggered by other dynamic concepts, or it can be activated as other dynamic concepts terminate. It can receive flows and can have a precondition for execution, and its results may involve generated flows and postconditions. It can be deactivated by terminating signals from other dynamic concepts, or on the basis of a specified duration.

Flow (E) A flow E can be either a data flow or a control flow. It has an origin, which provides the flow, and a destination that consumes or keeps it for later use. Both dynamic and static concepts may serve as origins and destinations.

Value_expression (E) This is any expression involving elements of conceptual models. It may be assignments or tests, but can just as well be more complicated acts like SQL queries.

Static_concept (E) Static concept E is any structure found in a conceptual model that can be counted as static with respect to the business being modeled (e.g. stores, external entities, agents, etc.).

Constraint_concept (E) An element E is a constraint concept if it is used to specify constraints in the conceptual model. The element’s relationship to an element E’ of the conceptual model is specified as an EML structure with reference constraint(E’,E).

The relations in DSM relates instances of concepts to each other and are listed in Figure 11.1. Most of them should be quite easy to understand, though it is not trivial to define the elements of DSM on a formal basis. Since concepts are often used slightly differently in different modeling languages, like the conception of process in PAISLey [256] and PrM [87], any definition tend to exclude certain languages, which is something we try to avoid here. Moreover, as the elements are only used for classification of language concepts, and not for any kind of modeling itself, a form definition is not strictly necessary for the system to work. We will return to this issue in Section 13.3 for a more deeper discussion of DSM and its relationships to rhetorical relations and discourse strategies.

Views of Source Model

We will here assume six different views, each of them suitable for describing explanation contexts and user preferences.

External The external view ignores abstractions like generalizations, aggregations, classifications, and associations. Phenomena are explained at a chosen level of detail
or conceptual level, and normally no references are made to phenomena at other levels. As such, the view leads to concise on-the-point explanations, which may be desirable for skilled users or situations where the focus is narrow (e.g. model execution).

**Internal** Contrary to the external view, the internal view emphasizes the role of abstractions and underlying modeling information to explain a phenomenon. All kinds of source model abstractions are visible, and this opens for more detailed explanations than does the external view.

**Restrictive** Sometimes, constraints on source model structures are naturally included when the structures are explained. In other cases, for example for descriptions of overall semantic properties, the constraints only clutter the explanation and are better left out of the text. In the restrictive view, the specified constraints on structures of the source model are included.

**Computational** The computational view puts emphasis on the transformation and flow of data in the model. The visible parts of the model are those that are needed to understand the computation of data values without going into the details of the model’s control aspects.

**Behavioral** In this view, the visible parts of the source model form a chain of actions, events, cause-effect relationships, etc. The idea is to focus on behavioral properties, how control signals interact and constitute the model’s behavior, and the view has proven especially useful for inexperienced or unskilled users [189].

**Illustrative** The illustrative view is meant to capture situations in which examples and graphical presentations are useful to clarify properties of source model elements.

Of course, there might be other views that are just as useful as the ones defined above, but these views serve as a good illustration of the explanation generation approach.

The definition of a view like this is simply a list of elements from the meta model, the conceptual model, and/or the execution trace. Instead of providing these lists here, we will come back to the views in Section 11.2 and see how they are associated with elements of the extended source model.

### Explanation Requests and Discourse Strategies

Our discourse strategies are text-structuring and based on a hierarchical planning algorithm with unification and backtracking. When a discourse goal is posted, a strategy matching the goal is used to decompose it into simpler subgoals or elements of the source model. Other strategies are employed to decompose the subgoals, and this process continues until there are no more goals left to attain. In the explanation request in Table 11.1, nucleus is a discourse goal that is attained by invoking the structure discourse strategy.
Table 11.1: Requesting a short explanation of the term "process".

The request includes all information that are needed to use the discourse strategies\(^1\). As shown in Table 11.1, it contains basically three kinds of information:

- An overall discourse goal is specified in NUCLEUS.
- Linguistic features that are to be given to the explanation's content elements are included in SALIENCE and P_OPERATORS. SALIENCE indicates the pragmatic functions of the elements involved, whereas P_OPERATORS specify grammatical properties of the clauses to be generated. In Table 11.1, the desired explanation's topic is process and the explanation is to be generated in the present tense.
- Features that are used to tailor the explanation to user and context are specified in VIEWS and USER. The explanation must be constructed by means of strategies and source model elements that are — or can be assumed to be — visible in the views specified in VIEWS. In Table 11.1, the views EXTERNAL and COMPUTATIONAL are explicitly specified as available, whereas the INTERNAL view is not available. Other views may be assumed to apply or not to apply as necessary. The value of USER indicates that only elements known or unknown to the user should be included, though both categories of elements can be included if no USER value is given (as in Table 11.1).

The view specification is the union of the user characterization submodel and the context specification. The value of USER indicates the user knowledge strategy, deciding how the user knowledge submodel should affect the generation process.

Both schema-oriented and operator-oriented strategies are useful. Operator-oriented strategies include a rhetorical relation as HEAD, which specify the way the satellite is related to the nucleus. In schema-oriented strategies, like the request in Table 11.1, there is no rhetorical relation (HEAD is nil) and the nucleus contains all the subgoals. If a nucleus or a satellite in themselves specify several goals, an implicit rhetorical relation list is assumed to relate to the goals.

Defining an operator-oriented discourse strategy like the one in Figure 11.2, we usually starts off with the discourse goal. This is the topic of the explanation and corresponds to

\(^1\)Except for the information in the extended source model of course.
aspects of some element(s) of the source model. Here, the goal is error(E,R,D), which means that we are to explain that element E in diagram D is a violation of verification rule R. A discourse strategy defines the application of a rhetorical relation in a particular context, so the next step is to decide on an appropriate rhetorical relation to use. "Evidence" seems to be suitable here, since it can be given a topic-dependent definition that decompose the discourse goal into two simple subgoals: Nucleus informs the user that the model is illegal, whereas satellite provides the evidence for this conclusion, explaining what element is illegally modeled and what verification rule is violated. This leads to two subgoals, illegal(D) and correction(E,R), that both require new discourse strategies for further decompositions. To complete the definition of "evidence", one might also add conditions about source model properties (precondition) or available views (views). This, however, is not necessary here, as the relation definition is applicable for all kinds of a posteriori verification errors and in all views.

When low-level discourse strategies are defined, the rhetorical relations will often directly reflect structures of the domain structure model. An interesting question, then, is whether we could make it without the DSM and instead relate source model elements directly to
rhetorical relations used in the strategies. This we will come back to in Section 13.3.

In tables 11.2 through 11.4, the discourse strategies defined so far are described. The rhetorical relations used and the formal definitions of these strategies are found in Appendix E.

Setting up the discourse strategies in a hierarchical planning system, one needs special mechanisms for

- being able to include all strategy goals or source model elements that satisfy a search goal (instead of just one),

- changing the linguistic information, the view specification, or the user knowledge strategy from the request or strategies used at a higher level of the explanation process.

Usually, when a search goal is posted, the component is satisfied when it can find one strategy or source model element that matches the goal. With reference to the request in Table 11.1, for example, one is interested in generating only one explanation that satisfies the request specification. However, if all the flows to a particular process should be included in an explanation, one must explicitly indicate this using a special all operator (see Section 10.3). The argument of all is a discourse goal or a reference to a source model element, and its purpose is to produce a list structure containing all those elements that satisfy the search goal specified as its argument. For example, the nucleus all(constraint\(E\)) of the constraints strategy in Appendix E means that all elements specifying a constraint on \(E\) should be included in the explanation. The list operator is used in a similar manner.

Changing the linguistic information, the view specification, or the user knowledge strategy is usually not needed, but there are some few cases where it might be useful. For example, when describing an object of some kind, the request may specify that only unknown properties should be included. However, if one of these properties are very complicated to explain, one should change the overall user knowledge strategy for that particular part and describe the property itself with reference to known phenomena. Also, even though history explanations are specified to be in the past tense, the present tense might be adopted when describing the underlying rules used in the computation. Looking at the first formulation of the background strategy in Appendix E, we see how this is done. When this strategy is used, the nucleus part must be planned without access to the RESTRICTIVE view, even if that view is used at higher levels of the generation process. Similarly, deontic modality is made neutral for the clauses in that part of the explanation.

The difference between discourse goal and rhetorical relation should also be noted. Contrary to many other generation systems, we distinguish between the two, specifying the discourse goal as a reference and the relation as a head value. The goal is used in the deep generation process, and the resulting deep explanation structure is really a tree of connected and instantiated discourse goals. The rhetorical relations govern the design of the strategies, in the sense that we are not allowed to define an operator-oriented strategy in which there is no rhetorical relation between nucleus and satellite. And when the
<table>
<thead>
<tr>
<th><strong>strategy</strong></th>
<th><strong>description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity(E)</td>
<td>The strategy describes the purpose of a system activity. If that activity is part of a higher-level activity, and the EXTERNAL view applies, the purpose of the higher-level activity is included as circumstantial information.</td>
</tr>
<tr>
<td>Aggregations(E)</td>
<td>The strategy describes a part element P of aggregation element E. Nucleus presents the aggregation declaration, and satellite describes the properties of P. Neither the COMPUTATIONAL nor the BEHAVIORAL view can apply if this strategy is to be used.</td>
</tr>
<tr>
<td>Application(C,F)</td>
<td>Element C is responsible for act or calculation F. If C is a conceptual model element, it is just presented to the reader; if C is a trace element, its type is presented. Additionally, if one of the views INTERNAL, COMPUTATIONAL, or BEHAVIORAL is available, it is shown how C does or causes F, using the computation strategy.</td>
</tr>
<tr>
<td>Associations(E)</td>
<td>The strategy describes association element E's member element M. Nucleus presents the association declaration, and satellite describes the properties of M.</td>
</tr>
<tr>
<td>Background(E)</td>
<td>If the argument E is a modeling concept, the strategy is used to present the declaration of this concept. Otherwise, E is presented as it is.</td>
</tr>
<tr>
<td>Cause(E)</td>
<td>The strategy is used to explain how some result E (from the trace) is computed from the conceptual model. There are two sub-strategies for generating this explanation: (1) If EXTERNAL is no and INTERNAL is yes, we go to the low-level element that produced E and explain how it works and what values it used. Actor(C,F,E) returns the element C whose post-condition, generation, or act produced E. F is the actual rule used for the production of E. (2) If EXTERNAL is yes and INTERNAL is no, the result E is explained by means of a higher-level activity than C from (1). Generalized Actor(G,F,E) returns G, which is an abstraction of C, and the explanation will therefore be given at a more superficial level.</td>
</tr>
<tr>
<td>Computation(C,F)</td>
<td>Element C is responsible for act or calculation F. Explaining how C does or causes F, C's condition and/or enablement can first be included, as well as the purpose of C (or its type). Element F (or its type if F is a trace element) is then presented, before the values used in calculation or act F are described. If the INTERNAL and the RESTRICTIVE views apply, the computations leading to these values are also explained.</td>
</tr>
<tr>
<td>Constraints(E)</td>
<td>The strategy can only be used in the RESTRICTIVE view. Its purpose is to list all the constraints associated with an element of the meta model or the conceptual model.</td>
</tr>
<tr>
<td>Correction(E)</td>
<td>Nucleus presents the model error from the verification check. Satellite, in contrast, informs the reader about the rule that was violated. Since satellite contains a requirement to the conceptual model, its paraphrase is given the linguistic feature d_modality = obl (for obligatory).</td>
</tr>
</tbody>
</table>

Table 11.2: Discourse strategies.
<table>
<thead>
<tr>
<th><strong>strategy</strong></th>
<th><strong>description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Decomposition(E)</td>
<td>Element ( E ) is decomposed, using aggregation or generalization, and each of its subelements is explained by means of the properties strategy.</td>
</tr>
<tr>
<td>Description(E)</td>
<td>If the source model contains a characterization of ( E ) or information about elements equivalent to ( E ), the strategy includes them as a satellite to a background description of ( E ) in nucleus.</td>
</tr>
<tr>
<td>Dynamics(E)</td>
<td>This strategy explains dynamic aspects of source model element ( E ). If the Behavorial view applies, the explanation will include the control signals that trigger and terminate ( E ), as well as those produced by ( E ). If the Computational view is applicable, ( E )'s preconditions and post-conditions are taken into consideration, and if ( E ) is a process the flows received and generated by ( E ) are put into the explanation. Provided that it is recorded in the source model, the act performed by ( E ) is also included. If ( E ) is decomposed and Internal is yes, its decomposition is also explained.</td>
</tr>
<tr>
<td>Error(E,R,D)</td>
<td>The strategy states that diagram ( D ) is illegal in nucleus, whereas the satellite provides the evidence by calling the correction strategy with the appropriate arguments.</td>
</tr>
<tr>
<td>Generalizations(E)</td>
<td>The strategy describes a specialization ( S ) of the generalized element ( E ). Nucleus presents the generalization declaration, and satellite describes the properties of ( S ). Neither the Computational nor the Behavioral view can apply if this strategy is to be used.</td>
</tr>
<tr>
<td>Instantiation(E)</td>
<td>The strategy simply presents some model instance ( E ) and describes its basic properties.</td>
</tr>
<tr>
<td>Justifications(E)</td>
<td>This strategy constructs a justification of some model element. If present in the source model, the justification(E) element is used to explain why the element was included in the model, and the rationale(E) is added to justify why ( E ) is modeled as it is.</td>
</tr>
<tr>
<td>Need(E)</td>
<td>The strategy justifies a user input requested by the system. If Internal is yes and either Computational or Behavioral is yes, it first explains the computations leading to the request. No matter the view specification, it describes the activity in which the input is to be used, invoking the activity strategy.</td>
</tr>
<tr>
<td>Paraphrase(E)</td>
<td>In the Restrictive view, it calls conventions to list all the conventions associated with the specified element ( E ). D ModalitY is then set to ob1, since the conventions state what has to be the case. Otherwise, it just paraphrases the element in natural language.</td>
</tr>
<tr>
<td>Possessions(E)</td>
<td>Describing the possessions of element ( E ), the strategy presents an element ( P ) possessed by ( E ). If the Internal view is used, the properties of the possession ( P ) is included as a satellite.</td>
</tr>
<tr>
<td>Presentation(E)</td>
<td>In the Illustrative view, the specified element ( E ) is paraphrased and illustrated with a view of the source model, if there is any appropriate view available. If Illustrative is no, ( E ) is just paraphrased.</td>
</tr>
</tbody>
</table>

Table 11.3: Discourse strategies.
### Table 11.4: Discourse strategies.

deep explanation is constructed, the relations are associated with the non-leaf nodes of the tree structure and are later used by the textual surface generation component to find suitable textual realizations of the structure. So, just to take an example, in the error strategy, \textit{error}(E,R,D) is the discourse goal and evidence the associated rhetorical relation (see Figure 11.2). When the strategy is to be used, the request will contain the goal \textit{error}(E',R',D'), where E', R' and D' are instantiated, but in the generated deep explanation evidence is put into the structure instead of \textit{error}(E',R',D') (see Figure 12.4).

### 11.2 Parts of an Extended Source Model

The extended source model is constructed in four steps:
1. An explanation meta model of $PrM$, is defined.

2. The explanation conceptual model is constructed from instantiations of meta model elements.

3. Explanation execution traces are constructed from instantiations of conceptual model elements.

4. For each element of the source model (meta model, conceptual model, and instances and traces), a list of view definitions decides in which views the element is visible and the user’s knowledge of it. The information is added to the elements as view specifications and user knowledge submodel indications. If the information is unknown, the attributes are left out of the element structure.

In the construction process, there are two aspects that have to be given special attention: (1) how model instances are created, and (2) how constraints are specified. After discussing these issues we will return to the source model and show how parts of the meta model, the conceptual model, and the trace are represented in EML.

Creating EML Representations

We distinguish between three kinds of elements, structure elements, model constraint elements, and constraint elements.

Structure elements A structure element represents an unrestricted structural property and can be used at the meta level, at the conceptual level, and at the trace or instance level. Both structure elements themselves and their instances can be instantiated to form new structure elements at lower conceptual levels. The reference of a structure element is on the form

$$<\text{dsm\_element}>,$$

where $<\text{dsm\_element}>$ is a concept with one argument or a relation with two arguments found in the domain structure model.

Model constraint elements A model constraint element is declared at the meta level and is instantiated to specify a constraint in the conceptual model. The instantiated model constraint element at the conceptual level cannot be further instantiated. It has a reference on the form

$$\text{model\_constraint}(<\text{dsm\_element}>,<\text{constraint}>),$$

where $<\text{constraint}>$ is a constraint on the structure element identified by reference $<\text{dsm\_element}>$. 
11.2. Parts of an Extended Source Model

Constraint elements These elements specify syntactic constraints on the structures of the conceptual model. They are not instantiated, but form a description of the syntax of the modeling language used in the environment. They also correspond to verification checks in the verification component. The references of these elements are on the form

\[ \text{{constraint(<dsm\_element>) or}} \]
\[ \text{{constraint(<model\_constraint\_element>)}} \]

where <dsm\_element> is the reference of a structure element and <model\_constraint\_element> is the reference of a model constraint element.

Elements of the explanation meta model are defined as specializations of DSM elements. All other elements of the source model are instantiations of more generic elements, conceptual model elements from meta model elements and trace elements from conceptual model elements. A number of rules are applied in this instantiation process, and these are briefly described below.

Instantiation rule 1 (Concept instantiation)

Given the EML structure \( \Gamma = \rho : \Sigma \), where reference \( \rho \) is a one-place predicate with argument \( \gamma \) and \( \Sigma \) is a structure. Creating an instance \( \Gamma' \) of \( \Gamma \), we assign an instance \( \gamma' \) to the argument of \( \Gamma' \)'s reference \( \rho \). The reference \( \rho' \) of \( \Gamma' \) is made from \( \rho \) by substituting the instance \( \gamma' \) for \( \rho \)'s argument \( \gamma \). \( \Gamma' \)'s structure is given as

\[
\begin{bmatrix}
\text{HEAD} & \gamma \\
\text{ZERO} & \gamma'
\end{bmatrix}
\]

The instances put into the structure may be supplemented with values for T.OPERATORS, ROLES, POSSESSOR, and RESTRICTORS.

Instantiation rule 2 (Relation instantiation)

Given the EML structure \( \Gamma = \rho : \Sigma \), where reference \( \rho \) is a two-place predicate and \( \Sigma \) is a structure. An instance \( \Gamma' = \rho' : \Sigma' \) of \( \Gamma \) is created by assigning instances to the arguments of \( \rho \). These are substituted for their types in \( \Sigma \) to form \( \Sigma' \); \( \rho' \) is constructed from \( \rho \) by replacing the arguments with their instances if the instances are single words and leaving them open if more complex instances are used.

The instances put into the structure may be supplemented with values for T.OPERATORS, ROLES, POSSESSOR, and RESTRICTORS.

Instantiation rule 3 (Structure Transformation)

Given the following EML structure \( \Gamma \):

[Description of the structure]
\[ \rho : \begin{bmatrix} \text{HEAD} & \text{relate} \\ \text{ZERO} & \beta_1 \\ \text{REFERENCE} & \beta_2 \end{bmatrix}, \]

where \( \beta_1 = [\text{HEAD} \gamma, \text{RESTRICTORS} \alpha] \cup \beta'_1, \alpha \) is a semantic role, and \( \beta_2 \) is a structure. \( \Gamma \) can then be replaced by \( \Gamma' \) given as \( \rho' : [\alpha \gamma] \cup \beta'_1 \cup \beta_2 \).

**Instantiation rule 4 (Structure Combination)**

Given two structures \( \rho_1 : \Sigma_1 \) and \( \rho_2 : \Sigma_2 \) such that \( \Sigma_1 \)'s head is equal to \( \Sigma_2 \)'s head. If \( \Sigma_1 \) and \( \Sigma_2 \) are unifiable, they can be replaced by their unification and we get the elements \( \rho_1 : \Sigma_1 \cup \Sigma_2 \) and \( \rho_2 : \Sigma_1 \cup \Sigma_2 \).

For example, the conceptual model element in Table 11.9(a) is a concept instance of the element in Table 11.5(a), whereas the element in Table 11.9(b) is a relation instance of the one in Table 11.5(e). The element in Table 11.9(c) is an instance of the element in Table 11.5(d) which has been refined using the structure transformation rule\(^2\). We will return to the structure combination rule in connection with the PPP realization in Chapter 12.

The extended source model is constructed on the basis of representations in the rest of the conceptual modeling environment. If no explicit modeling meta model is defined, the explanation meta model has to be produced manually, otherwise translation rules are responsible for the creation of the EML-represented source model. The views are then added to the source model by means of the \text{VIEWS} attribute, whereas the user knowledge submodel is associated with the \text{USER} attribute. These two attributes may either be fully integrated with the source model or just added when an explanation is to be generated.

**Extended Meta Model**

The elements of the meta model are of all three kinds. Structure elements are declared as specializations of DSM concepts and DSM relations, as shown in Table 11.5. A meta model concept is defined using a structure on the form

\[
<\text{dsm\_concept\_name}>(<\text{meta\_model\_concept}>):
\begin{bmatrix}
\text{HEAD} & <\text{dsm\_concept\_name}> \\
\text{ZERO} & <\text{meta\_model\_concept}> 
\end{bmatrix},
\]

where \( <\text{dsm\_concept\_name}> \) is the name of a DSM concept and \( <\text{meta\_model\_concept}> \) is the name of a specialization of \( <\text{dsm\_concept\_name}> \) included in the meta model. When relational elements of the meta model are defined, the generic structure

\(^2\)In addition, a \text{USER} attribute has been added.
11.2. Parts of an Extended Source Model

\[ \langle \text{dsm\_relation\_name} \rangle \langle \text{meta\_model\_concept}_1, \text{meta\_model\_concept}_2 \rangle: \]

\[
\begin{align*}
\text{HEAD} & \quad \text{meta\_model\_relation} \\
\langle \text{ROLE}_1 \rangle & \quad \text{meta\_model\_concept}_1 \\
\langle \text{ROLE}_2 \rangle & \quad \text{meta\_model\_concept}_2
\end{align*}
\]

is used. \( \langle \text{meta\_model\_relation} \rangle \) is a specialization of relation \( \langle \text{dsm\_relation\_name} \rangle \), \( \langle \text{role}_i \rangle \) is a semantic role, and \( \langle \text{meta\_model\_concept}_i \rangle \) is a concept from the meta model. In Table 11.5 we have shown some of the structure elements of the extended meta model for \( PrM_r \). The elements are not given any view specification and can consequently be assumed visible in any view. The user’s knowledge of them is unknown, which means that the elements can be included no matter the user knowledge strategy.

In Table 11.6 there are two model constraint elements for the use of ports in \( PrM_r \). As seen from the structure, the port expression for input flows and output flows are interpreted as constraints in the conceptual model. Both elements are visible in the RESTRICTIVE view and hidden in the BEHAVIORAL view. In general, the reference of a model constraint element contains two arguments: (1) a reference of a structure element of the meta model and (2) a constraint concept that is used to specify constraints on these kinds of structure elements. The associated structure is a clause that is neither an instance nor a specialization of any other structure.

Four constraint elements for \( PrM_r \) are shown in Table 11.7 and Table 11.8. The reference’s argument is a structure element or a model constraint element from the meta model, whereas the structure is simply a clause specifying the constraint in some language. The three constraint elements in Table 11.7 are declared as known to the user, i.e. the user knows that a process has a purpose and that it must receive and generate at least one flow. The element in Table 11.8 says that every process must be given an output port specification, i.e. it must generate a specified combination of flows.

Extended Conceptual Model

The conceptual model is formed from structure elements and model constraint elements, and these elements are all instances of meta model elements. In Kung’s terminology, the instances of structure elements and model constraint elements are referred to as concret and abstract knowledge, respectively [137]. The creation of instances is governed by the four instantiation rules above and is illustrated by some examples in Table 11.9. In (a), process P3 is defined, and in (b), it is declared to generate the flow Recommendation. The purpose of process P3 is given in (c), and the user is here assumed to know that purpose ((USER yes)). The postcondition in (d) illustrates how formula is used to include formal expressions in EML representations. The element states that rule r3.2.1’s postcondition is \( \text{offer\_recommendation} = \maximunm\_loan\_limit \). At last, the element in (e) shows how ports are represented as model constraint elements. The element corresponds to P3’s input ports in Figure A.2 and says that the process receives Requested\_loan and either Customer\_data or New\_customer\_data.
Table 11.5: Some structure elements in the meta model of \( PrM_r \).

Table 11.6: Two model constraint elements in the meta model of \( PrM_r \).

Extended Execution Trace

Only structure elements are used at the trace and instance level. They are instances of structure elements in the conceptual model and are created using the instantiation rules above.

In Table 11.10, some trace elements are shown. This trace assumes a simple execution of the model in Appendix A, in which the user applies for a $100 loan (element (b)). The balance of her account is $80 (element (c)), and the system’s computation leads to a loan offer of $80 (element (d)). In (a), the rule instance leading to the computation of the loan
Table 11.7: Some of the constraints on structure elements in the meta model of PrMr.

offer is shown. Using these instances, the system is able to track down the conceptual model elements activated in a particular execution and use them to explain aspects of the execution.

11.3 Generating Deep Explanations

We are now able to show how deep explanations can be generated in response to user questions or error messages from the verification component. Below, we describe the generation of a few representative deep explanations based on the language and the model
constraint(model_constraint(generate(process,flow),flow_combination)):

Table 11.8: A constraint on a model constraint element in P*M’s meta model.

(a)  
    modeling_concept(p3):
    [HEAD process ]
    [ZERO p3]

(b)  
    generate(p3,recommendation):
    [HEAD generate     ]
    [AGENS p3          ]
    [GOAL recommendation]

(c)  
    purpose(p3,..):
    [HEAD process ]
    [AGENS p3]
    [GOAL loan_request]
    [USER yes]

(d)  
    postcondition(r3.2.1,formula(offer_recommendation = maximum_loan_limit)):
    [HEAD result     ]
    [AGENS r3.2.1]
    [GOAL formula(offer_recommendation = maximum_loan_limit)]

(e)  
    model_constraint(receive(p3,..),..):
    [HEAD generate        ]
    [AGENS p3             ]
    [GOAL formula(requested_loan AND (customer_data XOR new_customer_data))]
    [VIEWS RESTRICTIVE yes]
    [           BEHAVIORAL no]

Table 11.9: Elements of the explanation conceptual model for the system in Appendix A.
Table 11.10: Portions of a trace from executing the bank model in Appendix A.

from Appendix A, though in Chapter 12 we will return to the discourse strategies introduced above and discuss what kinds of explanation types they can support in real CASE environments.

Consider the user question

user: "What is a process?"

This question can of course be given several answers, depending on the context and the user at hand. It might be the case, for example, that the user has never heard of the term and is just interested in an overall characterization of its function. Though, she might also be in the process of modeling, in which case she could need an explanation of the syntactic properties of processes. Other types of explanations may also be relevant, but we will here restrict ourselves to these two types for explanatory purposes.

An explanation request corresponding to user's question is constructed. Let us first assume an unskilled user requesting a superficial description of what a process is. The situation is characterized by the views external, computational, behavioral, and internal, and we get the explanation request as shown in Table 11.11. The values of tense and topic are deduced from the question itself. Using the discourse strategies explained earlier, the component generates the deep explanation in Table 11.11. The deep explanation is an RST structure, in which the rhetorical relations dynamics, interpretation, illustration, and enablement are included. There are four clauses selected from the source model, whereas the fifth leaf node, the display command, is a call to the view generation component for it to display a process. It should be noted, though, that this is not the only explanation that can be generated from the specified request. In the explanation generation process, the component assumed that the view illustrative should be used. If the component had not made this assumption, the display command would not have been included in the deep explanation.

In Figure 11.3, we have given the deep explanation an informal presentation. The deep explanation, which corresponds to the one in Table 11.11, is here given a more simple notation using a tree structure and a table. The tree structure shows the RST structure and illustrates the nested structures of nuclei (marked n) and satellites (marked s). The content
Table 11.11: Request and deep explanation for an answer to "What is a process?"
elements are specified in the table by means of their references and the attribute value pairs that are added to the references' structures from the source model.

We can also answer the question above from a more detailed syntactic point of view. We then specify the view RESTRICTIVE as present, and we can also say that the ILLUSTRATIVE view cannot be used. Moreover, assuming a reasonable knowledgeable user, we can restrict the explanation to including only unknown information. The request formulated in this case is shown in Figure 11.4, together with the resulting deep explanation and a possible surface explanation. In the explanation generation here, the RESTRICTIVE view causes the generator to include constraint elements instead of structure elements. For example, the element indicated by reference constraint(model_constraint(generate(process,)), flow_combination), which is shown in Table 11.8, states that a port combination must be specified for every process's flow generation by means of one or more ports. Also, since only unknown elements were to be included, the fact that every process must be given a textual purpose is ignored by the generator.
A process is a dynamic concept. Dynamically, every process must receive at least one flow, and a port combination must be specified for every process's flow reception by means of one or more ports. Every process must generate at least one flow, and a port combination must be specified for every process's flow generation.

Figure 11.4: Detailed syntactic explanation of a process.

The same discourse strategies may be used at the meta level, the conceptual level, and the trace level. To take an example, issuing the question

user: "What does P3 do?"

in the external and computational view and hidden to the behavioral, internal, and illustrative view, the user is given the explanation as shown in Figure 11.5. The structure of this explanation is almost similar to the one in Figure 11.3, but here properties of the conceptual model rather than of the modeling language are explained.

At last, we can indicate the generation of history explanations, which is based on information in both trace and conceptual model. With reference to the possible execution described earlier and the trace elements in Table 11.10, the user may ask the following question:
11.4 Tactical Component

We can assume that the views INTERNAL, COMPUTATIONAL, and BEHAVIORAL are all available, and the views EXTERNAL and RESTRICTIVE are hidden. The discourse goal is set to cause(8013), and the request in Figure 11.6 is constructed and sent to the strategic component. Using the strategies cause, application, and computation, the component generates the explanation shown in Figure 11.6. Note that the explanation log here prevents element D in Figure 11.6 from being included also in enablement's satellite.

11.4 Tactical Component

The responsibility of the tactical component is to realize deep explanations like the one in Table 11.11 as surface explanations. Since the graphical parts of the explanation is taken
Figure 11.6: Explaining the result of an execution.

care of by the view generation component, the tactical component here needs only handle textual elements of explanations.

Basically, textual surface generation from our RST-based deep explanations can be divided into two subprocesses:

- **inter-clause realization**, which concerns the coordination of clauses into coherent paragraphs of texts; and

- **intra-clause realization**, which concerns the translation from EML-represented clauses to clauses in natural language.
We will here just indicate the main tasks involved in these two processes. In [8, 126], the construction of a tactical component connected to our strategic component is treated in detail.

**Inter-Clause Realization**

In this process, suitable text structures are constructed on the basis of RST structures. Content elements are not considered, but the rhetorical relations included are used to generate texts that are intended to be coherent. Inter-clause realization from our deep explanations involves at least the following tasks:

- **Pruning** The display commands included in the deep explanations must be separated out from the structure and sent to the view generation component.

- **Linearization** In RST, there is no linear ordering of nucleus and satellite. When the deep explanation is to be realized, one must decide whether the satellite should precede or succeed its nucleus.

- **Sentence scope** A content element denotes a clause which can be realized as a separate sentence or as part of a larger sentence. Which solution to choose may depend on
Chapter 11. Building a Strategic Component Core

\[ A = \begin{bmatrix} \text{SALIENCE} & \text{TOPIC process} \\ \text{P.OPERATORS} & \text{TENSE pres} \\ \text{HEAD} & \text{dynamic_concept} \\ \text{ZERO} & \text{process} \end{bmatrix} \]

\[ \downarrow \]

(a) "A dynamic concept process"
(b) "A process is a dynamic concept"

Table 11.12: Two possible clause realizations in intra-clause realization.

the structure of rhetorical relations and/or adjacent content elements.

Cohesive constructions Some of the rhetorical relations in the deep explanation should be realized as cohesive constructions to make the rhetorical relationships between text segments more apparent.

In Figure 11.7, we have shown how the deep explanation from Figure 11.3 (and Table 11.11) is transformed to linear text in the inter-clause realization part.

Of course, this inter-clause realization can be made a lot more powerful and linguistically advanced. In [8], two additional tasks are included in inter-clause realization: pronominalization and clause type selection. Other tasks at this stage are suggested in [116, 131, 160].

Intra-Clause Realization

Realizing our content elements as clauses in natural language, one may first need to complete the representations with default values. When this is done, clauses are realized following the traditional tasks of choosing lexical items, and making syntactic and morphological decisions.

Inter-clause realization and intra-clause realization are not totally separated. In particular, co-operation is necessary when several clause realizations are possible, when no clause realization is possible, and when there are more complex relationships between natural language clauses, cohesive constructions, and sentence boundaries [7, 8, 111, 114]. For example, the choice of realization (a) or (b) in Figure 11.12 depends on how the element can be combined with adjacent content elements, which in turn is related to the RST structure and the rhetorical relations involved.
In this part, the strategic component core is validated and compared to other explanation generation systems. The part is structured as follows:

**Chapter 12** The strategic component core is realized for the modeling languages used in the PPP CASE environment. A possible realization for rule-based languages and state-based languages is also indicated.

**Chapter 13** The explanation types supported by the component core and their relationships to the standards of textuality are discussed. Also, some of the problems and limitations of our explanation component are discussed.

**Chapter 14** Our component core is compared with existing explanation generation systems and with systems using comparable internal structures in general.

Basically, Chapter 12 will show that the strategic component core delivers what it was supposed to, Chapter 13 that the delivered product is of an acceptable quality, and Chapter 14 that the component is an improvement on existing explanation generation systems.
Chapter 12

Realization of Strategic Component Core

In this chapter, we show how a full strategic component core can be built and exploited in real CASE environments. An architecture for including the component in the PPP CASE environment is developed, and it is shown how the component can support the explanation types needed in conceptual modeling. Discussing the integration requirements from Chapter 7, we also indicate how the same component could work on other kinds of modeling language.

12.1 PPP with Explanation Component

The PPP CASE environment is presented in Appendix B. Its modeling language, which comprises four sublanguages, is explained with reference to a simple banking system. In the following, we will assume the same banking model and build on the same modeling and validation environment as described in the appendix. However, even though we use the generation of Ada code to show the potential of explanation generation here, the discussion is just as relevant to the other techniques for model execution developed in PPP.

In the new PPP environment, which is discussed in [250], comprehensive PPP conceptual models are incrementally constructed and validated. Verification messages are given on request, and components for executing models and generating views of conceptual models are included. A meta model of the PPP language is not explicitly represented in the environment, though it is of course implicitly defined by means of the environment’s functionality.

To satisfy the explanation needs in conceptual modeling, the strategic component core needs access to the conceptual model, the definition of the modeling language, the results
from verifying the conceptual model, the traces from executing it, and the appropriate views generated from it. The interface between the strategic component core and the other representations in PPP is shown in Figure 12.1. After describing the internal structures of the strategic component core, we will return to this figure and show how the core is integrated into the CASE environment.

12.2 EML Representations in PPP

The explanation meta model of PPP includes structure elements, constraint elements, and model constraints elements. Some few meta model elements are shown in Table 12.1. In (a) and (b), the concepts entity class and data flow are defined, respectively. The cardinality concept is declared in (c), and in (d) entity classes are related to relationship classes. The constraint element in (e) says that an entity class can be related to any number of relationship classes. Some other elements of the meta model are equivalent to the elements found in tables 11.5 through 11.8.

In Appendix B, we use a small conceptual model as an illustration to the PPP language. This model will also be used here, and the conceptual model elements shown in Table 12.3 are taken from this model. The element in Table 12.2, which is also included in the ex-
planning conceptual model, is constructed by combining two instances of the meta model element (see the structure combination rule in Section 11.2):

Since PPP is an executable language, we can construct a trace exposing the internal dynamic behavior of our conceptual model. Using the execution in Appendix B as a basis, we get trace elements like those given in Table 12.4. A user wants to make a withdrawal of $100. The balance of his account is found to be just $80, however, so the system sends the flow Withdrawal_rejection saying that the available amount is $80. Note that the sending of a flow in Figure 12.4(c) corresponds to an instance of that flow, which is represented by adding an index to the flow’s name.

### 12.3 Explanation Generation in PPP

We have earlier discussed a number of areas of conceptual modeling in which explanations may be useful. These comprise language semantics, language syntax, verification checks, model inspection, modeling deliberations, and model execution.

For the explanation component to support explanations of these kinds, there are two prerequisites: First, the necessary information must be available in the source model. Explaining the syntax of the modeling language, for example, the meta model must be rather detailed and must specify constraint elements for each of its structure elements. Secondly, the generic part of the explanation component must be able to exploit the information to
relationship(transaction,update):

\[
\begin{array}{c}
\text{HEAD} \\
\text{ZERO} \\
\text{REFERENCE}
\end{array}
\begin{array}{c}
\text{HEAD} \\
\text{RESTRICTORS} \\
\text{HEAD}
\end{array}
\begin{array}{c}
\text{transaction} \\
\text{agens} \\
\text{update}
\end{array}
\]

+ relationship(account,update):

\[
\begin{array}{c}
\text{HEAD} \\
\text{ZERO} \\
\text{REFERENCE}
\end{array}
\begin{array}{c}
\text{HEAD} \\
\text{RESTRICTORS} \\
\text{HEAD}
\end{array}
\begin{array}{c}
\text{account} \\
\text{goal} \\
\text{update}
\end{array}
\]

\[\downarrow\]

relationship(update,account):

\[
\begin{array}{c}
\text{HEAD} \\
\text{AGENS} \\
\text{GOAL}
\end{array}
\begin{array}{c}
\text{update} \\
\text{transaction} \\
\text{account}
\end{array}
\]

Table 12.2: Combining two elements of the conceptual model.

(a) modeling_concept(account):

\[
\begin{array}{c}
\text{HEAD} \\
\text{ZERO}
\end{array}
\begin{array}{c}
\text{entity_class} \\
\text{account}
\end{array}
\]

(c) modeling_concept(pld1.2.3):

\[
\begin{array}{c}
\text{HEAD} \\
\text{ZERO}
\end{array}
\begin{array}{c}
\text{send\_box} \\
\text{pld1.2.3}
\end{array}
\]

(b) modeling_concept(flow):

\[
\begin{array}{c}
\text{HEAD} \\
\text{ZERO}
\end{array}
\begin{array}{c}
\text{data\_flow} \\
\text{withdrawal\_rejection}
\end{array}
\]

(d) generate(pld1.2.3,withdrawal_rejection):

\[
\begin{array}{c}
\text{HEAD} \\
\text{AGENS} \\
\text{GOAL} \\
\text{RECIPIENT}
\end{array}
\begin{array}{c}
\text{send} \\
\text{pld1.2.3} \\
\text{withdrawal\_rejection} \\
\text{customer}
\end{array}
\]

(e) precondition(pld1.2.2,formula(account_balance < amount)):

\[
\begin{array}{c}
\text{HEAD} \\
\text{GOAL} \\
\text{FORCE}
\end{array}
\begin{array}{c}
\text{enable} \\
\text{pld1.2.2} \\
\text{formula(account\_balance < amount)}
\end{array}
\]

Table 12.3: Elements of the conceptual model.
Table 12.4: Elements of the execution trace.

construct the desired explanations.

In the following, we show explanations generated by the strategic component core. Since the focus here is on the types of explanation, we will not look into how the meta model could be improved to support explanations of a higher textual quality. Instead, there will be an assessment of the strategic component core's abilities to utilize the meta model and generate the needed types.

Language Semantics

As an example, we look at the cardinalities used in the PhM language and assume the following question from the user:

"What do cardinalities in PhM express?"

Since the topic here is language semantics, the context will exclude the RESTRICTIVE view. If the user preferences submodel says that the views INTERNAL and EXTERNAL apply, but not the views ILLUSTRATIVE and RESTRICTIVE, we can construct an explanation request and send it to the strategic component. The resulting request and the deep explanation generated is shown in Figure 12.2. The third formulation of the structures strategy is first used (see Appendix E). Its nucleus is generated by means of the strategies properties, description, background, presentation, and paraphrase. The satellite explains all possible cardinalities, using the strategies aggregations, presentation, properties, semantics, and values. The content elements here are just informally stated; in reality, they are EML structures corresponding to FG predications. For example, the content element "A cardinality characterizes an entity class's involvement in a relationship" is represented in the deep explanation as
Figure 12.2: Request and explanation structure for the question about cardinalities in PhM.
12.3. Explanation Generation in PPP

Semantic descriptions of other language concepts are given in a similar manner.

Language Syntax

Syntactic descriptions of modeling language are generated from constraint elements of the explanation meta model. Consider the following question:

"How are entities modeled?"

Since the question is about syntactic properties, the RESTRICTIVE view is chosen to apply. Also, analyzing the user at hand, the system finds that the ILLUSTRATIVE view should not be used, whereas both the EXTERNAL and the INTERNAL should be visible. The complete explanation request is shown in Figure 12.3, together with the explanation structure generated by the strategic component core. In this explanation, the structures strategy first includes a specialization of entity class by invoking generalizations(entity_class). The specialization is subclass, which is syntactically explained from the properties strategy. The way entity classes and relationship classes are related, is then explained by means of relationships(entity_class) and properties(relationship_class). Throughout the explanation process, the RESTRICTIVE view and the paraphrase strategy make sure that constraint elements rather than structure elements are included.

It is worth noting that the generation of both semantic and syntactic descriptions starts out with the discourse goal structures. However, due to their different use of the RESTRICTIVE view, different strategies are employed to generate the explanations. In semantic descriptions, structure elements are preferred; in syntactic descriptions, constraint elements are used as far as possible.

Verification Checks

The verification component checks that the conceptual model is in accordance with what is specified in the meta model. If the user asks for a check on a posterial rules and a violation is detected, the component requests an explanation that informs her about the situation. The request specifies both the error in the conceptual model and the meta model element that was used to detect the error.
Two discourse strategies are needed to explain the violation of an a posteriori verification rule: model.error and correction. In model.error the precondition finds the abstracted element in which the error is included. The strategy states that this element is illegally modeled and invokes correction to explain the exact nature of the fault. Correction presents the model error using the presentation strategy and contrasts it to the meta model element that is violated. D.MODALITY ⇒ obl is added to the satellite in order to present the meta model element as something that has to be the case. Combining these two, we generate an explanation that tells the user what is illegal in the present model and what rule is violated. If the explanation is generated in the illustrative view — or an illustrative view can be assumed — the presentation strategy will also include a view of the illegal part of the conceptual model.
Table 12.5: The detection of a model error.
Imagine the situation in Table 12.5, in which process P1.2 is not given any output flows. The model does not satisfy the meta model element saying that every process must produce at least one flow, and the verification component sends the explanation request with the necessary background information to the explanation component. By means of the strategies model_error and correction, then, the deep explanation and the multimedia surface explanation in Figure 12.4 are generated.

Model Inspection

We will use two examples to show how explanations can be used at a model inspection stage: First, we explain the properties of an account, as it is modeled in Figure B.2. Secondly, we describe the behavior of the PLD diagram shown in Figure B.8. In addition, the explanation of process P3 in Figure 11.5 should give a good indication on how processes in PrM models are explained. As a start, we assume the following user question:

"What is an account?"
Figure 12.5: Describing an account.
Figure 12.6: Describing a PLD diagram.
With an appropriate view specification, the question leads to the request and explanation structure depicted in Figure 12.5. Here, the views BEHAVIORAL and COMPUTATIONAL are assumed hidden, which lets all(relationships(account)) include account's involvement in relationships with other entity classes. The attributes are described using the strategies properties and possessions. A more complicated question is:

"What does the PLD diagram for process P2.1 do?"

In this case, the explanation topic is a whole algorithm, and not just an isolated element of the model. The explanation structure gets quite complex, as Figure 12.6 clearly illustrates. Due to the decomposition strategy, all statements of the PLD diagram are included and their dynamic properties are briefly described. Although we have not included the whole structure, it should not be too difficult to see what the whole explanation would look like.

**Modeling Deliberations**

If modeling deliberations are recorded and associated with elements of the conceptual model, they can be presented as explanations to the user. The discourse strategy for doing this is justifications. There are two formulations of the justifications strategy, of which the first one can only be used in the INTERNAL view. Using this first formulation, the component justifies the inclusion of the element and evaluates its presence by invoking the second one. The second one, which is applicable in all views, just presents the model element and its underlying rationale or justification.

However, in PPP there are yet no deliberations modeled and recorded in the conceptual model, so the justifications strategy is more or less redundant in the current version of PPP. The use of the strategy, though, can easily be illustrated assuming that justifying texts can be associated with elements of the conceptual model. For example, if process P1.2 were given the following justification,

```
  P_OPERATORS
  HEAD
  AGENS

  justification(p1.2):
  TYPE
  HEAD
  GOAL
  RESTRICTOR

  POLARITY neg
  D_MODALITY perm
  withdraw
  customer

  HEAD exceed
  AGENS amount
  SEM.. balance
```

the request goal justifications(p1.2) in the EXTERNAL view results in the explanation in Figure 12.7.
Model Execution

Using the model execution indicated in Appendix B and Table 12.4, we can now generate history explanations and input justifications. Consider the following dialogue between user and execution environment:

system: "Error — available amount is 80."
user: "Why was my withdrawal rejected?"

The explanation request constructed has the discourse goal cause(withdrawal_rejection_{i5}). Technically speaking, the cause strategy is used to explain the creation of an instance of flow Withdrawal_rejection. With the assumption that the views INTERNAL and BEHAVIORAL are available, and not the views COMPUTATIONAL and RESTRICTIVE, we get the explanation structure shown in Figure 12.8. In this deep explanation, the first formulation of cause is utilized, and the two subgoals

background(withdrawal_rejection_{i5}) and
application(pld1.2.3_{i6}, generate(pld1.2.3_{i6}, withdrawal_rejection_{i5}))
Figure 12.8: The history explanation.
are posted. Pld1.2.3 is the send box that generates the flow. The first subgoal is attained by informing about the flow instance and showing a view displaying that flow. The second subgoal takes us to the application strategy, where it is decomposed into two new goals:

\[
\text{background(pld1.2.3) and}
\text{computation(pld1.2.3, generate(pld1.2.3, withdrawal_rejection))}.
\]

By using the computation strategy, the element generating the flow is paraphrased and the relevant precondition is described. Since pld1.2.3 does not have a precondition on its own, we follow succeed links backwards to the first element with a specified precondition and take that one. The precondition is included in the explanation, together with the values used in its evaluation. If the RESTRICTIVE view has been available, we would also justify these values from other rules and events.

Input justifications explain why the system needs a specific input from the user. After Withdrawal_rejection has been sent, the system asks the user to input a new withdrawal instead. The user can request an input justification by posting the following question:

\[
\text{user: "Why do you need New_withdrawal?"}
\]

The discourse strategy need is used to generate the justification. Depending on the view specification, the justification summarizes some ongoing activities or even go into the detailed computations leading to that input request. In Figure 12.9, we have assumed that the EXTERNAL and ILLUSTRATIVE views apply, but not the views INTERNAL and RESTRICTIVE.

In this case, the explanation is not very helpful in understanding why the original withdrawal was rejected. However, if the INTERNAL view were used, the text would also explain the computations leading to the request for New_withdrawal. The discourse goal cause(new_withdrawal) would be posted, and the result would be an additional text saying that New_withdrawal was requested because Withdrawal_transaction's Account_balance was less than Withdrawal_transaction's Amount.

12.4 Interface to Other Components

The explanation component depends on information from other parts of the CASE environment, but can only handle representations in EML. For the component to be integrated into PPP, one must include mechanisms that translate conceptual models and results from verifying and executing conceptual models to EML. Besides, meta model information must be made available to the component, as well as routines for generating views of the conceptual model.
Interface to Source Model

Currently, the PPP environment does not keep an explicit meta model of the PPP modeling language. The conceptual models are represented and stored as a collection of Prolog predicates, which are also used by the implemented code generators to produce executable program code. Syntactic properties of the modeling language are hard-coded in the editor and the verification component. When the models are executed, we are yet not able to build any trace documenting the execution, but this is expected to be possible quite soon [249].

In the explanation component, all elements of the source model are given unique identifiers. These are not known outside the component, but have to be assigned as representations are translated to EML. Internally in the explanation component, one has to store a table for mapping the identifiers outside the component onto our internal identifiers. This table we could call an *id conversion table*, and it would be used as a simple one-place function during model translation.
The explanation meta model has to be constructed manually by people familiar with the modeling language. No knowledge of the explanation component is necessary to do this. However, for each term used in the meta model, morphological information about the term have to be inserted into a simple lexicon used by the tactical component.

Translating the conceptual model to our EML representation, we need a general translator that has access to the representation scheme used both outside and inside the explanation component. The structure of such a translator is shown in Figure 12.10. The conceptual model, represented as Prolog predicates, is input to the translator, which exploits the correspondence between Prolog predicates and elements of the explanation meta model to create corresponding elements of the explanation conceptual model. Throughout the translation process, the id conversion table is consulted and modified. Since the explanation conceptual model is to contain basically the same information as the original one and the structures of the explanation meta model reflect those of the PPP language, this translation should not be that difficult to realize.

The execution trace is constructed as the conceptual model is exercised. In Appendix B, the trace is represented using a specially defined trace tuple from [249], but it could just as well have been represented in terms of instantiations of conceptual model elements. If we assume that this is the case, we can just extend the translation above to take traces and model instances in addition to conceptual models. Otherwise, we can use special translation functions that retrieve information from the traces represented [90].

The translator is only needed when representations outside the explanation component is to be used in explanation generation. It can therefore be implemented as an integral part of the complete explanation generation component, which means that the rest of the environment does not have to be substantially modified.
12.4. Interface to Other Components

Interface to Verification Component

As implemented in PPP, the editor and the verification component shares the same representation of the conceptual model. On the other hand, the verification rules themselves are hard-coded and error messages from verifying the model are today stored as canned texts and sentence templates.

The verification component sends information of two kinds to the explanation component, model errors and violated *a posteriori* rules. Both of them must be translated to *EML* for the explanations to be generated. Since model errors are present in the conceptual model, they must be representable in the language used to represent the whole conceptual model and we can then generate the corresponding *EML* representations using a translation algorithm similar to the one translating the conceptual model. Furthermore, being elements of the meta model, the verification rules are given in *EML* in the same manner as a description of the modeling language is made available to the component. The interface between explanation component and verification component, thus, is in complexity equal to the one between explanation component and information about modeling language and conceptual model.

Construction of Request and Extended Source Model

Our strategic component assumes an extended source model and a complete explanation request to be available. The extended source model includes the user knowledge submodel as well as annotations that describe the various model elements’ visibility or invisibility in predefined views. The request contains a view specification that specifies user preferences and explanation context.

To construct the requests and the extended source model, we need the following routines in the explanation component:

- A user modeling component must be connected to the strategic component. It should keep and modify the user knowledge submodel and the user preferences submodel on the basis of user’s behavior or requests. User modeling components of this kind are today found in many systems (see for example [135]).

- A routine for reporting the system’s status in terms of view specifications must be included. This routine is similar to the one administrating the user preferences submodel, except for that we are here analyzing system status instead of user preferences.

- A request construction routine is needed. When users request an explanation, the routine is responsible for creating the complete *EML* request that is sent to the deep generation process. The user preferences submodel and the context specification are added together to form the view specification, and a user knowledge strategy is determined. The request subject is received from the user or the verification component.
• The extended source model must be constructed from the source model, the user knowledge submodel, and the views defined in the environment. Basically, this is done by amalgamating the EML-represented source model and user knowledge submodel, and deciding the elements' associations with predefined views.

In Figure 12.11, the routines are illustrated. All the routines here can be hidden inside the explanation component. Two of them, however, can also be implemented outside the component, since there may be other components of the CASE environment that can take advantage of them. These are the user modeling component and the routine for constructing the extended source model. Both can be constructed and stored as part of the conceptual modeling component and can also be used by the view generation component to produce suitable views of the conceptual model.

Interface to Tactical Component

Within the PPP project, there are two works on textual surface generation. A simple inter-clause realization prototype has been constructed by Aune [8], whereas Johnsen has been dealing with intra-clause realization [126]. Our deep explanations are used directly by the inter-clause realization component, so there is no need for any translations or mappings here.

Realization of graphical explanation elements in PPP is discussed in [250]. As the view generation component in PPP is intended to be [215], the display commands can be sent directly to the component for realization. However, the explanation component and the
view generation component must share the same view definitions, which means that the views cannot be isolated to the explanation component alone.

12.5 Realization in Other Environments

PPP is a modeling language combining processes, entities, relationships, and simple block-structured algorithms. As noted in [87], all these language concepts are widely used in contemporary CASE environments. Showing that the component works for these kinds of concepts, thus, supports the claim that the explanation component can be adapted to other environments and other modeling languages as well.

In this section, we will briefly touch two other kinds of modeling concept and see how the component could be used for these formalisms. First, we examine a state-based language called Statecharts; then, we look into the rule formalism defined in TEMPORA. A full assessment of these two would lead too far here, but at least we will see how their distinct dynamic properties fit into our deep explanation generation approach.

Explanation Generation for Statecharts

Statecharts is a language used to describe behavioral aspects of information systems. It is based on state transition diagrams, but is somewhat extended to cope with hierarchical structures and more complicated system behaviors. Statecharts' underlying structures, Hi-graphs, are presented in [97], the language itself in [96], and the CASE environment using the language (STATEMATE) in [98, 99]. We will here not construct an explanation component for Statecharts, but rather indicate how our existing component could be introduced in environments using Statecharts.

Consider the Statecharts diagram in Figure 12.12, which illustrates some of the more characteristic features of the language. Blobs denote states, and in the figure state A is OR-decomposed into B and C, and B is AND-decomposed into D and E. OR decomposition means that all substates of a decomposed one must be active when the decomposed one is active. AND decomposition, in contrast, requires that only one substate be active at a time. Transitions between states are labeled as $\alpha L/\beta$, $\alpha$ is the event triggering the transition, $L$ is a logical expression that can involve other states, and $\beta$ is an internal event performed by the transition. Of course, there are also other features in the language, but the ones we have mentioned here are sufficient for our discussion, and the interested reader is referred to [96] for a more complete exposition of the Statecharts language.

In our explanation component, we would represent states in Statecharts as dynamic concepts. OR decomposition corresponds to the aggregation relation in DSM, AND decomposition to the generalization relation [222].

Basically, there would be two ways of representing the dynamic aspects of Statecharts
using EML and the elements of DSM:

- Only states are declared as dynamic concepts, and transitions are directly added as preconditions, triggers, terminators, postconditions, and acts to the states. This yields a very simple representation of the models, as indicated by the following EML references for modeling state D in Figure 12.12:

\[
\begin{align*}
\text{dynamic}_{-}\text{concept}(D) & \quad \text{act}(D,A1) \\
\text{precondition}(D,L1) & \quad \text{postcondition}(D,A2) \\
\text{trigger}(D,E1) & \quad \text{terminator}(D,E2)
\end{align*}
\]

There are two problems here. First, L2’s relationship to D is lost, since DSM does not allow terminating signals to have preconditions. Secondly, the fact that E1, L1, and A1 are attached to the same transition is not represented, and this is unacceptable if there are several transitions that can trigger the same state. So, in this case one must either modify DSM or assume that there is at most one transition leading to every state.

- Both states and transitions are declared as dynamic concepts. Provided that the transition with label E1 L1/A1 is given the id t1 and the other labeled transition is given the id t2, we would then get the following EML references when representing D and t1:

\[
\begin{align*}
\text{dynamic}_{-}\text{concept}(D) & \quad \text{dynamic}_{-}\text{concept}(t1) \\
\text{trigger}(D,t1) & \quad \text{trigger}(t1,E1) \\
\text{terminator}(D,t2) & \quad \text{precondition}(t1,L1) \\
 & \quad \text{cause}(t1,A1) \\
 & \quad \text{terminator}(A1,t1)
\end{align*}
\]

In this case, one need not modify DSM, and one would also be able to represent the more specialized Statecharts concepts like timeout and activities as duration and acts, respectively. The resulting EML model, however, would be substantially more complex than the first one.

Both ways will make the explanation strategies applicable without any modifications. As an illustration, assume the following (informal) explanation request:

\[
\begin{align*}
\text{request:} & \quad \text{structures}(D) \\
\text{view specification:} & \quad \text{behavioral} \Rightarrow \text{yes} \\
 & \quad \text{computational} \Rightarrow \text{yes} \\
 & \quad \text{illustrative} \Rightarrow \text{no} \\
 & \quad \text{internal} \Rightarrow \text{no} \\
\text{user knowledge strategy:} & \quad \text{none}
\end{align*}
\]
The strategies structures and dynamics are invoked to generate the deep explanation. As far as these strategies are concerned, it does not matter what modeling language is used in the environment: They are only working on EML references of the source model, and these are — as we have seen above — basically the same for Statecharts as for PPP. If only states are represented as dynamic concepts in our EML language, the request will lead to the deep explanation structure shown in Figure 12.13. Representing both states and transitions as dynamic concepts, we get the more compact, but less informative deep explanation structure in Figure 12.14.
Explanation Generation for TEMPORA Rules

In the ESPRIT project TEMPORA, a rule-based approach to information systems development is taken. The conceptual modeling language defined in TEMPORA has three components [236]: the ERT language, the Process Model language, and the External Rule language (ERL). We will not discuss the ERT language and the Process Model language here, since these languages are comparable to PhM and PrM in PPP from an explanation generation’s point of view. The External Rule language is rooted in temporal logic, and we will examine how the rules in ERL fit into our explanation component.

The general structure of ERL rules is

```
WHEN <trigger condition>
IF <precondition>
THEN <conclusion>,
```

of which the WHEN and IF parts are optional. <trigger condition> is a signal that triggers the rule, and <precondition> is a state operation or a state condition that is evaluated when the appropriate signal is present. If the precondition is found to be true, the actions in <conclusion> are performed. These rules are used to specify constraints on static ERT models, derivation rules, and action rules for dynamic parts of the TEMPORA conceptual models. The conditions and conclusions contain logical formulas and mathematical expressions.

In our explanation component we would represent each rule as a dynamic concept with associated triggers, preconditions, and postconditions. Consider the Process Model process and its associated ERL rule in Figure 12.15. The rule specifies the process logic and describes the relationship between input flows and output flows. Representing the model part in EML, we get the references

```
dynamic_concept(P1)   dynamic_concept(r1)
trigger(P1,A)          trigger(r1,A(x))
receive(P1,B)          precondition(r1,B(y))
generate(P1,C)        precondition(r1,C(z))
aggregation(P1,r1)     
```
The rule is given an id r1 and is related to its process using DSM’s aggregation relation. An explanation of r1 would be quite straight-forward to generate, and would — given the right features specified in the request — result in the deep explanation structure in Figure 12.16. The explanation strategies used are the same as have been used for PPP and Statecharts, and the explanation structure is not very different from what has been generated earlier. The rule formalism, it seems, fits well into the structural elements of DSM that help us construct the explanations.

However, the contents of the conditions and the conclusions of the rules are not analyzed and are just included as they are specified in the TEMPORA model. We do not simplify or structure the content of these expressions, but rather assume that the surface generator provide a satisfactory presentation or paraphrase of them. If the expressions are simple algebraic expressions, this may not be a problem, but since the rules here can involve complicated temporal formulas, it might be more difficult to make them understandable to unskilled users. Still, this is principally a surface generation problem, and we will not go more into this here.

12.6 Integration Requirements

With respect to the integration requirements from Chapter 7, the experiences from realizing the core for the PPP environment are as follows:

Generator applicability It is difficult to know how general the generic strategic component core is when it comes to modeling languages and conceptual models. What we have shown here, is the feasibility of using our explanation strategies on a process language (PrM), an ER-like language (PhM), a program design language (PLD), a state-based language (Statecharts), and a rule language (TEMPORA). We have also used the strategies on two different example models (Appendix A and Appendix B). Assessing the quality of these exploitations of explanation strategies, though, we need more testing with other kinds of language (like object-oriented languages) and other kinds of business area.
Generator Flexibility: Introducing the generic strategic component core in other CASE environments, we must provide a meta model of the environment's modeling language and modify the view generation component to accept display commands. No knowledge of explanation generation is needed for that. New explanation strategies can also be added at any time, but this requires some knowledge of how the explanation component works.

Isolation of Explanation Decisions: The deep generation process itself is isolated from other components of the CASE environment. Generating the appropriate views, however, the views defined in the explanation component must also be known and used by the view generation component. But having this sharing of views, we don't need to implement our own view generation routines, and we know that our views are consistent with views generated from other parts of the environment.

Tailoring of Explanations: This requirement will be discussed in detail in Chapter 13, using the seven standards of textuality as a measure for text quality and user-tailoring.

As compared to the general CASE architecture in Chapter 8, our integration of explanation facilities in PPP yields a slightly different architecture. The explanation meta model had to be constructed on the basis of what is implemented in the environment, since there is no explicit modeling meta model represented. The trace from the execution component must be directly translated to EML, as no trace information is centrally stored in PPP.
Chapter 13

Quality of Deep Explanation Generation Process

The purpose of this chapter is twofold: Most importantly, we will relate the explanations generated to the standards of textuality. In lack of a more concrete measure of text quality, we will use these standards to assess the communicative abilities of our explanations. Secondly, we will lay out a simple scenario that can shed some light on how the explanation component might be used in a real conceptual modeling process. The examples used in the chapter are all taken from the conceptual model in Appendix B and the explanation requests in Chapter 12.

13.1 A Conceptual Modeling Scenario

Let us assume that two actors are involved in the conceptual modeling process, an end-user and an analyst. The end-user is not familiar with the modeling language, but is at least expected to take part in the validation of conceptual models. The analyst knows the main structures of the language, though not necessarily all the details, and is responsible for constructing the models that are shown to the end-user. The end-user can be characterized by the (initial) user preferences submodel

\[
\text{end\_user} = \begin{bmatrix}
\text{INTERNAL} & \text{no} \\
\text{EXTERNAL} & \text{yes} \\
\text{RESTRICTIVE} & \text{no} \\
\text{COMPUTATIONAL} & \text{no} \\
\text{BEHAVIORAL} & \text{no}
\end{bmatrix}
\]

205
Figure 13.1: Simple description of entity classes.

This means that the end-user is not interested in deeper internal model properties or constraints on model structures, but rather in overall structures and overview descriptions. The analyst, on the other hand, may be given the following user preferences submodel:

\[
\text{analyst} = \begin{bmatrix}
\text{views} \\
\end{bmatrix},
\begin{bmatrix}
\text{ILLUSTRATIVE} & \text{no} \\
\text{INTERNAL} & \text{yes} \\
\text{RESTRICTIVE} & \text{yes} \\
\text{COMPUTATIONAL} & \text{yes} \\
\text{BEHAVIORAL} & \text{yes}
\end{bmatrix}
\]

The analyst will not need any illustrations and should concentrate on deeper model properties related to model dynamics and structural constraints. Naturally, these submodels may be modified later, but for our purposes here we can assume them to be valid.

In the model construction process, both actors might be asking questions about the modeling language. A question like

"What is an entity class?"
relationship(have, account):

\[
\begin{array}{l}
\text{HEAD have} \\
\text{POSITIONER customer} \\
\text{GOAL account} \\
\text{USER yes}
\end{array}
\]

Table 13.1: User knows that customers have accounts.

would be interpreted differently, then, due to their different user preferences submodels. If we assume the construction context to be

\[
context = \left[
\begin{array}{l}
\text{VIEWS ILLUSTRATIVE no} \\
\text{EXTERNAL yes}
\end{array}\right].
\]

the end-user will be given the explanation indicated in Figure 13.1, and the analyst gets the one in Figure 12.3. The end-user’s explanation is short and only sketches the entity class’s relationships to other language concepts. The explanation given to the analyst is more detailed and specify syntactic properties of entity classes.

Now, given that the PhM model in Figure B.2 is finished, the end-user can ask questions about it to control her own understanding. If the end-user submodel and the context specification are the same as above, the question

"What is an account?"

leads to the explanation structure shown in Figure 12.5. This explanation is rather extensive, and would be tiresome to read if the end-user is familiar to parts of it in advance. Let us then assume the end-user to know that customers have accounts, adding the USER attribute to the appropriate extended source model element as shown in Table 13.1. We can now specify in the request that only unknown elements are to be included, and we get the request and explanation structure depicted in Figure 13.2. The view specification is unchanged, though the explanation is reduced because the end-user already knows some of the properties of accounts.

At last, we can try to execute the PPP conceptual model from Appendix B. As in the previous chapter, the system rejects the original withdrawal and asks the user to specify a smaller amount to be withdrawn. There are at least two reasons for requesting an input justification in this situation:

**Guidance context** The user is uncertain about what is going on and does not know what she should do. The justification should then inform her about the activities going on in the system, without dwelling on any detailed functionality. This corresponds to a context specification, in which the EXTERNAL and the ILLUSTRATIVE views are available, whereas the INTERNAL view is hidden. The end-user role is consistent with this context, and consequently, to her question
"Why do you need New_withdrawal?"

the system responds with the explanation indicated in Figure 12.9.

**Debugging context** The user is validating the internal dynamics of the system and is interested in the computations underlying the input request. The context here would be that the views INTERNAL, COMPUTATIONAL, and BEHAVIORAL are available, and the view EXTERNAL is not available. The analyst’s view specification is consistent with this context, of course, since she has to check that the model works as it was intended to. So, in response to the analyst’s question

"Why do you need New_withdrawal?"
in this context, the system generates an explanation with the structure shown in Figure 13.3.

The two input justifications generated above differ in focus and detail. The one generated in the guidance context briefly describes the system’s ongoing activities and is meant to support the user in executing the rest of the model. The other one includes the detailed calculation that led to the input request and is expected to be used in validating the internal dynamics of the the model. Technically, the explanations are generated using two different formulations of need (see Appendix E).

### 13.2 Meeting the Standards of Textuality

The seven standards of textuality are **cohesion**, **coherence**, **intentionality**, **acceptability**, **informativity**, **situationality**, and **intertextuality**. Below, each of them is discussed with...
Cohesion This is the responsibility of the tactical component, since it concerns the actual wording of the explanation. However, the rhetorical relations included in the deep explanations can indicate the need for cohesive constructions in the final surface explanation. In Aune’s inter-clause realization component [8] (and other [116, 205, 213]), thus, each possible rhetorical relation is associated with a set of possible cohesive constructions that ensure the cohesion of the surface text.

For example, the relation dynamics is in the tactical component often realized as the adverb “dynamically”, which then precedes the relation’s satellite.

Coherence Coherence is achieved if we can guarantee that text units — at all levels — are rhetorically connected to form meaningful units at the higher level.

All explanations generated in our system must be coherent, as their deep explanation structures are hierarchical patterns of rhetorical relations. Each relation specifies how one text unit is related to another adjacent text unit.

The cost of this is that we are only allowed to define explanation strategies in which the relationship between nucleus and satellite can be characterized by a rhetorical relation. Actually, defining a schema instead of an operator, we are not restricted in this sense, but schemas are only recommended to be used at levels where conventions and not rhetorical relations determine text relationships (see [115, 205]).

In the deep explanation in Figure 13.3, for example, the hierarchical structure of rhetorical relations signals the coherence of the surface explanation.

Intentionality This standard is not very relevant here. The intention of all explanations is fixed, and that is to inform the reader about some property of the source model. In FG’s terminology, all clauses of our explanations are interpreted as declarative [67].

Acceptability An explanation is acceptable if its structure and content are tailored to the roles, interests, responsibilities, etc. of the reader — that is, if the explanation is relevant to her.

Explaining the concept of entity class above, we saw how the user preferences sub-model influenced the generated explanations. Since these submodels are supposed to represent exactly the user characteristics relevant to the acceptability of texts, we have here a mechanism for generating explanations acceptable to the intended reader.

However, having the appropriate mechanisms does not guarantee the acceptability alone. For our explanations to be acceptable (in this restricted sense), the user preferences submodel must be correct and the explanation strategies must be made visible or invisible in the appropriate views. This requires extensive testing, and the current generic strategic component core in Chapter 11 does definitely not claim to be optimal in this respect.

Informativity Informativity means that the text is tailored to the knowledge of the intended reader.
We have generated two different explanations of accounts (see Figure 12.5 and Figure 13.2). The user preferences submodels and the context specifications are the same, and the only difference in the explanation requests is the user knowledge strategy. When no user knowledge strategy was given, the generator did not distinguish between unknown and known model elements, and we got the quite large explanation in Figure 12.5. When the user knowledge strategy was set to no, only unknown elements were included and the resulting explanation was considerably smaller (Figure 13.2). Moreover, since the explanation strategies may override the user knowledge strategy indicated in the requests, parts of the explanations may be restricted to unknown elements and other parts to just known elements. In this way, we have a mechanism for tailoring the explanations to the reader’s level of expertise.

Still, for this mechanism to actually work, we need a correct user knowledge submodel, a satisfactory method for determining the user knowledge strategy to put into the request, and well-founded principles for when a strategy should change the user knowledge strategy for its subgoals. These are areas for further research, and we have in our component just shown how the process of user-tailoring can be achieved.

**Situationality** With the situationality of a text, we refer to how the text is tailored to a wider conversation setting.

In our component we use context specifications to represent the conversation setting. As we saw in connection with the input justifications, varying these specifications can have profound effects on the explanations generated. The context governs the use of explanation strategies just as the user preferences submodel. As a result, we are able to tailor the explanations to the particular context at hand.

In line with the standards of acceptability and informativity, we can here offer the proper mechanism, though we do not claim that the strategic component core in Chapter 11 is optimal what situationality is concerned. More testing is needed to find the best characterization of modeling contexts in terms of view specifications.

**Intertextuality** Intertextuality means that the explanations should — both in content and form — be adapted to other texts or messages used in the environment.

This standard is somewhat more difficult to discuss, since it depends on the user interface of the rest of the CASE environment, and that environment is not supposed to be fixed. Still, some brief remarks about language independence and conceptual model presentations should be appropriate.

In principle, any language could be used in the CASE environment, and the explanations generated must of course make use of the same language as the rest of the environment. This means that our EML representations must be suitable for representing clauses in arbitrary languages. Since EML-represented clauses can be translated to FG predications (see Appendix D), and the theory of FG is proven to have a certain typological adequacy (see [67]), there are good reasons for assuming this language independence.

In the environment, views of the conceptual model are used actively during the modeling process. When model illustrations are to be included in explanations, these graphical elements should be consistent with the views used in the rest of the environment. Also, the explanations must include references that makes it possible
to go back and consult the corresponding parts of the source model. Both these
requirements are met in our component. First, the illustrations are generated us-
ing the same view generation component as the rest of the environment. Secondly,
the explanations include identifiers of model elements, and the user can afterwards
use these identifiers to check the relevant parts of the model. For example, in Fig-
ure 13.3 the identifiers pl.2, New.withdrawal, pl.2.6, Withdrawal.transaction,
Account.balance, and Amount are all found in the underlying conceptual model
(though pl.2.6 is not directly visible).

The standards of textuality refer to the explanations' content and structure, and are there-
fore suitable means for discussing the quality of deep explanations. Another aspect of
deep explanations, though, is whether their linguistic content is of a quality that makes the
production of high-quality sentences possible. In [8, 126], it was concluded that surface
explanations can be produced on the basis of the given deep explanations. As long as
a complete tactical component is not integrated in our explanation component, however,
it is rather difficult to say if the linguistic information included in the deep explanations
are sufficient or should be supplemented with even more. Two facts seem to support the
chosen amount of linguistic information included in our deep explanations:

- A major hypothesis of FG is that the messages contain the necessary information for
  surface generation. Several works on surface generation support this view [10, 126,
  210], and since our EML representations can be translated to FG messages (with the
  addition of default values), they should also be sufficiently detailed. It might be the
  case that some of the default values should be generated instead of given as defaults,
  but this cannot be decided before a complete, integrated tactical component has been
  through the necessary testing.

- Other explanation generation systems using RST-based explanation strategies do
  not include more linguistic information in their deep explanations (see for example
  [116, 171, 174, 221]).

Generation of natural language is difficult and involves a number of decisions that tend to
interact. The works of Aune and Johnsen show that natural language realization is feasible
from our deep explanations, but it is still too early to draw any final conclusions about
how the deep explanations affect the syntactic and morphological quality of the generated
clauses in natural language. It seems to be difficult, on the one hand, to define textual
quality on a surface level, and on the other hand, to rigorously define what information
is needed to achieve that kind of quality [167]. Anyway, if it later turns out that more
linguistic information has to be put into the deep explanation, the attribute value structure
of EML representations should make these additions quite easy to make. New attributes
can be defined at any time, and the values of these can be assigned just as other values are
assigned in the deep generation process.
13.3 Some Problems and Limitations

Realizing our strategic component core, we met some problems related to the meta model and the explanation strategies that are not easily solved. To some extent, they can be handled by careful meta modeling, but there are also issues related to the more fundamental properties of the explanation component.

The Formulation of Source Model Elements

So far, we have assumed that text segments included in the source model can be used directly in the explanations. In practice, this may sometimes be a bit optimistic, since the person specifying the meta model and the conceptual model does not necessarily know how her text segments are used by the strategies. The explanation meta model is of course most crucial here, since the other submodels of the source model inherit terms represented in that model. However, new terms and phrases may also be introduced in the conceptual models, and these are equally vital to the explanation quality.

On the one hand, the text segments must be of the correct phrase type. For example, specifying the purpose of a process, the user must input a predicate phrase (like verify amount). If a nominal were indicated (like verification of amount), the discourse strategies activity and semantics would not work. The same observation has also been made in the Gist paraphraser [232].

On the other hand, the contents of these text segments must be chosen with care. They must be precise, of course, but also flexible enough to be useful in several types of explanation. To take an example, if accept amount were specified instead of verify amount, the input justification in Figure 13.3 would sound odd: A new withdrawal is needed because the system is accepting the amount.

Coverage of Meta Model

In this work, we have not required any extensions of the meta model (or the conceptual model). We formulated the explanation needs in Chapter 7 on the basis of what is traditionally included in these models, without elaborating on what should have been represented in the source model.

Looking back at the scenarios in Chapter 2, we realize that some of the explanations there are hard to generate in our system. Take this dialogue, for example:

user: "How is P2 Register customer activated?"

system: "P2 is activated if the user wants to register a new customer or a customer not already registered requests a loan."
Here, the explanation is given in terms of user goals and user abilities, but this information is not represented in the conceptual model. In our system, we could say that P3 is activated by either Customer name or No registration, and its purpose is to register a customer.

In general, our explanation component is very vulnerable to the amount of information put into the source model. Since executable conceptual models represent system functionality rather than user goals and user tasks, it is also difficult to generate any user abilities explanations. Still, explaining model elements in terms of how they are used to perform tasks or attain goals can be very effective on some occasions. The component can do this, though, if a task model is integrated with the executable conceptual model (following for example a methodology like USTM [159]), and the discourse strategies are modified to take advantage of this extra information. This means that the meta model must be extended so that other aspects of information systems can be modeled and analyzed.

Structures of Formulas

An unavoidable issue connected to deep generation systems is the granularity of the source model. Ideally, the content elements are undividable, so that they do not have any internal structures influencing the explanations. In our explanation component, though, there are two exceptions to this:

- Using the formula expression, we can represent information in some formal language that has internal structures. Usually, this is not a problem, since the whole formula can be included in the explanation. If we take the rule

  \[
  \text{IF (a AND b) or (c AND d) THEN e}
  \]

  as an example, however, we can see how the internal structure of the precondition can be used to improve the explanation generated. The precondition is represented as \(\text{formula}(\text{((a AND b) OR (c AND d))})\), and an instance of it can for example be \(\text{formula}(\text{(c \&\& d)})\), where \(c\) and \(d\) are instances of \(c\) and \(d\), respectively. Explaining why event \(e_i\) (instance of \(e\)) happened, one should then list the values \(c\) and \(d\), and relate them to the precondition of the rule. The whole precondition could of course be listed, but a more compact explanation would just say that \(e_i\) happened because \(c \&\& d\) — that is, only the relevant subset of the precondition formula is included. In [90], it is shown how an explanation is constructed with these optimization strategies in mind, but the proper generation mechanism is still not implemented. In general, such a mechanism can be quite complex, since it means that the explanation component must know the structures of any formal language that can be used.

- In PPP, as in many other conceptual modeling languages, the actual operations on databases are treated as undividable units. Explaining why a particular query failed, for example, is not possible, since the component is not able to analyze the query's structures.
This first problem can be solved by restricting formulas to only basic mathematical expressions, implementing for example logical connectives as rhetorical relations. The other problem can be handled if database operations (and results) are specified using an extended conceptual modeling language (like Yang's SDL [254]). In that case, one may introduce rhetorical relations and discourse strategies corresponding to structural terms in database operations.

**Complex Model Behaviors**

The discourse strategies generating history explanations and input justifications are still rather restricted. They work for simple models, but are clearly inadequate for more complex structures of dynamic elements.

Consider the PLD diagram in Figure 13.4, which is based on the PLD diagram in Figure B.7 (the shaded areas in Figure 13.4 are shared by both models). In this new model, the nationality is checked, so that the error message is given in either Norwegian or English. Also, after an acceptable amount of withdrawal has been input, a confirmation of requested amount is sent to the customer. Let us now assume a model execution, in which nationality is “nor” and account_balance is less than amount. When the withdrawal is rejected, the user may post the following history question:
"Why was Withdrawal_rejection sent?"

Using our current discourse strategies, the component would say that Withdrawal_rejection is sent because customer's nationality is "nor". The component is not able to distinguish between the two selection constructions, and just picks the closest one. Explaining the withdrawal, it should have used the fact that account_balance is less than amount — that nationality is "nor" just explains why the error message was given in Norwegian.

Also, if the user afterwards asks the question

"Why do you need New_withdrawal?"

the system will faultily say that the input is needed because nationality is "nor". One could try to mend this by instead referring to how the input will be used (as is done in input justifications in expert systems [203]), but this will not always be meaningful either. In this case, new_amount is needed to calculate amount, which in turn is needed to send a confirmation to the user. An input justification like that is not very informative, although one could provide additional discourse strategies so that the user could continue her questions and see how amount is used later also.

These two questions call for a deeper semantic analysis of dynamic models. The discourse strategies need more advanced precondition predicates that can check complex semantic relationships of the models. However, it will still be difficult to know which selection construction to use in the history explanation above. The first one determines that Withdrawal_rejection should be sent, the second one how this rejection message should be presented. So, in some sense, both are relevant, although they explain different aspects of the rejection. One solution here is to record explicitly in the conceptual model the precise semantic relationships between dynamic model elements, introducing for example affect_content and affect_expression links from causing constructions to affected elements. With the definitions of some new discourse strategies, this could work, though it is more demanding for the people constructing the models. Another solution is to restrict the way these dynamic models are constructed. For example, since the nationality test does not determine whether Withdrawal_rejection should be sent, one could require that the sending boxes be outside the scopes of the test's alternatives. This, however, could lead the rather awkward model constraints, which are not understood unless one knows how the explanation component works.

Rhetorical Relations and Domain Structures

With an explanation component like this, there are several kinds of representation and theoretical foundation that are related and have to interact with each other. The explanations generated appear to be directly related to the conceptual models, though the actual relationship is much more complicated, as shown in Figure 13.5. There are two representations of conceptual models involved, the environment's conceptual model and the
13.3. Some Problems and Limitations

![Diagram showing the relationship between conceptual models and explanations.]

Figure 13.5: The relationship between conceptual models and explanations.

An explanation conceptual model, and these are not identical. Moreover, a third model, the domain structure model (DSM), is used as an interface between explanation conceptual model and discourse strategies, and above all this there are theoretical aspects related to conceptual modeling languages, natural languages, and discourse structures. Making sure that the correct explanation — relative to some aspect of the conceptual model — is generated, we face three fundamental problems in our current component:

- The explanation conceptual model and the conceptual model are not identical, but the explanation conceptual model must still serve as a basis for explaining properties of the conceptual model. The correct semantic relationship between these two models must be established, though this requires that linguistic notions from the explanation conceptual model are correctly coordinated with conceptual ones from the conceptual model. Today, this depends on user's ability to manually formulate accurate and matching linguistic expressions, and there is no support from the environment in doing this.

- Since the strategies are formulated in terms of DSM elements rather than elements of the explanation conceptual model, it is paramount that the references attached to the explanation conceptual model are correct. This is problematic to verify, as neither the explanation conceptual model nor DSM is yet given a formal semantic definition. As the component is now, it is the responsibility of the meta modeler to relate the two models and specify appropriate references. No formal checking is available here.

- Discourse strategies define applications of rhetorical relations on elements of the explanation conceptual model. These strategies are formulated independently of any deeper semantic relationship between rhetorical relations and DSM elements, which ease the introduction of new specialized strategies in the component. Unfortunately, this lack of formal foundation makes it unfeasible to verify the appropriateness of a proposed strategy. We are therefore not able to check that a particular strategy is generating a valid explanation — or the best explanation — of some phenomenon.
These problems can be attacked from different angles. First, we could simplify the component by discarding the explanation conceptual model, DSM, or both. This would yield a more simple relationship between conceptual model and structures of rhetorical relations, though the result would be a less powerful component. If there is no explanation conceptual model, we just defer to the tactical component the problem of describing conceptual model elements in natural language. And an explanation component without DSM — or something corresponding to DSM — would not possess the desired flexibility with respect to modeling languages and meta levels.

Another approach would be to provide a formal semantics for the explanation conceptual model. This would help us to relate natural language expressions in the explanation conceptual model to elements of the conceptual model, thereby ensuring the appropriate relationship between the two. Even though it is not very easy to formalize natural language as a conceptual modeling language (see for example [14, 207]), the gains from doing this could be considerable. The CASE environment would be able to check that the explanation meta model constructed and the linguistic expressions introduced at the conceptual model level are consistent with other representations of the modeling environment. Also, this could open for a more formal and verifiable relationship between explanation conceptual model and strategies for explanation generation. Dignum’s Conceptual Prototype Language (CPL) [63], for example, might be a promising candidate for such a formalization, since it is rooted in the FG formalism.

DSM could also be given a firm semantic basis. With a formal explanation conceptual model and a formal DSM, the environment could run consistency checks on the two models, though this is still not sufficient for automatically checking the quality of discourse strategies. In addition, a deeper semantics specifying the interpretations and possible applications of rhetorical relations would be useful. This semantics would prevent meaningless strategies from being formulated, since strategies have to be consistent with the underlying semantics of rhetorical relations. An interesting approach here would also be to define rhetorical relations completely in terms of information system structures and use these definitions instead of DSM. Some suggestions along these lines can for example be found in [100]. When specifying the explanation meta model, then, we would add references to rhetorical relations instead of to elements of a generalized meta model (DSM). However, the exact nature of these relations are not well understood yet [161], so this would be a very difficult task to do.
Chapter 14

Related Explanation Generation Systems

So far, we have evaluated the explanation component with respect to the requirements from Chapter 7. In this chapter, we will see how other explanation generation systems relate to the same requirements. Afterwards, the internal structures of these systems are compared to what is found in our component.

14.1 Existing Explanation Generation Systems

Interesting explanation generation systems are today found in tools for information systems engineering, expert systems, tutorial systems, and help systems. There are yet no commercially available CASE environments with explanation facilities. However, with the recognition of the importance of validating conceptual models, there have been attempts in research communities to generate natural language texts from conceptual models. We here briefly describe the generation components in MOLOC, ALECSI, Gist-based environments, and Caravela. In addition, we discuss some representative examples from expert systems, tutorial systems, and help systems.

MOLOC

An explanation generation component for the modeling tool MOLOC [125] is being implemented by Dalianis [55]. The system explains structures of a conceptual model and is intended as a validation technique for the development of information systems. Although the component is under development, it seems likely to support at least structure descriptions and behavior descriptions. Explanations are basically theoretical, but instances of the
static parts of the model can be included as illustrations. System behavior explanations are not supported at all, and the explanations cannot include graphical elements or describe the modeling language's terminology.

The system makes use of discourse strategies like the one in Table 6.2, but these strategies are based on Hobbs's relations, and not RST relations. A method for user-tailoring explanations is suggested [54], but is only partly implemented. Basically, this method means that each conceptual model element is associated with user's knowledge of it, which can be at one of three levels (no knowledge, novice knowledge, or expert knowledge). The level of knowledge decides the content of explanations.

The explanation component is only applicable for the MOLOC conceptual modeling language, as the discourse strategies refer directly to concepts of that language. It cannot be introduced in other environments without major modifications of the strategies.

ALECSI

ALECSI is an expert system for requirements engineering [36], in which a conceptual model of the future information system is built to assess the requirements to the system. As a validation technique, a paraphraser is included to present parts of the conceptual model in natural language [202]. The paraphrases correspond to behavior descriptions and structure descriptions, and they are all textual. Terminological explanations are not offered, and model instances or execution traces are not available for explanation.

The discourse strategies used in ALECSI are single-sentence schemas, which are instantiated to form deep explanations\(^1\). The schemas are hierarchically arranged, and can include both linguistic notions (like sentence) and rhetorical relations (like circumstantial proposition). There is no coordination of sentences to form paragraphs of text, and no user-tailoring or context-tailoring of sentences.

ALECSI's discourse schemas work directly on the structures of semantic networks. They cannot be used on other modeling languages or in other CASE environments without substantial modifications and extensions.

Gist-Based Environments

The modeling language Gist [15] has been used as an underlying formalism for a series of systems within the field of information systems engineering. There are no real explanation generation components for Gist — only paraphrasers — but since Gist has been so central in the work of formalizing the development of information systems, we will still include it here. The explanation types covered are system theory explanations and history expla-

\(^1\)Our terminology deviates somewhat from the one used by Rolland and Proix [202]. They call the deep explanation a **surface structure**, whereas their **deep structure** is comparable to the result of the initial content selection process.
nations describing symbolic executions of Gist models. The explanations cannot include graphical elements or describe terms used in the modeling language.

The discourse strategies here are not explicitly represented, and there are no means for ensuring the coherence of the texts generated. The deep explanations are just sequences of attribute value structures. System theory explanations are not tailored and include all the information available in the Gist model [232]. History explanations try to meet the standards of textuality in the sense that some details are omitted from the explanations, proofs are summarized, referring terms are used for objects, repeating information is avoided, and "surprising" events are highlighted [233]. But there is no user model present, so the tailoring above represents the generator’s implicit and fixed assumptions about the user and the explanation context.

The discourse strategies are specifically designed for Gist and cannot be used directly in other environments. Since they are hard-coded in the environment, it is also quite awkward to modify them for other modeling languages or other environments. The paraphraser has later been employed in the KBSha project (see [129]), but the representation language used there is just a modification of the Gist language.

Caravela Workbench

A paraphraser that provides on-line documentation of conceptual models is implemented in the Caravela workbench [34, 124]. The component is used to document and validate models, but also to generate simple user manuals of future computer systems. The texts generated can be classified as either structure descriptions or explanation elaborations. There are no system behavior explanations available, no graphical elements can be included, and neither terms nor instances/traces can be explained.

The generation component is realized as a hypertext system, in which the user can initiate the paraphrasing of model elements by selecting referred elements in earlier paraphrases or inputting new elements. There is no user-tailoring or context-tailoring of explanations — they simply list all model information directly related to the given element using simple sentence templates. No discourse strategies are needed, since the user herself can find larger bodies of texts by navigating through the hypertext system.

The principle of using hypertext systems and sentence templates can easily be used on other languages and for other CASE environments. However, the templates themselves are very specific to the language used in Caravela (Infolog), and cannot be reused on other languages. Also, the templates are fixed, which makes it very difficult to later include user-tailoring mechanisms or flexible strategies for generating paragraphs of coherent text.
system: Simple-variables are storage locations named by symbols. This concept can be generalized to a storage location that can be named by any accessor function. These storage locations are called generalized-variables.

user: Huh?

system: For example, the car and cdr of a cons are generalized-variables, named by the accessor functions CAR and CDR. Other examples are an element of an array or a component of a structure.

Table 14.1: In EES the user response “Huh?” initiates an elaboration explanation of what the system thinks was not understood in the last explanation (adapted from [174]).

Expert Systems

Expert systems concentrate on explanations of system behavior and system theory. Within system behavior explanations, input justifications and history explanations have been given priority, but hypothetical and system abilities explanations have received an increased attention the last few years (see for example [1, 243]). Terminological explanations are available in some systems (e.g. EES [177] and XPLAIN [234]), as are explanations that correct user’s beliefs or compare source model elements with each other (e.g. RATIONALE [1] and LUC [243]). However, the source model in expert systems is a fixed rule base — possibly supplemented with fixed models of domain properties or generic structures of rule bases — so there is no need to support the construction process. Consequently, explanations about language syntax, verification results, or modeling deliberations are not supported.

As an example, take the Explainable Expert System (EES), which has an explanation component quite similar to the one we have constructed here. Its discourse strategies are RST-based plan operators (see Table 6.3), and the deep explanations constructed are hierarchical structures of rhetorical relations, instantiated goals, and content references (see Table 5.8). A user model specifying user’s goals is included and referred to by the strategies. Since the rule base is flat and structurally very simple, the discourse strategies are also simple and only a very restricted form of user-tailoring and context-tailoring is possible. Due to the textual nature of rule bases, it is also impossible to supplement the explanations with graphical elements. However, an underlying model of generic principles is included to provide more abstract high-level explanations, and by means of a simple context mechanism, EES is able to interpret follow-up questions relative to previously generated explanations (see Table 14.1).

EES’s explanation component can be used in all expert systems developed within the EES framework. Its discourse strategies, however, are far too simple to cope with the complexities of formal conceptual modeling languages. Abstraction mechanisms, the diversity of modeling concepts, and the intricate relationships between these concepts are all problematic for strategies intended for rule-based systems.
14.1. Existing Explanation Generation Systems

Tutorial Systems

In tutorial systems there are mostly system theory explanations. Most tutorial systems emphasize the structural properties of domain entities (e.g. TEXT [168]), but in EDGE [38] and RESEARCHER [189] behavior descriptions are also available. COMET [166], EPICURE [51] and MDC [163] generate user abilities explanations, telling the user what she should do to perform some task. Tutorial systems are usually textually oriented generation systems, but in COMET, EDGE and MDC graphical elements are also included (see Figure 5.2 and Figure 5.9).

EDGE (and also MDC) has discourse strategies formulated as plan operators. The operators are not based on any theory of textual coherence (like RST), so there is no guarantee that the resulting deep explanation be coherent. A user model is used to tailor the explanations to user's level of knowledge, and this model is associated with plan operators rather than source model elements.

EDGE's discourse strategies are not only restricted to the formalism used to represent the source model; they are also formulated with a specific domain in mind. This makes them very specialized, and quite difficult to reuse in other domains or on other languages.

Help Systems

User abilities explanations are the focus of most on-line help systems. Simple systems can only generate functionality explanations (e.g. [91]), whereas more advanced help systems will also offer enablement explanations and orientation explanations (e.g. Eurohelp [101]). Some system behavior explanations may be available — like state clarifications and result interpretations in Eurohelp — but in general this class has not been emphasized in help systems. System theory explanations, however, are found in some way or another in most of these systems. The explanations are theoretical, although examples are sometimes included as illustrations (e.g. examples of UNIX commands in [91]). They are comparative or centered, and the medium tends to be pure text.

Discourse schemas are used to generate deep explanations here. User-tailoring is achieved by choosing schemas on the basis of user class or user knowledge (e.g. Eurohelp and TAILOR [189]). In ROMPER [165], a user-dependent viewing mechanism is used to restrict what can be included in the explanations.

The discourse schemas are defined in terms of concepts of the underlying source model (see for example Table 6.1 and Figure 6.3), so they cannot be applied on other representation languages.
<table>
<thead>
<tr>
<th>explanation</th>
<th>MOLOC</th>
<th>ALECSI</th>
<th>Gist</th>
<th>Caravela</th>
<th>EES</th>
<th>EDGE</th>
<th>Eurohelp</th>
</tr>
</thead>
<tbody>
<tr>
<td>language semantics</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>language syntax</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>verification checks</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>model inspection</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>deliberations</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>model execution</td>
<td>no</td>
<td>no</td>
<td>some</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 14.2: Explanation types supported in various systems.

14.2 Deep Generation Principles

The strategies and techniques used in our strategic component core were presented in Chapter 9. Now, we will recapitulate the main principles and relate them to what is found in natural language generation systems in general.

First of all, the idea of generating deep explanations using a top-down planning approach is found in several systems (e.g. [38, 188, 163, 205]). Also, mechanisms for avoiding repeating information are common (e.g. [131]), and we will not go more into these here. There are differences, though, related to

- the formulation of explanation strategies,
- the tailoring of explanations,
- the inclusion of graphical elements, and
- the generality of the component,

and we will discuss each of these briefly below.

Formulation of Explanation Strategies

Explanation strategies based on Rhetorical Structure Theory is yet not found in information systems engineering, but have been tried out in expert systems [188], tutorial systems [116, 146, 205], and dialogue systems [72]. In these systems, there are plan operators specifying discourse goals, subgoals (nucleus and satellite), and various kinds of preconditions for using the operators. There are two main differences between our explanation strategies and those already used in generation systems:
• Our explanation strategies are not restricted to plan operators. Also traditional explanation schemas can be encoded, in which case only the nucleus specify elements to include in the explanation.

• In our system, we distinguish between the discourse goal and the rhetorical relation associated with a strategy. As seen from for example Figure 6.3, most other systems do not do that, and the rhetorical relation is also used as the discourse goal. This makes it difficult to decide when a particular strategy should be included as a substrategy of another one, since the strategy’s discourse goal characterizes the textual relationship between its subgoals rather than its content. To take an example, the motivation operator in Table 6.3 does not only provide the means for the act. In EES, they have tried to solve the problem by introducing an additional type of strategy whose discourse goals are communicative effects. These goals characterize the strategy’s content, and are used instead of strategies based on rhetorical relations at higher levels of the explanation generation process. The plan operators suggested by Hovy, though, has the same separation of rhetorical relation and content description as we have used in our system (see Table 6.4).

One could perhaps argue that these two differences imply that our strategies are not based on Rhetorical Structure Theory after all. Though, two facts distinguish our strategies from other non-RST-based strategies: (1) Even though schemas can be defined, these strategies are only to be used at very high levels where conventions rather than textual relationships decide coherence [72, 115, 205]. (2) Unlike the operators in Dalianis’s system, which also have content-describing discourse goals, our operators must also have rhetorical relations as well, and these are put into the deep explanation and later used in the surface generation process.

Tailoring of Explanations

Our user knowledge submodel corresponds to the overlay models found in help systems like Eurohelp. The user preferences submodel is developed from the viewing mechanism in ROMPER, though we allow a view to be specified as a combination of more simple views. Also, our views do not only decide which source model elements are relevant — they also decide which discourse strategies can be used. Our integration of context specifications and user preferences in not found in other generation systems.

Dalianis’s user model corresponds to our user knowledge submodel combined with a user preferences submodel consisting of only one defined view (expert or non-expert). Several views cannot be introduced in Dalianis’s system, and it is not possible to encode context specifications in the same tailoring mechanism. However, his view specification can differ from one conceptual model element to the other, which is not possible for our component.

In systems like EDGE and MDC, tests of user’s knowledge are included in the preconditions of the strategies. In our component, this is more restricted and user knowledge tests are only done when content elements are to be included in the explanations. User’s knowledge influences the explanation structure directly only when it prevents content elements
from being included so that other discourse strategies have to be used instead. Though, user’s preferences can also be interpreted as some stereotypical user knowledge submodel (e.g. an expert view means that all source model elements are known), and in that case we can include tests of user’s knowledge in the strategies.

Inclusion of Graphical Elements

In our system, visual acts are integrated into the explanation strategies as subgoals. This is also done in Maybury’s component [163] and in EDGE [38], but there is one important difference in the way this is done: In these two systems, there are two kinds of visual acts, one type for displaying an object and one for pointing out parts of a displayed object. In our system, we have only acts for displaying objects. However, these acts are modified by a view specification, so that the displayed picture is really a view of the specified object. In this way, we can focus on particular aspects or properties of an object without highlighting parts of a larger, more complex picture. Additionally, our approach makes it possible to use routines from the view generation component, and we do not have to add special routines for the explanation component.

In COMET [166], features specifying the explanation medium are added to the elements of the deep explanation instead of using visual acts. This could also have been done in our component, and the results would not have been very different. Still, as opposed to the elements of COMET’s source model, not all elements of our source model have independent graphical presentations. We would have to check properties of the elements before including graphical features, and it is then more practical to implement the whole thing as a separate explanation features strategy (the presentation strategy).

Generality of Component

In all generation systems available today, explanation strategies refer directly to concepts of the representation language. In EES, for example, the representation language is given by the methods used in an expert system, and the strategies refer to concepts like goal, step, and instance-of (see Table 6.3). This means that the strategies are restricted to systems using that particular representation language, though they can of course be used for all programs specified in the language. The same is the case for Daliani's system for conceptual modeling [55], where concepts like entity type and attribute are referred to directly in the strategies.

In our system, the strategies refer to concepts of a generalized meta model, the domain structure model. This model is assumed to capture the main structural concepts of graphical conceptual modeling languages for information systems engineering. As shown in Chapter 12, we are able to use the same strategies for process languages, state-based languages, algorithmic languages, and rule-based languages. And in Chapter 11, we used the same strategies for describing properties of both the meta model and the conceptual model.
14.3 Internal Formalism

Representing explanation strategies as predefined attribute value structures is now common in generation systems. Also, generating deep explanations in a top-down approach where discourse goals are unified and decomposed, and where backtracking is initiated when generation fails, is now used in several systems. For example, EES [188], EDGE [38], MOLOC [55] and Map Display System [163] all rest on this general approach to explanation generation.

Our use of general attribute-value structures, though, has opened for a more integrated and flexible representation of strategies and generation products. In particular, the following should be noted:

- The subgoals of our explanation strategies can involve any combination of AND, OR and XOR operators. Instead of specifying several formulations of the same strategy, then, one can use these operators and get away with just one formulation.

Constructing compact explanation strategies has also been tried earlier, using optional satellites in EES [174] or Hovy’s growth points [116]. In Table 14.3 and 14.4, we have shown how the optional satellite of strategy BEL from Table 6.3 and the growth points of strategy SEQUENCE from Table 6.4 are encoded in our formalism, respectively. Our AND structures, OR structures, and XOR structures make it quite easy to represent the same strategies, but we can also represent other structures of subgoals.
• View specifications and user knowledge strategies are — unless overridden by sub-strategies — inherited from one strategy to its sub-strategies. In Cawsey’s and Maybury’s systems, the user model is kept separate from the instantiated strategies and the strategies’ preconditions include predicates checking properties of these models. Including view specifications and user knowledge strategies in the instantiated explanation strategies has two advantages:

1. If the user specifications (view specifications and user knowledge strategies) are incomplete, a strategy can assume specific properties, and the inheritance mechanism will make sure that these assumptions will be enforced in all strategies invoked (directly or indirectly) from that one.

2. If the user specifications are irrelevant for parts of the explanation, a strategy can suppress them, and the specifications will also be suppressed in all strategies invoked (directly or indirectly) from that one.

In Cawsey’s and Maybury’s systems, a strategy can also assume or suppress user specifications using the predicates in precondition fields, but one cannot ensure that new strategies invoked (directly or indirectly) from the strategy make the same assumptions or suppressions. The user model cannot change from one part of the explanation to the other.

In systems like TEXT [168] or ROMPER [164, 165], a viewing mechanism resembling ours is used to separate out relevant parts of the source model, but this is done prior to the structuring phase, and the same view must apply to all parts of the explanation.

• All information relevant to the deep generation process is represented in EML. In fact, also information related to the surface generation process is representable in this formalism [8, 126]. As a result, it is straightforward to integrate various kinds of knowledge, for example adding linguistic information to the deep explanation as soon as they are determined. Since both the deep explanation and the textual part of the surface explanation can be represented in EML, it will also be easier to integrate deep generation and surface generation, if that later turns out to be desirable.

In most systems, there is a clear separation of deep representations and surface representations, which complicates the interface between deep generation and surface generation (e.g. [168, 165, 188, 116, 37]).

All in all, these additions have made it possible to integrate various kinds of knowledge and find very efficient representations of explanation strategies. The cost, however, is a rather complicated unification algorithm, which slows down the whole explanation generation process. As noted by [171], unification-based approaches to natural language generation is flexible and powerful, but also time-consuming and may require substantial memory resources. This, however, has not been an issue at this stage of the work — we have been focused on the feasibility of constructing an explanation component satisfying the requirements from Chapter 7. When the conceptual issues of the explanation component are all sorted out, though, the efficiency and effectiveness of the implementation should of course be addressed.
Part V

Extensions and Conclusions

This last part concludes the thesis and suggests some directions for further work.

Chapter 15 Here, we discuss how our strategic component can be supplemented by other components and routines to form a complete integrated explanation component.

Chapter 16 The major achievements of the thesis are summed up and some guidelines for further work are provided.
Chapter 15

Towards a Complete Integrated Explanation Component

In this thesis, we have worked out a formalism for deep explanation generation in conceptual modeling environments. We have also presented a general CASE architecture and a generic deep generation component, but these products are of a more secondary nature to the proposed formalism. A complete explanation component integrated into CASE environments, however, is much more than a simple formalism for generating deep explanations. In this chapter, we recapitulate some of the results from the realization in PPP and briefly describe what has to be done in order to construct a complete integrated component.

15.1 Improving Generic Strategic Component Core

The domain structure model and the explanation strategies from Chapter 11 gave the strategic component the required generality and made it possible to respond to the seven standards of textuality. However, the model and the strategies are both still at an experimental stage, and they can certainly be improved through testing with other modeling languages and other CASE environments. This also applies to the set of views defined in the system, which may be too small to differentiate between all the desired explanations.

The EML formalism itself and the principle for generating deep explanations are expected to be reasonably stable. Also, the idea of using a domain structure model to interface explanation strategies and source model should be suitable for other realization environments and should not require any changes. Though, for some very simple domains, it might be possible to discard the domain structure model and relate source model elements directly
15.2 Extending Meta Model

The partial PPP explanation meta model used in Chapter 12 is constructed in the tradition of CASE meta models. Its focus is on syntactic properties, and there is little information about semantic properties and what justifies the various model elements. If the meta model were extended, additional explanations could be generated and made available to the users. The explanation strategies can already handle some extensions (explaining semantic properties by means of denotation and extension elements, and justifying elements by means of justification and rationale elements), but a more fundamental revision of the meta model is likely to impose changes to the domain structure model and the explanation strategies.

It must be added, though, that an extensive meta model neither is nor should be necessary for the explanation component to work.

15.3 Adding a Tactical Component

So far, the explanation component lacks an appropriate tactical component. Some works have been done in this area (see Section 12.4), though these are far from being finished, and there are still some problems with the approach taken in these works.

In principle, there seems to be two strategies for adding a tactical component to our system:

- One can integrate and build on the works of Aune [8] and Johnsen [126]. Their systems are based on EML representations, but rely on well-established results within RST and FG. As reported in various publications, both RST and FG are well suited for this kind of surface generation (for RST see [52, 115, 116, 205], for FG see [10, 139, 210]).

- A quicker solution would be to include some suitable surface generator that is already available. This would not be as difficult as it sounds, since many existing surface generators are general and require the same kind of attribute value structures as is encoded in EML. Using Penman [117] as a surface generator, for example, one would have to rename many of the attribute names defined in Section 10.2, but the overall way of representing clauses in nested functional structures would be the same. What feature-based generators are concerned, the explanation structures would be basically the same no matter the particular formalism used in the generator.

If the first strategy is chosen, the surface generator might be quite small, since only aspects related to the realm of conceptual modeling need to be considered. The second strategy
would yield a larger surface generator, but it would be a general one that could also be used for other natural language generation tasks.

15.4 Additional Routines in Explanation Component

As discussed in Section 12.4, a complete explanation component is more than simply a strategic component followed by a tactical component. Recapitulating the main points from that section, we note that the following routines are also needed in the explanation component:

**Model translator** The translator is responsible for building the EML source model from representations outside the component.

**User modeling component** This component keeps and updates the user knowledge sub-model and the user preferences submodel, which are both assumed to be dynamic and possibly individual.

**System status reporter** The routine reports the status of the CASE environment in terms of relevant and irrelevant views.

**User input interface** This interface is to accept explanation requests from the users and construct the corresponding complete EML requests.

**Source model adder** We need a routine that can add information about views and user’s knowledge to the elements of the source model, thereby constructing the extended source model.

Moreover, what we have constructed in Chapter 11 is the core of a strategic component. We have deliberately avoided those parts of a strategic component core that may be influenced by decisions made in the tactical component (e.g. optimization of explanation structures). Actually, it is possible to use a core as the complete strategic component alone, and this is done today in many systems (e.g. [38, 131, 163, 168, 174]). But this would require a powerful surface generator that is able to do all optimizations on its own and that can realize all possible deep explanations from the strategic component core. If this cannot be expected, one must consider extending the core with additional routines that co-operate with the tactical component in some way or another. The interface between strategic component and tactical component would then be more complicated, though this is the approach advocated in many recent works in natural language generation (e.g. [114]).
Chapter 15. Towards a Complete Integrated Explanation Component
Chapter 16

Conclusions

This chapter sums up the major achievements of the thesis and points out some directions for further work.

16.1 Major Achievements

In this thesis, we have investigated the potential of explanation generation in CASE environments. The introduction of explanation facilities has been motivated, and we have developed a representation formalism and some generation principles that make this component feasible. Going into some more detail, the major results and achievements of the work are as follows:

- We have discussed how various kinds of explanations can support the construction and validation of conceptual models. In particular, we have shown their usefulness in explaining syntactic and semantic aspects of modeling languages, results from verification checks, aspects of conceptual models, modeling deliberations, and behaviors from model executions. This is not to say, however, that explanations cannot be useful for other parts of the conceptual modeling process as well.

- We have presented a general CASE architecture that integrates explanation facilities in conceptual modeling (see Chapter 8). In this architecture, explanations can be requested by the users or the verification component, and they can make use of traces from the execution component and views generated by the view generation component. The whole architecture is an extension of a CASE architecture suggested in [250], which only deals with model validation.

- Principles for generating user-tailored and context-tailored deep explanations are suggested. The principles are used to explain properties of source models comprising meta models, conceptual models, execution traces, and conceptual model instances in general. User's knowledge is recorded as an overlay model to the source
model, whereas her preferences and the context are defined as combinations of views of that model. RST-based discourse strategies use all this information to build a hierarchical deep explanation, where terminals are text segments of the final explanation and non-terminals describe the rhetorical relationships between parts of the explanation. Some of these strategies have already been published in [90].

• Our principles for deep explanation generation makes it possible to construct explanation components that can work on several modeling languages and in several CASE environments. This is achieved by defining all discourse strategies in terms of elements of a generalized meta model and relating all source model elements to this same general meta model.

• We have worked out a formalism for representing all the knowledge pertaining to deep explanation generation. The generation algorithm is formulated as a top-down planning process with unification and backtracking.

A prototype system for deep explanation generation is implemented, but is so far only demonstrating the feasibility of building these kinds of components. The system is intended to be refined and extended as the work on explanation components in conceptual modeling in carried on.

16.2 Further Work and Conclusions

This work is only a first step towards a general explanation component for CASE environments. There are several possible directions for further work, and many of these have already been alluded to in previous chapters. In particular, one should

• do extensive field studies to assess the desired content and structure of explanations, given a particular user and a particular context. McKeown [168] and McCoy [165] have done this in their selected domains, but there are yet no such studies in the field of conceptual modeling. Having done this, one could expect a refinement of the discourse strategies and the views defined in Chapter 11, so that the quality of the explanations could be improved.

• consider a formalization of the relationships between linguistic notions and information system concepts. With a firm foundation here, it would be easier to ensure that appropriate source model elements are constructed and to check the quality of discourse strategies (see Section 13.3).

• do more thorough analyses and comparisons of various kinds of conceptual modeling languages. This would help us to refine our generalized meta model, DSM, which in turn would influence the whole deep generation process. Our current DSM is based on the results in AMADEUS [152], but we have not at this stage seen it necessary to introduce the whole Unified Model from AMADEUS.
16.2. Further Work and Conclusions

- complete the tactical component and add the necessary routines to finish the whole explanation component (see Chapter 15).

- consider implementation issues like efficiency and effectiveness of generation. Our prototype, which is implemented in Prolog, should in due course be replaced by an implementation with a more satisfactory performance.

In conclusion, our work has provided a basis for explanation generation in conceptual modeling, and shown how to construct and integrate a general explanation component into contemporary CASE environments.

The remaining and overall consideration, now, is whether explanations in conceptual modeling are so useful that it justifies the effort of building the explanation component. We have indicated how explanations could be used in the modeling process (Chapter 2 and Chapter 13), and we have also discussed its potential as a unifying comprehension-enhancing strategy in Chapter 4. However, since there are no such components available today, it is at this state rather difficult to measure any real improvements of comprehensibility in conceptual modeling. This has to be done as the whole component is being implemented and put into operation. And in this connection, it is worth noting that the explanation component has a generic aspect. Even though its realization can be expensive and demanding, the costs could be justified, since it afterwards can be reused on other modeling languages and in other CASE environments without any substantial modifications.
Part VI

Appendices and Bibliography
Appendix A

Conceptual Model for a Banking System

In this appendix, we are going to model a part of a computerized information system. First, we will give a general description of the system that is to be modeled. After an introduction to a simple modeling language, we will construct a model of the system using that particular language.

A.1 System Description

The system is part of a computerized banking information system. Interacting with the bank’s loan advisors and the available databases, the system can offer functions that guide the handling of loan applications. For the purpose of our work, we assume the system to keep at least the following information about its customers: name, income, total savings, total loans, and classification. Classification is used to characterize the customer’s reliability as a bank customer, and can be any of the following three values: good, neutral, or bad.

There are two main functions in the system, registration of customers and processing of loan applications.

Registering a new customer, the user must provide her name and current income. Since there is no previous record on the reliability of her, a default classification of neutral is set. The information is stored in a customer profile database.

When a loan application is processed, the system needs the name of the customer and the size of the loan requested. The processing itself is divided into two consecutive stages: (1) find customer’s income and classification, and (2) decide whether the customer should
be granted the requested loan, or perhaps offered a smaller loan. To find the necessary information about income and classification, the customer profile database is consulted. If there is no record of her, she has to be registered in order to proceed the processing. Afterwards, the customer’s total savings and loans are analyzed. According to the policy of the bank, the following rules govern the processing of loan requests: (1) A bad customer is never to be granted a loan; (2) if the classification is neutral, the customer’s total amount in loans must not exceed the sum of her income and savings; and (3) a good customer can be granted a loan as long as her total loans are lower than or equal to her savings plus twice her income.

Using these rules, the system recommends that the loan application be either approved or rejected, and a notification on the decision is given to the user at the end of the process. If the application is rejected, the system will suggest a smaller loan that is recommendable from the policy of the bank.

### A.2 A Language for Representing the System Model

The system is modeled using an extended process model. Taking the traditional data flow diagrams as a basis, the following additions have been made:

- Describing input flows to and output flows from processes, we are using AND ports and XOR ports taken from PrM [87] (see Figure A.1(a)-(c)). We have adopted the notion of triggering flows from PrM, but as Figure A.1(d) shows, a different notation

```
WHEN <triggering flows>
IF <inflow expression> THEN
<flow expression>
SEND <outflows>
```

Table A.1: Process logic rules.
Figure A.2: The overall diagram P0 Register customers and process loan requests.

- Data flows are annotated with a record indicating the kinds of data associated with the flows. In Figure A.2, the flow Loan_request encompasses a record containing the data items name and amount.

- The logics of leaf processes are modeled as process logic rules of the form shown in Table A.1, where <triggering flows> denotes an event described by the arrivals of certain data flows, <inflow expression> is a logical expression involving items from the input flows, <flow expression> a logical expression that may involve both input flows and output flows, and <outflows> the flows produced by the rule.

Of course, this process language is limited in expressiveness as well as in formality. It is basically a simplification of the PPP language [87], but it should be sufficiently detailed to illustrate the use of explanations when modeling information systems and executing these models. We will call the language PrM∗, since it is a modification of PrM where process logics are specified as rules.

A.3 Conceptual Model
Figure A.3: Decomposition of P3 Process loan request.

The overall process diagram is shown in Figure A.2. Process P3, Process loan request, is decomposed in Figure A.3, while the behavior of all the other processes are described as decision tables in Table A.2 through A.5.

<table>
<thead>
<tr>
<th>id</th>
<th>process logic rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1.1</td>
<td>WHEN No.customer THEN Requested_loan.amount = Loan_request.amount AND No.registration.name = Loan_request.name</td>
</tr>
<tr>
<td></td>
<td>SEND No_registration AND Requested_loan</td>
</tr>
<tr>
<td>r1.2</td>
<td>WHEN Customer_information THEN Requested_loan.amount = Loan_request.amount AND Customer_data.name = Loan_request.name AND</td>
</tr>
<tr>
<td></td>
<td>Customer_data.classification = Customer_information.classification AND Customer_data.income = Customer_information.income</td>
</tr>
<tr>
<td></td>
<td>SEND Customer_data AND Requested_loan</td>
</tr>
</tbody>
</table>

Table A.2: Process logic rules for process P1 Check customer.

Using P3 as an example, we can indicate the semantics of the language PrM:

P3 receives data from two flows, Requested_loan and either Customer_data or New_customer_data. Due to the XOR port in the process's input specification, P3 cannot receive data from both Customer_data and New_customer_data. It is triggered when Customer_data or New_customer_data arrives, while Requested_loan is received at some later stage of the execution.

The output specification of P3 simply says that a recommendation containing the items result and offer is made available to the loan advisor (agent A1).
A.3. Conceptual Model

<table>
<thead>
<tr>
<th>id</th>
<th>process logic rule</th>
</tr>
</thead>
</table>
| r2.1 | WHEN Customer.name THEN  
| | Registration.done = true AND  
| | New_registration.name = Customer.name.name AND  
| | New_registration.classification = 'neutral' AND  
| | New_registration.income = Customer.income.income  
| | SEND Registration.done AND New_registration |
| r2.2 | WHEN No_registration THEN  
| | New_customer.data.name = No_registration.name AND  
| | New_customer.data.classification = 'neutral' AND  
| | New_customer.data.income = Customer.income.income AND  
| | New_registration.name = No_registration.name AND  
| | New_registration.classification = 'neutral' AND  
| | New_registration.income = Customer.income.income  
| | SEND New_customer.data AND New_registration |

Table A.3: Process logic rules for process P2 Register customer.

P3 is decomposed into processes P3.1 and P3.2. On the basis of name, classification, income, savings and loans, P3.1 calculates the maximum size of loan that can be granted to the customer. The rules governing this calculation are given in Table A.4. The third rule from this table, for example, says that if the customer is classified as good, then maximum is equal to her current savings plus twice her income minus her current loans. P3.2 is triggered by Loan.limit, receives data from Requested_loan and Loan.limit, and recommends that the loan application be approved (Recommendation.result = approved) if the requested amount is lower or equal to the calculated maximum (rule 2 in Table A.5). If Requested_loan.amount is higher than maximum, Recommendation.result is set to rejected and a loan offer equal to maximum is suggested instead.

The interpretation of the rest of the model should be obvious.
<table>
<thead>
<tr>
<th>id</th>
<th>process logic rule</th>
</tr>
</thead>
</table>
| r3.1.1 | WHEN Customer.data  
|       | IF Customer.data.classification = 'bad' THEN  
|       | Loan_limit.maximum = 0 AND  
|       | Loan_limit.name = Customer.data.name  
|       | SEND Loan_limit |
| r3.1.2 | WHEN Customer.data  
|       | IF Customer.data.classification = 'neutral' THEN  
|       | Loan_limit.maximum = Total_savings.savings +  
|       | Customer.data.income - Total_loans AND  
|       | Loan_limit.name = Customer.data.name  
|       | SEND Loan_limit |
| r3.1.3 | WHEN Customer.data  
|       | IF Customer.data.classification = 'good' THEN  
|       | Loan_limit.maximum = Total_savings.savings +  
|       | 2 * Customer.data.income - Total_loans AND  
|       | Loan_limit.name = Customer.data.name  
|       | SEND Loan_limit |
| r3.1.4 | WHEN New_customer.data  
|       | Loan_limit.maximum = New_customer.data.income AND  
|       | Loan_limit.name = New_customer.data.name  
|       | SEND Loan_limit |

Table A.4: Process logic rules for process P3.1 Calculate maximum size of loan.

<table>
<thead>
<tr>
<th>id</th>
<th>process logic rule</th>
</tr>
</thead>
</table>
| r3.2.1 | WHEN Loan_limit  
|       | IF Loan_limit.maximum < Requested_loan.amount THEN  
|       | Recommendation.result = rejected AND  
|       | Recommendation.offer = Loan_limit.maximum  
|       | SEND Recommendation |
| r3.2.2 | WHEN Loan_limit  
|       | IF Loan_limit.maximum ≥ Requested_loan.amount THEN  
|       | Recommendation.result = approved AND  
|       | Recommendation.offer = Requested_loan.amount  
|       | SEND Recommendation |

Table A.5: Process logic rules for process P3.2 Respond to request.
Appendix B

The PPP CASE Environment

PPP\(^1\) is an experimental integrated CASE environment. It is developed at the Norwegian Institute of Technology, but there are also some ties to the ESPRIT projects Tempora [136, 149] and IMSE [185]. Its main focus is currently on the early stages of development, although it is intended to cover all stages as the research going on in the PPP project matures and gets integrated into the present environment.

A prototype of the environment has been implemented in BIM-Prolog and runs on Sun work stations under Unix and Sunview. It includes a graphical conceptual modeling language, a syntax-oriented editor for constructing conceptual models, various verification checks, a report generation facility, and some code generators for producing complete runnable program code from PPP models. The functionality of the prototype is presented and discussed in [87, 148, 150].

Major extensions to this first version are envisioned in the years to come (see for example [250]). There are ongoing works on several aspects of the environment, and these are meant to govern the implementation of a new, more comprehensive environment for information systems development. In particular, the following topics have been investigated with an eye on future improvements and extensions:

**Specification databases and Configuration Management** A specification database includes all information about the stage products as well as the organization and control of the development project. For PPP, the work on these databases is expected to give a more flexible and powerful repository underlying the environment [5].

**Performance modeling** In [185], the inclusion of performance data into conceptual models is discussed. A suggestion for performance modeling is made, and a small PPP-

---

\(^1\) "PPP" stands for Phenomena, Processes, and Programs.
like environment is implemented to prove its potential and importance to information systems engineering.

**Model validation** In addition to the already implemented code generators, some other validation techniques are intended to be included in the environment. In [216], the use of abstraction mechanisms and view generation routines is proposed. An internal language for executing PPP models is documented in [249], and in this work an explanation generation component that exploits model execution techniques and view generation techniques is presented.

**Modeling language and verification** The conceptual modeling language in PPP is revised in [254]. Among the advantages of this modified modeling environment is the increased support for verification checks and a cleaner conceptual approach to operations on databases.

For the purpose of this work, we will here only present the conceptual modeling part, and with an emphasis on techniques for validating PPP models.

### B.1 Conceptual Modeling in PPP

In the PPP CASE environment, conceptual models of the future information system are incrementally and iteratively constructed and validated. Starting with an initial conceptual model, the user can validate it using a number of validation techniques, and the results from this process can necessitate new cycles of model modifications and validation. The conceptual model is iteratively modified and validated until it is deemed a satisfactory representation of the information system to develop.

Key words characterizing the PPP architecture are **formality** and **integration**. The modeling language used in PPP is given a formal foundation that opens for a wide variety of tool-supported checks, analyses, and transformations. The various aspects of the conceptual model are tightly integrated, products at different development stages are related, and the new validation techniques being developed are supposed to be combined in an integrated validation environment.

The modeling approach is illustrated in Figure B.1, and in the following we will go into some more detail on the construction and validation of PPP conceptual models.

**Model Construction**

The PPP language is an integration of four independent sublanguages, each addressing different aspects of the business area and its associated information system: (1) *PrM (Process Model)* is an extension of the traditional data flow diagrams and is used to model hierarchical, functional aspects. (2) *PhM (Phenomenon Model)*, which belongs to the class of
semantic data models, is included for the modeling of structural, static aspects. (3) PLD (Process Life Description) is a program design language and is needed for the specification of process logic of bottom-level processes in PrM. (4) UID (User Interface Description) is a language for describing user interface objects and their reactive behavior. In the current environment, UID is not included and the modeling must start with either PrM or PhM, but as the new architecture is realized any of the languages may initiate the modeling process.

During the modeling process, requirements to the future information system are incorporated into the conceptual model. Similarly, on the basis of user’s validation of the model, modifications are introduced to improve its qualities. As details are gradually added to the models, parts of the PhM model are associated with the content of flows and stores in the PrM model, and PLD models are created for bottom-level PrM processes. Automatic transitions from PrM processes to PLD models are introduced to speed up the transition from PrM modeling to PLD modeling. At all stages of the model construction part, a priori verification rules are enforced and the user can initiate checks on a posteriori rules.

Model Validation

In the current prototype environment, only the generation of program code is supported. From the PrM model and the associated PLD models, the environment can generate Ada code, TEQUEL/C code, and Simula code; from the PhM model SQL code can be generated; and from the UID model, Motif code is generated.

Apart from this work (and the smaller projects [8, 126]), the research in PPP concerning validation is concentrated on complexity reduction and model execution. The work in complexity-reducing (abstraction) techniques is performed in close co-operation with the Tempora project, and the intended outcome is a theoretical basis for generating user-
Appendix B. The PPP CASE Environment

tailored views of the conceptual model. On the execution side, an internal language is
designed for better controlling the execution of PPP models, and a tracing component for
storing and retrieving information from model executions is being implemented.

B.2 Constructing a PPP Model

Explaining the main features of the sublanguages of PPP, we will use the case study in [86]
as a basis. Since UID is yet not related to the other sublanguages and is not part of what is
traditionally referred to as the functional specification of an information system, the UID
modeling part is left out of the presentation. The interested reader is referred to [109] for
a discussion of the UID language.

The business area used in our example modeling is banking and can be summarized as
follows:

A customer of this bank can have several accounts and loans registered. She is
identified by an id, but the bank also keeps her name, address, and classification. The
classification is the bank's judgement of the credibility of the customer, and is assigned
one of the values good, neutral, or bad.

In the banking system, new accounts may be opened and monthly account state-
ments can be issued to the customers. If the customer is to open an account for the
first time, she must also be registered and given an initial classification as a neutral
customer.

Another task of the system is to provide support for transactions on bank accounts.
A new transaction is first checked to find its category (deposit or withdrawal), and to
see if the specified account exists for the customer. A withdrawal is also checked to
see if the balance exceeds the requested amount. If so, the customer is notified, and
she must respond with a lower amount. She can abort the transaction by giving a zero
in response to the request. The verified withdrawal transaction is then processed by
updating customer's account.

The banking system also allows customers to apply for loans, but loans can only
be granted if the customer has a bank account and is not classified as a bad customer.
The requested loan amount is compared to her salary, current loan, and current bank
account balances, and the bank may offer either the requested loan or a somewhat
smaller loan.

We will in the following show how the banking system is modeled in PhM, PrM, and PLD,
though we will not present a complete model of the system.
PhM Modeling

The PhM language is used to model static structures of the information system. It includes the traditional concepts of entity classes, relationship classes, attribute classes, and data types, as shown in Figure B.2. Attribute classes can be associated with both entity classes and relationship classes (commonly referred to as ER classes), and appear on the arrows between ER classes and data types.

As an extension to the classical entity relationship model [43], PhM offers a more powerful attribute relation, a more precise cardinality specification, and a concept for modeling subclasses of entities.

There are four types of attribute relations. An attribute class may be an identifier (id), in which case its value uniquely determines instances of the associated entity class. It may be equal to the traditional attribute concept (att), when it denotes single-value properties to instances of ER classes. It can be a repeating group (rep), which denotes multiple-value properties to instances of the class. And at last, it can be a quality relation (qual), and this relation is used to describe properties that are characteristic to a class rather than the elements of the class.

The relationship classes between two entity classes may be one-to-one (1:1), one-to-many (1:N), many-to-one (N:1), or many-to-many (N:M). In addition, a covering specification indicates an entity class’s involvement in a relationship class. If all instances of an entity class have to participate in a relationship class, the entity class in said to be fully (f) related to this relationship class. Otherwise, the entity class is partially (p) related to the relationship class.

Subclasses of entity classes are modeled by means of a special subclass construct. It looks like the rectangle for entity classes, except for the extra “s” put into the lower right corner of the rectangle, and an arrow is set to point from the subclass to the entity class.

In Figure B.2, then, Account is an entity class that is involved with two relationship classes (updates and has). Instances of Account have the attributes acc_no, balance, and interest_rate, of which acc_no identifies the instances of the class. A customer may have several accounts, but an account must be owned by just one customer. The covering specification says that all accounts are owned by some customer and all customers have at least one account. However, as seen from the model, a customer does not necessarily send any loan application.

When the PhM model is finished, it is possible to automatically generate SQL statements for creating the corresponding database. For the entity class Account and relationship class has, for example, the environment generates the statement

---

2 A relationship class in not allowed to have an identifier attribute.
3 Following the guidelines in [240], we include sends into Account, since there is an N:1 relationship between Account and Customer.
create table account
    (acc_no        int     not null,
     balance      float   null,
     interest_rate float   null,
     c_id         int      null);

The contents of stores and flows in PrM models are specified as views of the SQL tables created from the PhM model.

PrM Modeling

The PrM language is used to model functional aspects of a system and the interaction with its environment. The top level PrM diagram for the banking system is shown in Figure B.3, and Figure B.4 and Figure B.5 show how processes P1 and P2 are decomposed into separate diagrams.

PrM processes, stores, and flows have the same meanings as in DFD. However, flow contents can be described by connecting them to data types or entity types from a PhM model. External agents, like Customer in Figure B.3, have the same semantics as external entities in DFD.
Figure B.3: Top-level diagram for the PrM model.

Figure B.4: Decomposition of process P1.
Figure B.5: Decomposition of process P2.

Having developed a DFD-like model, we can add further details using additional constructs of PrM. These constructs deal mainly with the modeling of control flows, more precise descriptions of process interfaces, and temporal information.

Control flows are modeled by the use of triggering and terminating properties of data flows. These properties determine when a process will start and stop its execution. If a process is passive and receives the right combination of triggering flows, it will start executing. On the other hand, an active process terminates by sending a combination of terminating flows. Non-triggering and non-terminating flows can be sent/received any time while the process is active. In Figure B.4, process P1.4 can be activated if it receives the triggering input flow Ok_withdrawal⁴, and will terminate by sending the terminating flow Withdrawal_result.

Process interfaces, i.e. logical relationships between input flows and output flows, can be modeled using input ports and output ports, respectively. There are three basic kinds of ports corresponding to the three logical connectives: conjunction (AND), disjunction (OR)⁵, and exclusive disjunction (XOR). From the decomposition diagram in Figure B.4, we see that the input port of P1.1 is an AND port with two input flows, i.e. both input flows must be received during an execution of the process. The outer output port of the process is an XOR port, which means that exactly one of its elements (flows or flow groups) is to be sent.

Moreover, a port may be conditional, repeating, or both in any combination. A conditional

⁴Marked with a 'T'.
⁵The OR port is not accepted by the code generators.
port reflects a situation where flows are sent or received depending on some condition. A repeating port means that flows may be received or sent a number of times during a single execution of a process. Also, composite ports can be formed by placing ports inside each other. For instance, the input port of P1 corresponds to the logical expression \( \text{AND} (\text{Previous\_balance}, \text{Transaction}, \text{COND} (\text{New\_withdrawal})) \). The dotted line of the AND port for the input flow \text{New\_withdrawal} indicates conditionality.\(^6\)

The timer shown in Figure B.3 sends a signal to process P4 at the first of each month. In general, timers can either send a signal at a specified point of time or represent a specified delay from its time of activation.

Now, as an example we can describe a single execution of process P1.2 as follows: If the process is passive, it can be activated by receiving the flow Withdrawal\_transaction. It terminates by sending either the flow Aborted\_transaction or the flow Ok\_withdrawal. During execution, it may receive the flow New\_withdrawal, and it may send the flow Withdrawal\_rejection.

It is not necessary — and in many cases unlikely — that process port information can be given in top level models, before a process is decomposed. As can be seen from the model above, the output port of process P1 is rather complex, and probably not modeled before the process is decomposed. In the PPP environment, consistency checks can be performed to see if a process is consistent with its decomposition, and as a side effect of this, it is possible to abstract ports of a higher level process from its decomposition.

**PLD Modeling**

PLD modeling starts off where PrM decompositions end. PLD models are used to specify process logic of bottom-level processes which will be automated. Constructs for assignment, iteration, and selection are defined. In addition, two constructs for receiving and sending data provide interprocess communication and communication with users and databases. Figures B.6 through B.8 show some of the PLD models for the automated parts of P1 and P2 in our banking system, and we will in the following use the model in Figure B.6 to describe the language. The rest of these PLD models can be found in [88].

As the developers decide to describe the process logic of a process, an initial PLD model is generated from the process interface by the PPP environment. After that, modeling proceeds by adding additional computational details to the PLD model. The shaded areas in Figure B.6 indicate information added after the initial diagram has been generated. The initial diagram is consistent with the ports, both with respect to the logical relationships among flows, and with respect to the temporal order of receipts and sendings of flows, as specified through their triggering and terminating properties.

The control flow of a PLD model is top-down and from left to right. The **start construct** simply marks the start of the PLD model. First, the process receives Withdrawal\_transac-

\(^6\)Repetition is indicated by an unbroken line.
The `receive construct` holds this information, together with variable names and types of the data received. Here, we see that the information includes account number, account balance, customer name, amount, and date.

A `choice construct` is then used to distinguish between two alternatives: either the balance exceeds the specified amount, or it does not. The choice construct is a compound of one `selection construct` which marks the choice situation, and one or more `alternative constructs`, one for each alternative to be evaluated. Each alternative contains an expression. If this expression evaluates to true, the block below the alternative construct will be executed.

For the case that the balance is OK, the flow `Ok_withdrawal` is sent for update of the account. Otherwise, the available amount (`Withdrawal_rejection`) is sent to the customer, with a request for a new amount. This amount is then received (`New_withdrawal`), and then the withdrawal amount is updated to this new amount in an `assignment construct`. Next, a new choice construct determines whether the user aborted the transaction by giving a zero amount. If this is the case, the flow `Aborted_transaction` is sent. Otherwise, `Ok_withdrawal` is sent to P1.4.

The `send construct` identifies the data flow and the receiver. In addition, it contains one expression for each data element to be sent. The data elements communicated by means of send and receive statements are defined by the corresponding flows in the PrPM model. Besides, if a flow is sent to a store, or received from a store, an appropriate SQL statement is associated with the operation, though the statement refers to the database view defined
for the store rather than the store itself.

The **loop construct** is not illustrated in the example model. The loop construct contains an expression to be evaluated each time the loop body can start execution, and corresponds to a WHILE-loop or a FOR-loop of high level programming languages. In addition, it is possible to declare local variables within a PLD diagram.
B.3 Validating the PPP Model

Currently, only a translational approach to prototyping is available in the validation step. Since we have already set up the new architecture for model validation [250], though, we will here outline a validation environment that also comprises the future techniques of tracing and view generation.

Model Execution

The PPP environment supports model execution by providing translations from PPP models to various executable target languages. Translation from PPP models to Ada is described in [84, 87, 150]. Translations to other target languages have been experimented with as well. In [136, 149] translations to a rule-based language (TEQUEL/C) are described, in [92] translation to Simula/Demos is presented, and in [249] PPP models are translated to an internal representation for execution. Details of these translations will not be presented here, but the result from generating Ada from the PLD model in Figure B.6 is given for illustration.

In short, a generated Ada program consists of a set of concurrently executing tasks, one task for each PLD model. These tasks communicate by a rendezvous mechanism corresponding to the receive and send constructs found in the PLD models. Port structures of bottom-level processes are represented by special synchronization structures in Ada. For each construction in PLD, then, a special construction in Ada is defined. As the code generator is implemented, however, types and variables from Ada have to be added to the PLD model for the manipulation of data flows. We will not go in further details here, but Figure B.9 shows the body of a task generated from the PLD model in Figure B.6.

Anticipating the ongoing work on traces, we have in Figure B.9 also indicated how traces can be recorded during execution. For each event to be recorded, a probe has to be inserted into the generated code. This is simply a call to a procedure which stores information about the events according to a general predefined schema. There are mainly two kinds of events: Application of transitions (dynamic laws) at various levels of abstraction, as well as interaction with the environment (input). A global clock is used to keep the temporal ordering of events.

The lower part of the figure shows parts of a recorded trace from an execution of the task. It represents an execution in which a transaction requesting a withdrawal that is too large is received (1). The withdrawal amount is found to exceed the available amount (2), hence the flow Withdrawal_rejection is sent (3). After a new amount has been received from the user on the flow New_withdrawal (4), a new test is performed to see if the transaction will be aborted or not. In this execution, the user responds with a zero amount, the test for abortion succeeds (5), and the flow Aborted_transaction is sent.

Information recorded about transitions includes an identifier, a list of values referenced
in preconditions or as new values are computed, temporal information giving start state and end state of the transition, and possibly information about the results of updates of variables or flows. For instance, the second tuple represents the execution of the PLD block initiated with the second alternative construct of the first choice construct from the top. It references values of account.balance and amount in its precondition, it was applied in state S2, and terminated in S5.

Trace information can be retrieved using a specialized query language which offers different trace views into the trace. For instance, it is possible to select information about particular transitions, to follow dependencies between different transitions, and to retrieve changes made to a particular static object. The views may also be combined by nesting queries, providing more flexible retrieval mechanisms.
Complexity Reduction

So far, complexity reduction in PPP is only achieved by means of the inherent abstraction mechanisms in the modeling language. For the co-operative development of large-scale information systems, however, a number of complexity-reducing mechanisms or simplifications is now being developed, and these should ease the communication among stakeholders on properties of the information system [216, 215].

Simplifications are provided by allowing different views of a model, each view focusing on a different aspect of the model. Rather than operating on a full model, relevant views can be applied at different stages of the modeling and validation process depending on the problem to be solved and the actors involved. In this way, a systematical approach for suppressing irrelevant details and highlighting relevant details of a model is provided.

![Diagram](image)

Figure B.10: (a) A model view resulting from horizontal abstraction. (b) Ports abstracted away and layout is improved.

In Figure 5 the abstraction mechanisms are illustrated. Given a situation where we want to emphasize on how withdrawal amounts are verified in our banking system only one process, P1.2, and its surrounding need to be shown. A model view is generated based on horizontal abstraction as depicted in Figure B.10(a). Furthermore, the details of the ports are superfluous and only clutter the diagram. In Figure B.10(b), the ports are abstracted away and the diagram is restructured to improve its layout.
Appendix C

Outline of Functional Grammar

In this appendix, we present a summary of the theory of Functional Grammar (FG). We will concentrate on aspects relevant to our explanation generator, so readers more interested in the linguistic motivations and considerations are strongly encouraged to consult the FG literature. Some of the features of FG are omitted, or at least simplified, due to their lack of relevance to our exploitation. The presentation is made from a technical and computational point of view, but this is of course not the only possible point of view.

Functional Grammar is a general theory of the grammatical organization of natural language. It is developed by Simon C. Dik [67], and has in recent years been the subject for intensive studies within linguistics (see for example [94]) as well as from an information systems or artificial intelligence perspective. To take some examples, it has been used as a general knowledge representation language [65, 246, 247], as a conceptual modeling language [63], as a foundation for a layered logical system [66], and as a underlying formalism for parsing and generating natural language [50].

In FG, the nature of language is seen from a functional point of view. Predicates form the central structuring unit of the theory, and these are used at all levels of representation. Similar conceptions of natural language systems are also found in LFG [28], FUG [133], HPSG [193], and systemic grammars [93, 252], among others.

The theory of Functional Grammar contains two main components, one is forming a knowledge representation language and another is keeping and using a set of expression rules. As a knowledge representation language for clauses, FG has a layered representation philosophy, encompassing predicates, predications, propositions, and messages. The expression rules map underlying clause structures of FG onto linguistic expressions, and in that process the form of each part of the expression, their internal ordering and prosodic contour (accent and intonation) have to be determined. Both components are discussed in some detail below.
C.1 Knowledge Representation Language

The FG knowledge representation language is described at four different clause levels\(^1\):

- **Predicates** are expressions which designate properties of, or relations between entities. **Entities**, then, are represented as terms, which is a structure consisting of a head (typically a noun predicate frame from some lexicon) and an optional set of restrictors. There are derived and basic predicates, of which the last group is contained in the lexicon as predicate frames. With reference to languages in general, predicates lead to a conceptualization, in which conceptual structures of the world are specified.

- **Predications** are created by inserting terms into predicate frames and, among other things, instantiating the frame at a spatio-temporal dimension. They denote a state of affairs SoA and in that sense build models of some real or imaginary world.

- **Propositions** are made up from predications and include information expressing the speaker’s attitude towards the propositional content. As opposed to predications, propositions denote possible facts, something that can be true or false with respect to a certain state of affairs.

- **Messages**, also called **speech acts** or **clause structures** [67], consist of a frame of a certain illocutionary type that contains a proposition. Pragmatic functions are also added at this level. Clause structures correspond to the use of language in communication processes, and in a representational perspective they provide an interface to the information to be communicated.

In Weigand’s formalization [246], these four levels are said to correspond to the notions of **data dictionary**, **database**, **knowledge base\(^2\)**, and **message base** (the interface to the external world).

A more detailed illustration of the clause structures in FG is given in Figure C.1. As seen from the figure, **nuclear predications** are formed from predicate frames and terms. Core predications, extended predications, propositions, and clause structures are developed by adding more operators (\(\pi_i\)), satellites (\(\sigma_i\)) and grammatical functions.

In the following, we will confine ourselves to issues relevant to the theory’s function as an underlying representation formalism for explanation parts. We will focus on the structures of predicate frames and extended predications, and readers more interested in the linguistic background for these structures are referred to [67, 94]. As an illustration, we will analyze the sentence

> "Every process must produce at least one flow."

---

\(^1\) In our exposition of FG, we will mostly stick to the terminology of Weigand [247].

\(^2\) A knowledge base is here assumed to maintain several databases at the same time.
at each of these levels.

Predicate Frames

The lexicon is an integral part of FG, specifying grammatical, structural and conceptual information for each word of the language. For our purposes, though, it is not necessary to dig into all the complexities of FG lexica, and we will instead focus on the representation of predicate frames and terms, and on the use of the subtyping mechanism.

Predicate frames specify argument structures and are the main building blocks of FG. As an illustration, take the predicate frame

\[ \text{produce}_V(x_1: \text{animate}(x_1)) \text{Ag}(x_2) \text{Go}, \]

where we have indicated the argument structure of "produce". Produce forms the predicate head, and V marks the head as a verbal. Other possible head types are pN for proper
nouns, N for nominals, and A for adjectives. Ag and Go stand for agents and goals, respectively, and indicate the semantic roles assigned to the arguments of produce. We will return to the system of semantic roles in FG a little bit later, but for now we focus on the frame structures.

The same type of predicate frames is also used to represent nominals in the lexicon:

\[ \text{process}_N(x_i) \]
\[ \text{Georges}_pN(x_i; \text{human}(x_i); \text{male}(x_i))_0 \]

\( \emptyset \) says that no semantic role is assigned. Human\((x_1);\text{male}(x_1)\) is a selection restriction which requires the term to be substituted for \(x_1\) in the predicate of Georges to be both a human and a male. Correspondingly, in the produce frame above, the producer had to be an animate.

Terms, predicate frames, and all the other levels of representation are analyzed as functions of the form

\[ \text{Operator Variable}_i: \ [\text{Specification}] (\text{Variable}_i), \]

of which Specification is a one-place function and Variable\(_i\) is a parameter. The notation introduces a certain redundancy, since Variable\(_i\) is repeated, and following the conventions of [67, 247], we will in the following simplify the representations and omit the last Variable\(_i\) in the structure.

In the lexicon, taxonomic relationships between terms and predicates are included as part of the structural information. We can represent produce and process in the lexicon as

\[ \text{produce}_V(x_i; \text{animate})_A g(x_2)_{G_0} < \text{make}_V(x_i)_{A g(x_2)_{G_0}} \text{ and} \]
\[ \text{process}_N < \text{concept}_N, \]

which say that produce is a subtype of make and process of concept. Note how the representations of terms and frames are made simpler by omitting the functions' variables.

Semantic Roles

Semantic roles describe the roles which entities play within the SoA designated by the predication. Each argument to a predicate is given a distinct semantic role, and the assignment of these roles follow a rather complex analysis of argument structures and interpretations of predicates.
<table>
<thead>
<tr>
<th>class</th>
<th>semantic role</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>Agens</td>
<td>the entity controlling an action.</td>
</tr>
<tr>
<td></td>
<td>Positioner</td>
<td>the entity controlling a position.</td>
</tr>
<tr>
<td></td>
<td>Force</td>
<td>the non-controlling entity instigating a Process</td>
</tr>
<tr>
<td></td>
<td>Processed[Exp]</td>
<td>the entity that undergoes a process.</td>
</tr>
<tr>
<td></td>
<td>Ø(Zero)[Exp]</td>
<td>the entity primarily involved in a state.</td>
</tr>
<tr>
<td>(2)</td>
<td>Goal[Exp]</td>
<td>the entity affected or effected by the operation of some controller</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Agens/Positioner) or Force.</td>
</tr>
<tr>
<td>(3)</td>
<td>Recipient[Exp]</td>
<td>the entity into whose possession something is transferred.</td>
</tr>
<tr>
<td></td>
<td>Location</td>
<td>in place where something is located.</td>
</tr>
<tr>
<td></td>
<td>Direction</td>
<td>the entity towards which something moves/is moved.</td>
</tr>
<tr>
<td></td>
<td>Source</td>
<td>the entity from which something moves/is moved.</td>
</tr>
<tr>
<td></td>
<td>Reference</td>
<td>the second or third term of a relation with reference to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>which the relation is said to hold.</td>
</tr>
</tbody>
</table>

Table C.1: Semantic roles used in FG.

Let $A^i$ be the $i$-th argument of a predicate. With reference to the classification of semantic roles in Table C.1, we have the following rules governing their assignment in FG [67]:

*In all predicate frames, $A^1$ has one of the roles in (1). In two-place predicate frames, $A^2$ has one of the roles in (2) or (3), whereas in three-place frames $A^2$ is the goal and $A^3$ has a role from (3).*

Some examples illustrating combinations of semantic roles are listed below:

"John (Ag) kissed Mary (Go)"
"John (Ag) apologized to Peter (RecExp)"
"John (Ag) jumped from the table (So)"
"The apple (Pro) fell from the tree (So)"
"The boy (Ø) resembles his father (Ref)"
"John (Ag) gave the book (Go) to Mary (Rec)"
"John (Pos) kept his money (Go) in a sock (Loc)"
"The wind (Fo) blew the leaves (Go) into the kitchen (Dir)"

All these roles refer to arguments of the specified predicate frames, but there are also some optional arguments called satellites. For example, "by plane" in "I went to Nice by plane" is called a manner satellite. We will come back to satellites when the structures of FG predications is explained.
Terms

A term is an expression that refers to an entity in some world. The general structure of terms is

$$\Omega x_i : \Phi_1 : \ldots : \Phi_n,$$

where $\Omega$ stands for one or more term operators, $x_i$ symbolizes the intended referent and each $\Phi_j$ is a predication open in $x_i$. For example, in our example we could use term structures like

\[
(*[s] \; x_1: \; \text{process}_N) \quad \Rightarrow \quad \text{"every process"}
\]
\[
(*[s] \; x_1: \; \text{process}_N: \; \text{completed}_A) \quad \Rightarrow \quad \text{"every completed process"}
\]
\[
(i[1,+] \; x_2: \; \text{flow}_N) \quad \Rightarrow \quad \text{"at least one flow"}
\]

There is a whole range of different term operators, of which two of the most important ones are illustrated below:

- **definiteness** is realized as a term operator, where the value $d$ indicates "definite" and $i$ "indefinite".

  \[
  (d[1] \; x_1: \; \text{man}_N) \Rightarrow \text{"the man"}
  \]
  \[
  (i[1] \; x_1: \; \text{man}_N) \Rightarrow \text{"a man"}
  \]

- Quantities are term operators which inform us in one way or another about the size of what is intended referred. Some possible specifications of quantities are given below:

  \[
  (i[7] \; x_1: \; \text{man}_N) \Rightarrow \text{"seven men"}
  \]
  \[
  (i[\text{many}] \; x_1: \; \text{man}_N) \Rightarrow \text{"many men"}
  \]
  \[
  (i[1] \; x_1: \; \text{man}_N) \Rightarrow \text{"one man"}
  \]
  \[
  (*[p] \; x_1: \; \text{man}_N) \Rightarrow \text{"all men"}
  \]
  \[
  (i[2,+] \; x_1: \; \text{man}_N) \Rightarrow \text{"two or more men"}
  \]
  \[
  (i[2]/d[4] \; x_1: \; \text{man}_N) \Rightarrow \text{"two or the four men"}
  \]

$s$ and $p$ denote singular and plural, respectively. The question of quantification, however, is a very complex matter, and Dik has worked out some additional, but very specific, term operators to cope with more subtle sentence constructions.

There are more term operators than shown in this very simple presentation (e.g. generality operators, ordinators and sortal classifiers), and the interested reader is referred to [67] for a comprehensive treatment of term specifications.

\[3\text{We have here adopted the notations of quantities from Weigand [247] and Dignum [64], since they are more tailored to the domain of conceptual modeling.} \]
\[ \pi_2 \epsilon_1 : ((\pi_1 \text{Pred}(\text{Arg})^k(\sigma_1)^m)(\sigma_2)^n) \]

\hspace{1cm} \text{core}

\hspace{1cm} \text{extended}

Table C.2: Various levels of FG predications (adapted from [67]).

Predications

There are several sublevels of predications, but in this presentation we will be dealing mostly with what is called the \textit{extended predication}. This is the most advanced one, and since it subsumes all the other predication types, we will refer to it as just the predication. In Table C.2, the relationships between the various levels of FG predications are depicted.

A \textit{nuclear predication} is made by inserting appropriate term structures into the argument slots of predicate frames. A \textit{core predication} is an extension of a nuclear one, in which \(\pi_1\) operators and \(\sigma_1\) satellites are added. To form an (extended) predication, \(\pi_2\) operators, a predication variable, \(\sigma_2\) satellites, and the syntactic functions of subject and object are added to the core predication.

The operators capture the grammatical means for including additional information in the clause, whereas satellites specify the lexical means. A lexical mean is an additional argument to the predicate (e.g. the manner satellite \textit{"by plane"} in \textit{"I went to Nice by plane"}) or a restrictor saying something about the predication (e.g. \textit{"tenderly"} in \textit{"I kissed her tenderly"}). \(\pi_1\) operators and \(\sigma_1\) satellites provide additional specifications of the SoA, \(\pi_2\) operators and \(\sigma_2\) satellites leave the structure of SoA intact, but locate it with respect to spatial, temporal, and \textit{"objective"} cognitive dimensions. In Table C.3 and Table C.4, the possible operators and satellites are described, respectively.

Using our example from earlier, we get the predication

\[
\text{OBL PRES } \epsilon_1: \ (\text{PF } \text{produce}_v \ (*[s] \ x_1: \ \text{process}_N)_{\text{AgSubj}} \ \\
\quad (i[1,+] \ x_2: \ \text{flow}_N)_{G\text{Obj}})
\]

Abbreviations in upper-case letters are operators, \(\epsilon_1\) is a predication variable, \(x_1\) and \(x_2\) are term variables, \textit{Subj} is subject and \textit{Obj} is object.

Propositions and Messages

In our deep explanation, we will not be dealing with propositions or messages, so we will here just briefly describe their properties.
<table>
<thead>
<tr>
<th>type</th>
<th>operator</th>
<th>values</th>
<th>abbreviations</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\pi_1)</td>
<td>Polarity</td>
<td>negative, positive</td>
<td>NEG, POS</td>
</tr>
<tr>
<td></td>
<td>Perfectivity</td>
<td>perfective, imperfective</td>
<td>PF, IMPF</td>
</tr>
<tr>
<td></td>
<td>Internal phasal aspect</td>
<td>ingressive, progressive,</td>
<td>INGR, PROGR,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>eggressive</td>
<td>EGR</td>
</tr>
<tr>
<td>(\pi_2)</td>
<td>Tense</td>
<td>past, present, future</td>
<td>PAST, PRES,</td>
</tr>
<tr>
<td></td>
<td>External phasal aspect</td>
<td>prospective, perfect</td>
<td>FUT, PROS,</td>
</tr>
<tr>
<td></td>
<td>Quantificational aspect</td>
<td>Deals with habit, frequency,</td>
<td>PERF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>continuity and intensity,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>but is not relevant here.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Epistemic modality</td>
<td>possible, certain</td>
<td>POS, CERT</td>
</tr>
<tr>
<td></td>
<td>Deontic modality</td>
<td>obligatory, permissible</td>
<td>OBL, PERM</td>
</tr>
</tbody>
</table>

Table C.3: Operators included in FG predications.

<table>
<thead>
<tr>
<th>type</th>
<th>semantic role or restrictor</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\sigma_1)</td>
<td>Beneficiary</td>
<td>entity for whose benefit SoA is effected</td>
</tr>
<tr>
<td></td>
<td>Company</td>
<td>an entity with whom SoA is effected</td>
</tr>
<tr>
<td></td>
<td>Instrument</td>
<td>the tool for doing or maintaining something</td>
</tr>
<tr>
<td></td>
<td>Manner</td>
<td>the way something is done or maintained</td>
</tr>
<tr>
<td></td>
<td>Speed</td>
<td>speed of action or process</td>
</tr>
<tr>
<td></td>
<td>Quality</td>
<td>a quality of an action or a position</td>
</tr>
<tr>
<td></td>
<td>Source</td>
<td>point of origin of movement</td>
</tr>
<tr>
<td></td>
<td>Path</td>
<td>orientation of movement</td>
</tr>
<tr>
<td></td>
<td>Direction</td>
<td>terminal point of movement</td>
</tr>
</tbody>
</table>

Table C.4: Satellites included in FG predications.

Propositions specify possible facts and include the extended predications just mentioned. In conformance with the construction of core predications and extended predications, an extended proposition is constructed by adding a new level of operators \(\pi_3\) and satellites \(\sigma_3\). This additional information reveals the speaker’s attitude towards the clause content, and can be exemplified with the possible satellite “in my opinion” and the subjective modality expressed by the adverb “seemingly”.

Messages are constructed on the basis of propositions, but may include additional operators \(\pi_4\) and satellites \(\sigma_4\) as well as the pragmatic roles of topic and focus. \(\pi_4\) operators spec-
ify the basic illocutionary force of the clause (i.e. DECL(arative), INT(errogative), and IMP(licative)), and the satellite say something about how the speaker wishes the speech act to be understood by the addressee (e.g. "briefly" in "Briefly, John is a fool"). Topic is a pragmatic role saying what the message is about, and focus is signifying which constituents of the message are communicatively the most important or salient ones.

If we should follow our example all the way down to a message, the result would be

\[
\text{DECL}(OBL \text{ PRES } e_1 : (\text{PF } \text{produce}_V \ (\star [s] \ x_1 : \text{process}_N) A_0 \text{SubjTop} \\
(\{1, +\} x_2 : \text{flow}_N) G_0 \text{ObjFoc}))
\]

Our example sentence is declarative, the topic is the process term and the focus is the flow.

### C.2 Expression Rules

The message is supposed to keep all those elements and relations that are necessary to the semantic and pragmatic interpretation of the clause. At the same time, though, it should also enable generation of natural language sentences that realize the content of the structures. In response to this last requirement, a collection of expression rules are used to map messages onto equivalent sentences of some natural language. The responsibilities of the rules, then, concern three aspects of natural language generation:

- form of constituents,
- order of constituents, and
- prosodic contour.

The form of constituents concerns the morphological properties of constituents and are taken care of by something called morpho-syntactic operators. Basically, these operators are in the form of \text{operator (input form)} = \text{output form}, where the input may either be a predicate from the clause structure or an output from an earlier operator rule. The output is added morpho-syntactic features that makes it more NL-like than the input form. Examples of such rules are

\[
\text{PAST (have}_V) = "had}_V" \text{ and } \\
\text{OBL PRES (produce}_V) = "must}_V \text{produce}_V".
\]

Rules governing the order of constituents within a sentence have two components: a syntactic template with several slots, and a set of subrules stating what elements from the clause structure or from the output of the morpho-syntactic operators will fill which slot under which conditions. In general, these placement rules are represented as follows:
underlying structure : \{a,b,c,d\}
ordering template : 1 2 3 4
placement rules : a := 2
               : b := 3
               : c := 1
               : d := 4
output sequence : c b d a

In the underlying structure, there is an unordered set of constituents. The placement rules relate the constituents to positions of the sentence, assuming that the form of each constituent has already been determined by the morpho-syntactic rules. To take an example, we can start with our example structure

\[
\text{DECL}(\text{OBL e_1: (PF produce_V (*[s] x_1: process_N) Subj Top} \\
(i[1,+] x_2: flow_N) Obj Foe))\]

Using the appropriate morpho-syntactic rules, we get a new structure that is input to the template above:

underlying structure : \{must_V, produce_V, every process_N, subj, at least one flow\}
ordering template : 1 2 3 4
placement rules : Vf := 2
               : V := 3
               : Subj := 1
               : Obj := 4
output sequence : "Every process must produce at least one flow"

Usually, though, placement rules are considerably more complicated than shown here and involves a whole rule set for determining the ordering of constituents (see for example [10]).

In recent years, FG clause structures have been experimented with in a number of natural language generation systems [10, 139, 210]. Although these systems are still in their infancy and have not reached the proper level of maturity, they are already able to generate rather complicated nested sentences with the correct form and ordering of constituents. None of the systems today, however, are able to generate paragraph-length text.
Appendix D

Translation from EML to Functional Grammar

With its abilities to represent natural language clauses, EML constitute a subset of what can be represented as FG\(^1\) predications. An EML clause structure, thus, can be translated to a corresponding predication in FG, as will be shown below.

Given an EML structure \(\Sigma\), where default values have been inserted for all missing attribute values. Let the dot notation for EML structures, \(\Sigma.\alpha\), return the value of \(\Sigma\)'s attribute \(\alpha\), and let the function \(f_{EML\rightarrow FG}\) be defined from Table D.1. The FG predication for \(\Sigma\) is computed using the following rules:

1. Give the FG predication the general structure

\[
(\pi_2)^{k_1} e_i : (\pi_1)^{k_2} Pred(I)(Arg)^{k_3}(\sigma_1)^{k_4}(\sigma_2)^{k_5},
\]

where \((\xi)^{k}\) means that \(k\) instances of \(\xi\) are included. \(\pi_i\), Pred, I, Arg, and \(\sigma_i\) are all variables and are discussed in Appendix C.

2. \((\pi_2)^{k_1}\) is given as the sequence \(\nu_1\nu_2\nu_3\nu_4\), where

\[
\begin{align*}
\nu_1 &= f_{EML\rightarrow FG}(\Sigma.p.operators.d.modality), \\
\nu_2 &= f_{EML\rightarrow FG}(\Sigma.p.operators.e.modality), \\
\nu_3 &= f_{EML\rightarrow FG}(\Sigma.p.operators.ext.phase), \text{ and} \\
\nu_4 &= f_{EML\rightarrow FG}(\Sigma.p.operators.tense).
\end{align*}
\]

\(\nu_i\) is left out of the sequence if the corresponding \(f_{EML\rightarrow FG}\)'s argument is not included in the structure.

3. \((\pi_1)^{k_2}\) is given as the sequence \(\nu_1\nu_2\nu_3\), where

\(^{1}\)"FG" stands for "functional grammar".
Table D.1: The function $f_{EML\rightarrow FG}$.

\[
\begin{align*}
\nu_1 &= f_{EML\rightarrow FG}(\Sigma.p.operators.polarity), \\
\nu_2 &= f_{EML\rightarrow FG}(\Sigma.p.operators.perfectivity), \text{ and} \\
\nu_3 &= f_{EML\rightarrow FG}(\Sigma.p.operators.int.phase).
\end{align*}
\]

$\nu_i$ is left out of the sequence if the corresponding $f_{EML\rightarrow FG}$'s argument is not included in the structure.

4. $\text{Pred}$ is given the value $\Sigma.\text{head}$.

5. $I$ is set to $V$ if $\Sigma.\text{head}$ is a verb, $A$ if $\Sigma.\text{head}$ is an adjective, and $N$ if $\Sigma.\text{head}$ is a noun.

6. $(Arg)^k$s denotes a number of obligatory arguments to the predicate in $\text{Pred}$. For each $\Sigma.\alpha$ where $\alpha$ is a semantic role included in $\text{Pred}$'s predicate frame, produce the obligatory argument term given by the structure

\[
(\Omega x_i : \Phi_1 : \ldots : \Phi_n)_{\text{Roles}}
\]

where $i$ is unique for each argument.

(a) $\Omega$ is given the value $\nu_1 \nu_2 / \nu_3$, where

\[
\begin{align*}
\nu_1 &= f_{EML\rightarrow FG}(\Sigma.\alpha.t.operators.def), \\
\nu_2 &= f_{EML\rightarrow FG}(\Sigma.\alpha.t.operators.num),
\end{align*}
\]
and \( \nu_3 \) is the translation of \( \Sigma.a\.t.operators.set \) to FG (\( \nu_3 \) is produced using rule (6)). \( \nu_i \) is left out of the sequence if the corresponding \( f_{EML-FG} \)'s argument is not included in the structure.

(b) \( \Phi_1 : \ldots : \Phi_n \) is a sequence of restrictors where each \( \Phi_i, i \geq 2 \), is the value of an element in \( \Sigma.a\.possessor \) or \( \Sigma.a\.restrictors \). \( \Phi_1 \) is given the value of \( \Sigma.a\.head \).

(c) Roles is given as \( \nu_1\nu_2\nu_3 \), where

\[
\nu_1 = f_{EML-FG}(\alpha),
\nu_2 = f_{EML-FG}(\Sigma.alpha.roles.syntax), \text{ and}
\nu_3 = f_{EML-FG}(\Sigma.a\.roles.pragmatics).
\]

As before, \( \nu_i \) is left out of the sequence if the corresponding \( f_{EML-FG} \)'s argument is not included in the structure.

The sequence of these terms is substituted for \( (Arg)^b \) in the FG predications.

7. \( (\sigma_1)^{k_1} \) and \( (\sigma_2)^{k_2} \) denote satellites to the predicate in \( Pred \), as given in Table C.4 in Appendix C. For each \( \Sigma.\alpha \) where \( \alpha \) is a \( \sigma_1 \) satellite or a \( \sigma_2 \) satellite, produce a term using the same rules as in (6). The sequence of terms from \( \sigma_1 \) satellites is substituted for \( (\sigma_1)^{k_1} \), the sequence from \( \sigma_2 \) satellites for \( (\sigma_2)^{k_2} \). For each \( \Sigma.\beta \), where \( \beta \) is a restrictor, produce a term using the rules in (6) and add it to the \( \sigma_1 \) terms.

As an example, consider the EML structure in Table D.2. Using the rules above, we get the FG predication

\[
PAST \ e_1: \ \ (PROGR \ kiss_{\nu}(d[s] \ x_1: \ boy)_{A_0SubjTop} \ (i[s] \ x_2: \ girl: \ \text{beautiful})_{GoObjFoc} \ (tender)_{Man}).
\]
Assuming a declarative illocutionary force and that no propositional attitude is expressed, we can construct the complete FG message as

\[
\text{DECL (PAST e_1: (PROGR kissy(d[s] x_1: boy)_{SubjTop} (i[s] x_2: girl: beautiful)_{ObjFoc} (tender)_{Man}))}.
\]

In natural language, the message is realized as the statement "The boy is kissing a beautiful girl tenderly".
Appendix E

Discourse Strategies

The discourse strategies defined in our strategic component core are listed below. Every strategy has a HEAD that, unless its value in nil, specifies the rhetorical relation between the parts generated by NUCLEUS and SATELLITE. These relations are briefly described in Table E.1 — the rationale for many of these are further discussed in the RST literature [160, 161]. A HEAD value of nil signifies a schema-oriented strategy with no textual dependencies indicated.

The strategies’ preconditions make use of predicates that check properties of the source model, and these predicates work as ordinary Prolog rules. They have to succeed for the strategies to be used, and if they succeed, they bind the variables given as arguments. At the end of the appendix, their functions are briefly described.

VIEWS decides in what views a specific discourse strategy can be used. If the value of VIEWS is not unifiable with the view specification currently applied in the generation process, the strategy is discarded and another one is attempted.
<table>
<thead>
<tr>
<th>rhetorical relation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dynamics</td>
<td>satellite describes the dynamics of nucleus</td>
</tr>
<tr>
<td>interpretation</td>
<td>satellite is an interpretation of nucleus</td>
</tr>
<tr>
<td>condition</td>
<td>satellite is a condition for nucleus to be activated</td>
</tr>
<tr>
<td>enablement</td>
<td>satellite provide the data necessary for nucleus to run</td>
</tr>
<tr>
<td>consequence</td>
<td>satellite describes control signals caused by nucleus</td>
</tr>
<tr>
<td>result</td>
<td>satellite contains the data produced by nucleus</td>
</tr>
<tr>
<td>instantiation</td>
<td>satellite describes instances of elements in nucleus</td>
</tr>
<tr>
<td>elaboration</td>
<td>satellite elaborates on information in nucleus</td>
</tr>
<tr>
<td>illustration</td>
<td>satellite is a graphical illustration of nucleus</td>
</tr>
<tr>
<td>evidence</td>
<td>satellite provides evidence for nucleus</td>
</tr>
<tr>
<td>example</td>
<td>satellite is an example of nucleus</td>
</tr>
<tr>
<td>association</td>
<td>satellite describes elements of concept in nucleus</td>
</tr>
<tr>
<td>aggregation</td>
<td>satellite describes parts of concept in nucleus</td>
</tr>
<tr>
<td>generalization</td>
<td>satellite describes specializations of nucleus</td>
</tr>
<tr>
<td>relationship</td>
<td>satellite has some neutral relationship to nucleus</td>
</tr>
<tr>
<td>possession</td>
<td>satellite is possessed by nucleus</td>
</tr>
<tr>
<td>duration</td>
<td>satellite specifies a duration of nucleus</td>
</tr>
<tr>
<td>cause</td>
<td>satellite causes nucleus</td>
</tr>
<tr>
<td>contrast</td>
<td>satellite is in contrast to nucleus</td>
</tr>
<tr>
<td>evaluation</td>
<td>satellite evaluates information in nucleus</td>
</tr>
<tr>
<td>justify</td>
<td>satellite justifies nucleus</td>
</tr>
<tr>
<td>circumstance</td>
<td>satellite provides the circumstance for nucleus</td>
</tr>
<tr>
<td>motivation</td>
<td>satellite motivates nucleus</td>
</tr>
</tbody>
</table>

Table E.1: Rhetorical relations used by our discourse strategies.
dynamics(E):

[HEAD
  condition
  NUCLEUS dynamics(E)
  SATELLITE
    all(presentation(succeed(D,E)))
    all(presentation(trigger(C,E)))
  VIEWS
    [ BEHAVIORAL yes ]

[HEAD
  enablement
  NUCLEUS dynamics(E)
  SATELLITE
    presentation(precondition(E, _))
  [P_OPERATORS [D_MODALITY perm]]
  VIEWS
    [ COMPUTATIONAL yes ]

[HEAD
  duration
  NUCLEUS dynamics(E)
  SATELLITE
    presentation(duration(E, _))
    all(presentation(terminate(E, _)))
  VIEWS
    [ BEHAVIORAL yes ]

[HEAD
  consequence
  NUCLEUS dynamics(E)
  SATELLITE
    all(presentation(succeed(E, _)))
    all(presentation(trigger(E, _)))
  VIEWS
    [ BEHAVIORAL yes ]

[HEAD
  result
  NUCLEUS dynamics(E)
  SATELLITE
    presentation(postcondition(E, _))
  [P_OPERATORS [D_MODALITY perm]]
  VIEWS
    [ COMPUTATIONAL yes ]

[HEAD
  nil
  NUCLEUS decomposition(E)
  VIEWS
    [ INTERNAL yes ]

[HEAD
  nil
  NUCLEUS
  presentation(act(E, _))
  [P_OPERATORS [D_MODALITY perm]]
  all(presentation(succeed(E, _)))
  all(presentation(duration(E, _)))
  all(presentation(terminate(E, _)))
]
error(E,R,D):

<table>
<thead>
<tr>
<th>HEAD</th>
<th>evidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUCLEUS</td>
<td>illegal(D)</td>
</tr>
<tr>
<td>SATELLITE</td>
<td>correction(E,R)</td>
</tr>
<tr>
<td>P_OPERATORS</td>
<td>TENSE pres</td>
</tr>
</tbody>
</table>

generalizations(E):

<table>
<thead>
<tr>
<th>HEAD</th>
<th>elaboration</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUCLEUS</td>
<td>presentation(generalization(E,S))</td>
</tr>
<tr>
<td>SATELLITE</td>
<td>properties(S)</td>
</tr>
<tr>
<td>VIEWS</td>
<td>BEHAVIORAL no</td>
</tr>
<tr>
<td></td>
<td>COMPUTATIONAL no</td>
</tr>
</tbody>
</table>

instantiation(E):

<table>
<thead>
<tr>
<th>HEAD</th>
<th>elaboration</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUCLEUS</td>
<td>background(E)</td>
</tr>
<tr>
<td>SATELLITE</td>
<td>properties(E)</td>
</tr>
</tbody>
</table>

justifications(E):

<table>
<thead>
<tr>
<th>HEAD</th>
<th>evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUCLEUS</td>
<td>justifications(E)</td>
</tr>
<tr>
<td>SATELLITE</td>
<td>justification(E)</td>
</tr>
<tr>
<td>VIEWS</td>
<td>INTERNAL yes</td>
</tr>
</tbody>
</table>

justifications(E):

<table>
<thead>
<tr>
<th>HEAD</th>
<th>justify</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUCLEUS</td>
<td>presentation(modeling_concept(E))</td>
</tr>
<tr>
<td>SATELLITE</td>
<td>{justification(E)}</td>
</tr>
<tr>
<td></td>
<td>{rationale(E)}</td>
</tr>
</tbody>
</table>
properties(E):

- HEAD: nil
- NUCLEUS: structures(E)
- VIEWS:
  - INTERNAL yes
  - EXTERNAL no

properties(E):

- HEAD: elaboration
- NUCLEUS: properties(E)
- SATELLITE: all(presentation(syntax(E,S)))
- PRECONDITION: constraint_concept(E)
- VIEWS: RESTRICTIVE yes

properties(E):

- HEAD: interpretation
- NUCLEUS: properties(E)
- SATELLITE: semantics(E)
- VIEWS: RESTRICTIVE no

properties(E):

- HEAD: possession
- NUCLEUS: description(E)
- SATELLITE: all(possessions(E))
- SALIENCE: TOPIC E
- VIEWS: EXTERNAL yes

properties(E):

- HEAD: description(E)
- NUCLEUS: semantics(E)
- all(extension(E,..))

relationships(E):

- HEAD: elaboration
- NUCLEUS: presentation(relationship(E,R))
- SATELLITE: properties(R)

semantics(E):

- HEAD: evidence
- NUCLEUS: presentation(denotation(E,..))
- SATELLITE: all(values(E))
- VIEWS: RESTRICTIVE no

semantics(E):

- HEAD: nil
- NUCLEUS: presentation(denotation(E,..))
- all(values(E))
values(E):
[ HEAD nil 
  NUCLEUS presentation(extension(E,...)) ]

values(E):
[ HEAD elaboration 
  NUCLEUS presentation(extension(E,V)) 
  SATELLITE properties(V) 
  VIEWS [ INTERNAL yes ] ]

The following precondition predicates are defined:

actor(C,F,E) C is a dynamic concept that produces conceptual result E by means of formula or act F. Or, C can be an instance of a dynamic concept that produces instance result E by means of instantiation F of a formula or an act.
closest(R,T,P) Relation R(P,E) holds, in which E is either T or T's nearest previous or abstracted dynamic concept (following chains of succeed and trigger relations backwards from T, or chains of generalization and aggregation relations upwards) that is associated with an R relation.
concept(C) C is a DSM concept or a specialization of a DSM concept.
conceptual_level(T,C) If C is at the instance level, T is the type of C. If C is at the conceptual level, T is the same as C.
constraint_concept(E) E is a constraint concept.
decomposition(E,P) P is a part or a specialization of E.
dynamic_concept(E) E is a dynamic concept.
generalized_actor(G,F,E) Same as actor(C,F,E), except for that G is a generalization of C or an aggregate including C.
in_formula(F,S) The members of set C are the concepts used in formula F.
in_relation(E,C) The members of set C are the concepts included in relation E.
in_illustration(C) It is feasible to generate a view illustrating and including the elements in set C.
relation(C) C is a DSM relation.
type(T,C) C is an instance of T.
Bibliography


