Executable Conceptual Models in Information Systems Engineering

Geir Willumsen

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

In this thesis, we address the problem of supporting validation of user requirements to information systems. We are concerned with conceptual modeling of complex systems. Two major contemporary problems with such modeling are inadequate languages and the lack of support for model validation. We review previous research on validation, in particular work related to executable conceptual modeling languages.

The contributions of this thesis are organized in a modeling cycle framework, which includes conceptual modeling, model translations to executable representations, model executions, and explanations of the observed behavior. Translations are specified in a rule-based notation, which lets source and target patterns be specified at the metamodel level. In particular, we exploit the rule-based language to generate prototypes from conceptual models. The executable target language is chosen on the basis of the semantics of the source language. We have developed an expressive, yet compact internal language which covers a wide range of conceptual modeling languages, and hence lets translations be easily specified. The internal language is based on an analysis of existing languages. Hierarchical structure plays an important role, both for the modeling of static and dynamic system aspects. The static aspects are modeled using constructs similar to those found in semantic datamodels, with an additional construct for the modeling of complex constraints. The dynamic aspects are modeled by primitives for external events and dynamic laws, which prescribe how state changes take place. Dynamic laws may be organized in hierarchies. The semantics of the control structures in these hierarchies may be specified declaratively, e.g. non-determinism and concurrency can be expressed.

In order to explain model behaviors, traces are stored during execution. These may be accessed directly using a trace query language, which provides different views into the trace. A connection to a general explanation generator developed in a companion thesis has also been established through these queries. Users may ask questions about the behavior of a conceptual model. Some of the information included in the explanation is fetched from execution traces.

Prototypes for the translation, execution and tracing components of the framework have been developed. An integration with the PPP CASE environment has been proposed. Due to the generality of our approach, we see a potential of integrating it in meta environments which offer language customization. Our proposal is evaluated against stated requirements and the state of the art.
Preface

This thesis is submitted to the Norwegian Institute of Technology for the doctoral degree “doktor ingeniør”. The work has been supervised by Professor Arne Sølvberg, and carried out at the Information Systems Group, Faculty of Electrical Engineering and Computer Science, the Norwegian Institute of Technology, The University of Trondheim, Norway, from January 1990 to November 1993. From February to August 1992, I had a research stay at the Computer Science Engineering Department, the University of Texas at Arlington, USA.

During this work, I have been employed as a researcher at SINTEF, and as a research assistant at the Norwegian Institute of Technology for short periods. From February 1991 to September 1993 I had a scholarship from the Royal Norwegian Council for Scientific and Industrial Research (NTNF).

Acknowledgements

There are a number of people to be thanked for their contributions to this work. In particular, I would like to mention the following people:

- Professor Arne Sølvberg for his continuous encouragement and constructive criticism, and for giving me the opportunity to spend four good years in his group.

- Fellow members of the Information Systems Group for an inspiring working environment, constructive criticism on drafts of this thesis, and a nice social atmosphere. In particular, I am grateful to Jon Atle Gulla and Odd Ivar Lindland for many interesting discussions, and for the good cooperation we had in writing research papers.

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

- Astrid for all her support, encouragement, and understanding.

Trondheim, November 1993

Geir Willumsen
# Contents

## I Introduction

1 Introduction ................................. 1
   1.1 Research Questions .......................... 3
   1.2 Approach - Support of a Modeling Cycle .... 4
   1.3 Main Results ................................ 5
   1.4 Thesis Structure ............................ 6

## II State of the Art

2 Conceptual Modeling in IS Engineering ........... 9
   2.1 Requirements Modeling in IS Engineering ... 11
   2.2 Conceptual Modeling .......................... 15
   2.3 On the Expressiveness of CML’s ............... 25
   2.4 Model Validation .............................. 34
   2.5 Modeling Support in CASE Environments ....... 43
   2.6 Conceptual Modeling in PPP .................... 48
   2.7 Conclusions .................................. 58

3 Translation Facilities in CASE Environments ..... 59
   3.1 Syntax-Directed Tasks in CASE Environments 59
   3.2 Requirements to Translation Facilities ....... 61
   3.3 General Translation Principles ............... 63
   3.4 The Parser Generator YACC ................. 65
   3.5 Syntax-Directed Experts in POPART .......... 66
   3.6 Translation Facilities for Phase Integration 68
   3.7 Conclusions .................................. 71

4 Executable CML’s in CASE Environments .......... 73
   4.1 Prototyping Languages .......................... 73
   4.2 Impacts on the System Lifecycle ............... 74
   4.3 Execution by Interpretation and Translation . 76
   4.4 Execution Mechanisms .......................... 78
   4.5 Requirements to CASE Environments Supporting Executable CML’s 79
   4.6 Some Selected Languages and Environments .... 80
   4.7 Conclusions .................................. 90

5 Execution Tracing in CASE Environments ........ 93
   5.1 On the Use of Execution Traces ............... 93
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2</td>
<td>Requirements to Tracing Components</td>
<td>94</td>
</tr>
<tr>
<td>5.3</td>
<td>General Tracing Principles</td>
<td>95</td>
</tr>
<tr>
<td>5.4</td>
<td>Principles for Explaining Model Behavior</td>
<td>97</td>
</tr>
<tr>
<td>5.5</td>
<td>Example Program Tracers</td>
<td>103</td>
</tr>
<tr>
<td>5.6</td>
<td>Conclusions</td>
<td>106</td>
</tr>
<tr>
<td>III</td>
<td>Contributions</td>
<td>107</td>
</tr>
<tr>
<td>6</td>
<td>Support for an Integrated Approach to Conceptual Model Validation</td>
<td>109</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>109</td>
</tr>
<tr>
<td>6.2</td>
<td>Support for the Modeling Cycle</td>
<td>110</td>
</tr>
<tr>
<td>7</td>
<td>Constructs for Executable CML's - ECML</td>
<td>113</td>
</tr>
<tr>
<td>7.1</td>
<td>Introduction</td>
<td>113</td>
</tr>
<tr>
<td>7.2</td>
<td>A Unified Metamodel and Derivation of Language Constructs</td>
<td>115</td>
</tr>
<tr>
<td>7.3</td>
<td>Constructs for Modeling of State Spaces</td>
<td>118</td>
</tr>
<tr>
<td>7.4</td>
<td>Constructs for Modeling of State Transitions</td>
<td>123</td>
</tr>
<tr>
<td>7.5</td>
<td>The Operational Semantics of ECML</td>
<td>133</td>
</tr>
<tr>
<td>8</td>
<td>A Rule Language for Translation Specification and Implementation</td>
<td>147</td>
</tr>
<tr>
<td>8.1</td>
<td>Introduction</td>
<td>147</td>
</tr>
<tr>
<td>8.2</td>
<td>The Form of Translation Rules</td>
<td>148</td>
</tr>
<tr>
<td>8.3</td>
<td>Rule Semantics and Control of Rule Execution</td>
<td>153</td>
</tr>
<tr>
<td>8.4</td>
<td>Use of TRL - Three Examples</td>
<td>156</td>
</tr>
<tr>
<td>8.5</td>
<td>On an Implementation of TRL</td>
<td>165</td>
</tr>
<tr>
<td>9</td>
<td>Execution Traces and Explanation of Model Behavior</td>
<td>169</td>
</tr>
<tr>
<td>9.1</td>
<td>Introduction</td>
<td>169</td>
</tr>
<tr>
<td>9.2</td>
<td>An Expanded System History Definition</td>
<td>170</td>
</tr>
<tr>
<td>9.3</td>
<td>A General Trace Schema</td>
<td>171</td>
</tr>
<tr>
<td>9.4</td>
<td>Interface to a Model Executor: The Reported Events Handler</td>
<td>174</td>
</tr>
<tr>
<td>9.5</td>
<td>A Query Language to Retrieve Trace Information</td>
<td>174</td>
</tr>
<tr>
<td>9.6</td>
<td>Understanding Executions Through Trace Queries</td>
<td>180</td>
</tr>
<tr>
<td>9.7</td>
<td>Finding Fragments of Explanations</td>
<td>188</td>
</tr>
<tr>
<td>10</td>
<td>A Description of Component Prototypes</td>
<td>193</td>
</tr>
<tr>
<td>10.1</td>
<td>Some Technical Considerations</td>
<td>193</td>
</tr>
<tr>
<td>10.2</td>
<td>A TRL Interpreter</td>
<td>193</td>
</tr>
<tr>
<td>10.3</td>
<td>An ECML Interpreter</td>
<td>194</td>
</tr>
<tr>
<td>10.4</td>
<td>A TRQL Interpreter</td>
<td>211</td>
</tr>
<tr>
<td>11</td>
<td>Integration with PPP</td>
<td>217</td>
</tr>
<tr>
<td>11.1</td>
<td>Introduction</td>
<td>217</td>
</tr>
<tr>
<td>11.2</td>
<td>Translation from PhM</td>
<td>218</td>
</tr>
<tr>
<td>11.3</td>
<td>Translation from PrM</td>
<td>221</td>
</tr>
<tr>
<td>11.4</td>
<td>Translation from PLD</td>
<td>225</td>
</tr>
<tr>
<td>11.5</td>
<td>Object-Oriented, Real-Time Systems, and ECML</td>
<td>230</td>
</tr>
<tr>
<td>11.6</td>
<td>Concluding Remarks</td>
<td>232</td>
</tr>
</tbody>
</table>
IV Evaluation and Conclusions 233

12 Evaluation 235
12.1 Evaluation with Respect to Stated Requirements 235
12.2 Comparison with Existing Work 239
12.3 Experiences with the Prototypes 243
12.4 Integration with PPP 244
12.5 Integration with Meta CASE Environments 244

13 Conclusions 245
13.1 Main Results 245
13.2 Further Work 246

V Appendices 249
A Type Definitions in ECML 251
B ECML Grammar 255
C TRL Grammar 259
D Translation Rules for PPP to ECML Translation 261
E TRQL Grammar 283
F TRQL Queries in Relational Calculus 287

Bibliography 293
List of Figures

1.1 The modeling cycle framework .............................................. 5
1.2 The validation support architecture ........................................ 6
1.3 Thesis structure (chapter numbers in parentheses) ..................... 7

2.1 A simplistic view of information systems engineering .................. 12
2.2 The datamodel used for metamodelling of CML’s ......................... 24
2.3 Portions of a metamodel for an executable DFD ......................... 25
2.4 A metamodel of Wand and Weber’s ontology .......................... 27
2.5 A metamodel corresponding to the methodology framework .......... 28
2.6 The unified model in AMADEUS ........................................... 29
2.7 The metamodel of GDR ................................................... 30
2.8 Excerpts of the ARIES metamodel ...................................... 31
2.9 A general semantic datamodel ............................................ 32
2.10 A comparison of the unified metamodels ............................... 34
2.11 Verification and validation of conceptual models ...................... 35
2.12 Alternative ways to develop prototypes from conceptual models ... 37
2.13 A simple architecture for meta environments ........................ 47
2.14 A PhM model of the banking system .................................... 50
2.15 Portions of a PrM model for the banking system ....................... 52
2.16 A decomposition of transaction processing ............................ 53
2.17 A PLD describing process P1.2 ........................................ 54
2.18 The major constructs of the PPP metamodel ........................... 57
2.19 Syntax for conditions and expressions in PLD .......................... 58

3.1 The architecture of a general translation facility ...................... 63
3.2 The general structure of syntax-directed experts in POPART .......... 66
3.3 Control of syntax-directed experts in POPART ........................ 67
3.4 A syntax-directed expert in POPART ................................... 68
3.5 The IPSEN framework .................................................... 69

4.1 Prototyping in the Waterfall model ...................................... 75
4.2 Operational specification paradigm ...................................... 76
4.3 The bank model in Statecharts ......................................... 81
4.4 The bank model in ARIES’ Gist language ............................... 83
4.5 The bank model in TEMPORA .......................................... 85
4.6 The bank model in BNM ................................................ 86
4.7 The bank model in TaxisDL ............................................. 88
4.8 The bank model in eG-Nets ............................................. 89

5.1 A general architecture of a tracing component .......................... 96
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2</td>
<td>Questions to execution of the bank model</td>
<td>98</td>
</tr>
<tr>
<td>5.3</td>
<td>A general architecture for explanation generation</td>
<td>99</td>
</tr>
<tr>
<td>5.4</td>
<td>Answer to an event question</td>
<td>100</td>
</tr>
<tr>
<td>5.5</td>
<td>Example tables and queries in Snodgrass' approach</td>
<td>104</td>
</tr>
<tr>
<td>5.6</td>
<td>Another database approach to program debugging</td>
<td>106</td>
</tr>
<tr>
<td>6.1</td>
<td>Components for an integrated approach to model validation</td>
<td>110</td>
</tr>
<tr>
<td>7.1</td>
<td>A new unified metamodel</td>
<td>116</td>
</tr>
<tr>
<td>7.2</td>
<td>The document class Monthly_statement</td>
<td>121</td>
</tr>
<tr>
<td>7.3</td>
<td>Parts of a law structure for the banking system</td>
<td>130</td>
</tr>
<tr>
<td>7.4</td>
<td>Transitions between the sets $E$, $C$, and $N$ during execution of an ECML model</td>
<td>135</td>
</tr>
<tr>
<td>7.5</td>
<td>Division of sublaws made by the candidate and select functions</td>
<td>142</td>
</tr>
<tr>
<td>8.1</td>
<td>Basic-construct rules and construct-property rules</td>
<td>149</td>
</tr>
<tr>
<td>8.2</td>
<td>Execution of translation rules</td>
<td>154</td>
</tr>
<tr>
<td>8.3</td>
<td>Execution of translation programs</td>
<td>155</td>
</tr>
<tr>
<td>8.4</td>
<td>Execution of basic-construct rules</td>
<td>155</td>
</tr>
<tr>
<td>8.5</td>
<td>Execution of construct-property rules</td>
<td>156</td>
</tr>
<tr>
<td>8.6</td>
<td>Execution of subgoals</td>
<td>156</td>
</tr>
<tr>
<td>8.7</td>
<td>The Petri net metamodel</td>
<td>157</td>
</tr>
<tr>
<td>8.8</td>
<td>A simplified SQL</td>
<td>160</td>
</tr>
<tr>
<td>8.9</td>
<td>Translation of PrM ports to initial PLD structures</td>
<td>163</td>
</tr>
<tr>
<td>8.10</td>
<td>A sketch of an implementation of translation specifications</td>
<td>166</td>
</tr>
<tr>
<td>9.1</td>
<td>Parts of a system history for the bank model</td>
<td>171</td>
</tr>
<tr>
<td>9.2</td>
<td>Parts of an execution trace</td>
<td>173</td>
</tr>
<tr>
<td>9.3</td>
<td>A process model to illustrate explanation of obtained results</td>
<td>181</td>
</tr>
<tr>
<td>9.4</td>
<td>Understanding applied laws in a process model</td>
<td>184</td>
</tr>
<tr>
<td>9.5</td>
<td>A process model to illustrate concurrency</td>
<td>185</td>
</tr>
<tr>
<td>9.6</td>
<td>A process model to illustrate explanation of event sequences</td>
<td>186</td>
</tr>
<tr>
<td>9.7</td>
<td>A process model to illustrate explanation of state law evaluations</td>
<td>187</td>
</tr>
<tr>
<td>9.8</td>
<td>An example plan operator</td>
<td>189</td>
</tr>
<tr>
<td>9.9</td>
<td>The interface between EML and the trace queries</td>
<td>191</td>
</tr>
<tr>
<td>9.10</td>
<td>A deep explanation</td>
<td>192</td>
</tr>
<tr>
<td>10.1</td>
<td>Main loop of interpreter</td>
<td>196</td>
</tr>
<tr>
<td>10.2</td>
<td>Execution cycle of interpreter</td>
<td>196</td>
</tr>
<tr>
<td>10.3</td>
<td>Searching for applicable simple laws</td>
<td>198</td>
</tr>
<tr>
<td>10.4</td>
<td>The generic structure of apply</td>
<td>199</td>
</tr>
<tr>
<td>10.5</td>
<td>The generic structure of terminate</td>
<td>199</td>
</tr>
<tr>
<td>10.6</td>
<td>A simple execution example</td>
<td>200</td>
</tr>
<tr>
<td>10.7</td>
<td>Evaluation of functional expressions</td>
<td>201</td>
</tr>
<tr>
<td>10.8</td>
<td>Evaluation of sentences</td>
<td>202</td>
</tr>
<tr>
<td>10.9</td>
<td>Evaluation of queries</td>
<td>202</td>
</tr>
<tr>
<td>10.10</td>
<td>Insertion of a subclass instance</td>
<td>203</td>
</tr>
<tr>
<td>10.11</td>
<td>Deletion of class instances</td>
<td>204</td>
</tr>
<tr>
<td>10.12</td>
<td>Updates of variables and class instances</td>
<td>205</td>
</tr>
</tbody>
</table>
10.13 Decision tables for each of the parts of the procedure apply .......... 207
10.14 Generation of an interpreter procedure from the SEQ specification .... 208
10.15 Step-by-step execution in the ECML interpreter ....................... 209
10.16 Breakpoints in the ECML interpreter ................................. 209
10.17 Retrieving information about applied dynamic laws ................... 212
10.18 A function which performs subqueries of law application queries ....... 213
10.19 An algorithm for answering queries in the refers view ................. 214
10.20 An algorithm for answering queries in the state component view ....... 214
10.21 An algorithm for answering queries from the law context view ....... 215

11.1 Data and control integration with PPP .................................. 218
11.2 Parts of the bank model and its representation in ECML ................. 219
11.3 Translation of decomposed processes to ECML ........................ 223
11.4 Translation from PLD structures to ECML law structures ............... 226
11.5 Translation of loop constructs ........................................... 230
11.6 A Statechart transition in ECML ........................................ 232
# List of Tables

2.1 The major properties of some existing meta environments ........................ 48

7.1 Description of relationships between static constructs .......................... 117
7.2 Description of relationships between dynamic constructs ....................... 117
7.3 Explanation of cand<sub>sr</sub> specifications ........................................ 141
7.4 Specification of the cand<sub>sr</sub> function ............................................ 141
7.5 Explanation of select<sub>sr</sub> specifications ........................................... 142
7.6 Specification of the select<sub>sr</sub> function ........................................... 142
7.7 Explanation of addC<sub>sr</sub> specifications ............................................ 143
7.8 Specification of the addC<sub>sr</sub> function ............................................ 143
7.9 Explanation of taddC<sub>sr</sub> specifications ............................................ 144
7.10 Specification of the taddC<sub>sr</sub> function ........................................... 144
7.11 Explanation of term<sub>c</sub> specifications ............................................ 145
7.12 Specification of term<sub>c</sub> ............................................................... 145
7.13 Sublaw relationships in PPP ............................................................. 146
7.14 Other examples of sublaw relationships .............................................. 146

9.1 Relationships referenced in plan operators ........................................... 190
Part I

Introduction

In Chapter 1, we state the problems addressed in the thesis, we describe the approach taken to overcome the identified problems, the major contributions of the thesis, and the thesis structure.
Chapter 1

Introduction

In March 1993 a large Norwegian government institution announced that it had terminated its contract with a software company, which was supposed to develop a new nationwide information system supporting the majority of the institution’s activities. According to institution officials, the reason for so doing was a series of delays with respect to the agreed-upon schedule of deliverables. It further estimated that these delays would cost the institution approximately 50MNOK for 1993 alone. It was decided to continue developments in-house, using own software professionals. A representative of the contractor explained the problems by insufficient knowledge of system requirements.

This and numerous similar cases clearly illustrate the severe consequences from the inability to cope with the communication problem (Sølvberg and Kung[81]) in the development of information systems. More specifically, it exemplifies the importance of validation of system requirements at early stages of the system lifecycle.

The research community has responded to this problem in many distinct ways, of which two are central to this thesis. First, a number of conceptual modeling techniques including languages (CML's) and methods, have been developed in order to model systems using constructs close to the users’ world. It is assumed that the user-oriented/problem-oriented nature of CML’s enhances communication and hence the possibility to validate requirements (Borgida [18], Bubenko [21], Roman [104]).

Second, a number of rapid prototyping approaches have emerged, supporting the development of system prototypes (mainly user interface and functional prototypes) using a powerful, yet inefficient prototyping language (Agresti [2], Carey [23], Davis [34], Gomaa [46]). Special benefits are expected if the two approaches are combined into executable conceptual modeling languages. Such languages can be directly interpreted, or translated to other executable languages, and hence provide early feedback on the dynamic properties of the models. This combined approach is found in the PPP environment [48], as well as in e.g. STATEMATE [54], ARIES [62], BNM [68], REMORA [77], TEMPORA [78], and PAISLey [141], to mention a few.

CASE environments have been developed to support modeling and prototyping, but so
far the support for executable CML's is mostly found in research systems. The cost of supporting languages suitable to the users and developers involved, and to the problem domain, has led to the development of meta CASE environments. These let the environments be tailored to specific preferences, to a differing degree of sophistication. Some offer metamodeling facilities which include specification of CML's. Others rely on a set of semantically overlapping languages to be chosen from, for a particular project or situation. Common to most meta environments is the lack of prototyping support.

1.1 Research Questions

While conceptual modeling and rapid prototyping are seen by many as promising approaches to the validation problem, the costs of developing adequate support environments are high. This is particularly so when the modeling languages are expressive, with complex control structures, and cover many aspects of the system being modeled. In many cases, transformations and translations are needed on the conceptual model level even before they can be interpreted and translated into an executable representation. The cost of developing support for these activities also hampers the modifications and/or introductions of new modeling constructs and languages. The first research question is therefore:

*How can we build CASE environments which include executable CML's?*

While the execution of conceptual models reveal their dynamic properties, the executing model appears at the user interface as a 'black box'. This is because only the *external behavior* can be observed, i.e. inputs required and inputs given, and outputs resulting from execution. For complex models, it may become difficult to understand why particular events took place. However, such understanding is necessary in case unexpected behavior is observed. A full understanding requires *explanations* which link the external behavior to the way the model is built and to the particularities of the current execution instance. This has been recognized in expert systems research, and hence many expert systems have facilities which offer users the possibility to ask questions about the problem solving activities and the conclusions produced, see e.g. Chandrasekaran [24]. This has also been recognized in programming environments research, where various facilities for tracing program executions and inspecting execution traces have been developed, even if this is rarely seen in commercial CASE environments. Execution of complex conceptual models needs similar support. The second research question can therefore be stated as follows:

*How can we explain observed behavior during execution of conceptual models?*

Combining solutions to these two research problems, we may form an integrated approach to conceptual model validation.
1.2 Approach - Support of a Modeling Cycle

We suggest an integrated modeling cycle as a framework for our research. The modeling cycle consists of four activities as shown in Figure 1.1.

The modeling cycle suggests a way of working in which conceptual models are developed, translated/transformed to other models and finally to an executable representation, subsequently executed. The observed behavior is validated by comparing with expected behavior and explaining the observed events. The validation results are used to modify the conceptual model, and hence to start another cycle.

The research questions suggest the activities of translation/transformation, execution, and validation by explanation to be focused. For the evaluation of our approach, the PPP experimental CASE environment [48] provides an excellent testbed.

For translation and transformation, we strive for general mechanisms in which references can be made to a CASE repository containing source and target models. It is important that these mechanisms offer expressive translations and transformations. If these mechanisms are made language independent, their generality is enhanced, and they can be used for other purposes than generating executable representations.

In some cases the constructs of a CML can be translated to statements of a high level programming language quite easily, though not with an optimal operational performance. However, many CML's contain constructs which are not so easily mapped to such languages. To deal with this problem, we propose a set of modeling language constructs which are executable. By careful choice of these constructs, we can offer translation to executable representations for a wide range of CML's.

For validation by explanation, we partly rely on the work presented in a companion thesis by Gulla [47] where a general explanation generator has been developed. This generator accepts questions from users, and returns natural language explanations which refer to the events which have occurred during execution. To support explanation of model behavior, we propose to record events in an execution trace. We provide a trace query language which offers different views into the execution trace. The trace query language may also
be used directly by system developers for rapid explanations of behavior. The language may also be tailored to a particular CML, so that queries can be made more naturally, using the constructs of the CML. To this end, we can also use the proposed translator.

1.3 Main Results

Support for the modeling cycle has been realized by the environment shown in Figure 1.2. We have investigated both theoretical and practical aspects of the validation support architecture. The theoretical achievements are related to development of three languages for each of the focused activities in the modeling cycle. The practical aspects relate to development of interpreter prototypes for these languages, as well as to integrating these components in a CASE environment. The main results of this thesis are:

- For model translations, we have defined a rule based language TRL to be used at a high level of abstraction. We have also developed an interpreter which can execute such rules. The applicability of the approach has been demonstrated through specification of various translations in the PPP environment. We also propose how to transfer the mechanisms to other CASE environments.

- To ease the translation from CML's to executable representations, ECML which is an executable CML is proposed. A major concern has been language expressiveness and generality. ECML includes features of semantic data models, and relies on a typed first order logic for specification of constraints. The language also provides an extensible set of hierarchical control structures to be used for modeling of system dynamics. We have implemented an interpreter for the language, and demonstrate how PPP models can be translated to ECML constructs. Usually, the price to be paid for language expressiveness is less efficient computation. This has however been recognized as an acceptable feature of prototyping languages.
1.4 Thesis Structure

For the support of model behavior explanations, we propose a schema for execution traces, based on the general notion of system history. We have identified typical explanation situations, and based on these we propose a trace query language TRQL. The queries form the interface to the explanation generator. We show how the language can be used to provide rapid explanations from a potentially in comprehensible execution trace. Also for this language an interpreter has been developed, and routines for reporting events from an executor have been implemented.

The three languages and their interpreters form together with the modeling support of PPP an integrated approach to conceptual model validation. Our focus has been on designing an environment for validation, rather than to provide detailed methodological guidelines for how it should be used. These issues remain topics for further research.

1.4 Thesis Structure

The structuring principle of the thesis is based on the modeling cycle framework, as shown in Figure 1.3.

Part I: Introduction The first part contains this introductory chapter.

Part II: State of the art This part (Ch. 2-5) contains the state of the art analysis of the thesis. We describe the role of conceptual modeling and its relationships to general problems in information systems engineering in Chapter 2, in particular the relationships to the validation problem. In Chapter 3 we present some existing approaches to model translation. An overview of executable conceptual modeling languages is presented in Chapter 4, and their implications for the information systems lifecycle is discussed. In Chapter 5 we treat various published approaches to explanations of model behavior.
Part III: Contributions In this part (Ch. 6-11) we present the novel contributions of the thesis. First we describe an architecture of a validation support environment in Chapter 6. The three languages ECML, TRL, and TRQL, their requirements and usage are presented in the next three chapters. In Chapter 10 we briefly describe the design and implementation of the interpreter prototypes. Finally, in Chapter 11 we demonstrate the feasibility of our approach by integrating the different components into the PPP environment, and by specifying a complete translation from the current PPP language to ECML.

Part IV: Evaluation and Conclusions In Chapter 12 we evaluate our results and compare them with the state of the art, while Chapter 13 offers conclusions and suggestions for further work.

Part V: Appendices and Bibliography The appendix includes the grammar for the three languages TRL, ECML, and TRQL, as well as translation rules from PPP to ECML, and the type definitions of ECML. The retrieved data from queries expressed in TRQL are also specified.
In Chapter 2, we concentrate on conceptual modeling in information systems engineering. We take up issues like system requirements, language expressiveness, validation techniques and tool support.

In Chapter 3, translation facilities in CASE environments are described.

In Chapter 4, we go in more detail on executable conceptual modeling languages, and on the tool support required.

In Chapter 5, execution tracing is described. In particular, its role in explanation generation systems is emphasized.
Chapter 2

Conceptual Modeling in IS Engineering

In this chapter we stress the importance of requirements modeling in information systems engineering. We define some basic terminology related to conceptual modeling, and list properties such languages should have to avoid errors in modeling, and to help in error detection. Language specification is treated briefly, and we analyze the expressiveness of CML’s. We then focus on tool support for conceptual modeling, including meta-environments which let languages and tools be specified and hence provide a higher degree of flexibility. The problem of model validation is then discussed, and different validation techniques are presented. We illustrate some major modeling issues through a presentation of the PPP CASE environment. Finally, the conclusions give further motivations for our work.

2.1 Requirements Modeling in IS Engineering

A Simplistic View of IS Engineering

*Information systems* (IS) are integrated in organizations. They help organizations store, retrieve, and process information in their strive to reach their goals. An information system may consist of both manual and automated parts. Parts which are automated through the use of computer software and hardware, are called *computerized information systems*. Many hold the opinion that computerized information systems are models or representations of real world systems, e.g. as pointed out by Borgida [18] and Wand [128]. In this sense, the information system is recognized as representing the states and state transitions found in the real world system, be it a concrete or conceptual one.

*Information systems engineering* is the discipline for the development of information systems. It involves the management of tasks, products, resources, and people during the development of a system.
Figure 2.1 shows a simplistic view of IS engineering. The decision to develop a computerized information system is taken based on goals, policies etc., of an organization. The developed system will be integrated with the existing information system. For managerial purposes, the development process has traditionally been divided into a number of phases. Each phase ends with a product which forms the basis for the next phase, the final product being the installed system. There are many lifecycle models which offer different guidelines for how the phases and their deliverables should be managed. Common to most of these are separate phases for analysis, design, and implementation. Central to this thesis is the analysis phase, in which properties and constraints of the system to be, is determined. It is widely agreed that obtaining correct requirements is vital for the successful development of an information system. From this recognition, requirements engineering has emerged as a separate discipline for requirements capture and representation.

In this thesis, a central premise is that the choice and use of languages and methods are vital for a successful development project. Languages are used to express the products of development phases, while methods provide guidelines for how the languages should be used to arrive at high quality products.

Information systems development involves humans. For this thesis, it is sufficient to divide the roles that humans play into four different categories. First, the customer is a representative of the organization for which a system is to be developed. The person(s) in this role has the authority to take the necessary decisions on behalf of the organization. A domain expert has detailed knowledge of the inner workings of the real world system be-
2.1. Requirements Modeling in IS Engineering

...ing studied, i.e. the application domain. An end-user is a person which uses the developed system in her work. We may group these three categories using the term user. Finally, a developer is any person involved in the creation of products for any phase of development, e.g. project managers, analysts, designers, and programmers.

To sum up, our simplistic view of IS engineering includes

- the management of a process, i.e. the methods used in development phases and the associated tasks to be performed by developers,

- the management of products and resources consumed and produced by these tasks, and

- the coordination of humans in the assignment of tasks and responsibilities to carry out the development project.

The Role of Requirements in IS engineering

There is no general agreement on what requirements are and how they should be captured and represented. While some hold the opinion that the requirements should be black-box descriptions of the input/output behavior of the system to be developed e.g. Berzins in [12], others emphasize that they should rather represent a model of a real world system, e.g. Borgida [18], Olle et al. [93], and Wand [128]. Requirements have the dual roles of on the one hand to serve as a basis for agreement with the customer on what properties the system should have, and on the other hand be a basis for designing the system. As noted by Dubois [39] and many others, the languages used should hence preferably be problem and user oriented, while at the same time be precise enough in their syntax and semantics to provide a basis for further development, and facilitate automatic analysis.

The discussions on the 'what/how'-issue illustrate the disagreement on what requirements are. Many agree that requirements should express what a system should do, and not how it should be achieved. However, there are diverse opinions on what level of abstraction the 'what' should be given, and still be regarded as a requirement. Davis illustrates this dilemma in [33]. The 'what' and 'how' aspects are intertwined, so that what is a part of 'how' on one level is equal to a 'what' on the immediate lower level. Davis takes the view that requirements should only express the external behavior of the system to be developed. Others rather make a distinction of problem-oriented and implementation-oriented models, the former representing the requirements to the system, e.g. Balzer et al. [8, 9], and Zave [141, 142].

Requirements can be divided into the functional and non-functional categories. Functional requirements describe the essential input/output behavior, relevant internal states, and state transitions. This corresponds to what Wand calls the deep structures of a system [128]. Non-functional requirements are constraints that need to be satisfied by the developed system, including interface constraints, performance constraints, reliability constraints,
economic constraints etc. In this thesis we focus on functional requirements only, although the importance of non-functional requirements are certainly recognized.¹

**Causes and Effects of Requirements Errors in IS engineering**

Errors in requirements are most likely to occur, and it is important to detect them at an early stage to avoid the propagation of errors to later phases. In order to achieve this, an understanding of the nature of errors is necessary. We may divide errors into five different groups. **Syntactical errors** occur when the requirements are not expressed according to the syntax of the language used to express them. **Incorrect facts** are requirements stated incorrectly with respect to users' needs or with respect to the real world. **Incompleteness** in requirements means that some information is missing. This may either be because a construct is not completely described according to the language, or because some requirements are simply missing. **Inconsistency** means that there are conflicting requirements, i.e. requirements which can not be satisfied simultaneously. **Ambiguity** occurs when a requirement has multiple interpretations. Note that some errors are unavoidable and should be allowed temporarily, e.g. incompleteness. For a completely finished product, however, there should be as few errors as possible.

There are numerous reasons for why such errors occur. Some important causes are listed below:

- The inherent complexity of information systems makes requirements capture and representation difficult, as pointed out by Brooks [20], Rittel [81] and many others. As we have observed, information systems are models of the real world, hence the complexity of the real world is transferred to the information system representing it.

- Requirements capture involves communication across different worlds. People from the users' world and the developers' world are likely to have different conceptions of the world, and use different languages to describe these. Hence, the communication between them is hampered. In [81], Sølvberg and Kung recognize the communication problem to be the major problem in IS engineering.

- The importance of obtaining correct requirements has for a long time not been recognized to the extent that sufficient resources have been allocated to the task.

- Users are often uncertain of what they want, partly because they are unfamiliar with computers. It is often the developers' task to analyze a problem, and then propose what systems should be made to overcome it. With increasing utilization of computers, this problem is likely to be reduced. A problem associated with this is that requirements may easily change.

- An inappropriate language to express requirements can hamper the communication with users, reduce the ability to express all relevant information, and make automatic analysis of requirements impossible. There is a tendency to make languages more

¹In fact, our research group has been addressing performance requirements in particular [97].
formal and complex, which gives certain advantages to model analysis, but also leads to problems in model comprehension.

- Lack of good methods and requirement capturing techniques are likely to result in errors. Methods should prescribe the steps to be taken, while at the same time be flexible enough so as to exploit the creativity of developers. Contemporary capturing techniques are often based on more or less structured interviews. Additional techniques are clearly needed, as recognized in research on knowledge-based systems development.

- Lack of appropriate tool support for system development, in particular for requirements capture and representation. We will discuss the problems of current CASE environments in a later section.

The quality of the requirements document is obviously of vital importance for the successful development of a system. Some of the problems cited for IS engineering e.g. by Sølvberg [81] and Bubenko [21] can be related to erroneous requirements. For instance, projects being behind schedule, with overrunning budgets may be caused by a lack of understanding of the 'real' requirements. Perhaps even more serious, the developed system will most certainly not satisfy the needs of the customer, causing further delays and waste of money and time. Furthermore, erroneous requirements not detected during system development, are almost certainly found during use of the system, and result in increased maintenance costs.

We have recognized the importance of language in the development of information systems, and will later discuss in more depth the language properties deemed important for the ability to detect requirement errors in particular. We will also present various techniques to find errors, and explain how many of these depend directly on language properties. As discussed in the next section, we are particularly interested in a class of languages called conceptual modeling languages for the representation of functional requirements to information systems.

2.2 Conceptual Modeling

In this section, we give a general discussion of issues related to conceptual modeling. It is not our intention to give a survey of existing languages at this stage. Rather, in Chapter 4 we present some languages which are executable. Also, in the final section of this chapter, a presentation of the conceptual modeling languages of the PPP environment is given.

Language, Model and Modeling

For the purposes of this thesis, we may define a conceptual modeling language informally in the following way:
A conceptual modeling language (CML) is a language for creating abstract representations of relevant system phenomena.

There are four different aspects of this definition: Language, system, relevant phenomena, and abstract representation:

**Language**  As a language, it has a defined syntax, and a more or less well-defined semantics. The syntax determines the basic constructs which can be used, their properties, and legal relationships to other basic constructs, through a number of formation rules. These relationships make it possible to form more complex constructs in order model systems. The semantics assign meaning to the language constructs.

**System**  The systems to be modeled are real-world systems, existing and concrete systems, or conceptual systems existing only in humans’ minds. Systems are regarded as having basically two aspects: 1) A static aspect which covers the passive phenomena of systems, i.e. the systems’ state spaces, and 2) a dynamic aspect which covers the active phenomena, i.e. the ways state transitions happen for the passive phenomena as results of interaction with the systems’ environments. A conceptual modeling language should also be able to represent this environment and its interaction with the system of interest. By regarding the environments as systems as well, the same constructs can usually be used.

**Relevant phenomena**  What are then these phenomena which should be described for a system? As will be discussed below, different languages offer different perspectives of a system to be modeled. There exist numerous languages, in fact it seems that each research community with self-respect must develop its own. On the surface, it may seem as if there exist very different opinions on what constructs a language should have in order to be suitable for systems modeling. However, recent studies by e.g. Black et al. [13, 91], Hull et al. [60], and Lubars [80] indicate that the meanings of constructs from different languages are remarkably alike, and that many languages differ mostly at a syntactical level. Constructs used for modeling of static aspects include entities, objects, things, attributes, properties, relationships, state laws, and constraints, many of these with the same or very similar semantics. Constructs for modeling dynamic aspects include processes, activities, tasks, events, transitions, control processes etc.

**Abstract representations**  To be able to deal with the complexity of the real world, we need to create abstractions of it. It would be impossible and at least a waste of resources to model information systems at the level of atoms. Hence, the constructs used to describe systems should be at a sufficient level of abstraction, and hide away details not relevant for the study. As pointed out by Brooks [20], there is always the danger of oversimplifying the world. It is important that the appropriate abstraction level is chosen so that the relevant aspects and complexity of a system become apparent. A basic idea with conceptual modeling languages is that the constructs used should be suitable for user communication, so as to facilitate a representation of the user’s perceptions of the world. In that sense, one can talk about user-oriented and problem-oriented languages, used to model problem domains.
Two additional terms important for our work are conceptual model and conceptual modeling:

*A conceptual model is a description of a system expressed in a conceptual modeling language. Conceptual modeling is the activity of creating conceptual models.*

Note that according to this terminology, a datamodel should rather be termed a data modeling language. However, when talking in particular about datamodels, we will obey the established terminology of the field. Also note that the creation of conceptual models is guided by, or should at least be guided by, an associated method.

**The Roles of Conceptual Models**

In [67], Kung have identified four different roles for conceptual models in IS engineering:

- Conceptual models can represent models of systems in user-oriented and problem-oriented terms, and hence increase the developers’ understanding of them. The models can be descriptive, in the sense that they are used to represent and analyze existing systems. They can also be prescriptive, meaning that they model a system to be developed. The same model may be both descriptive and prescriptive, in the case that an existing system is going to be computerized directly as is. Prescriptive models actually embody the functional requirements to an information system.

- They serve as a common reference framework for communication with users. The user-oriented constructs provide a means for talking about systems in a language closer to the users’ professional languages. However, as we noted in the previous section, there are many obstacles to an efficient communication with customers, domain experts, and end-users.

- They serve as a basis for design and implementation phases, and for testing of designs and implementations to check if they comply with the requirements.

- They provide documentation which is useful for later maintenance of systems, as they represent the functional requirements to the existing systems. Additional or changed functionality can then be related directly to a model of an existing system. Also, the model provide abstract views of systems, and it is hence much easier to get an overall picture of them than by inspecting designs or codes.

In this thesis, we will mostly focus on the former two roles.
Modeling Perspectives

Different languages are more or less expressive in what phenomena they can describe. By selecting certain constructs, and omitting others, a language offers different perspectives of the systems it is used to model [58]. The selection determines to an extent a language’s expressiveness, however the detail level offered to describe phenomena is also a determining factor for this language property. Some of the perspectives offered in existing languages are as follows:

The data perspective is offered by languages which have a high degree of expressiveness for modeling passive phenomena, or data, in a system. A typical example is the ER model developed by Chen [27], and other so called semantic datamodels. A language which primarily has a data perspective of systems is called a data-oriented language.

The process perspective is given by languages which offer constructs for modeling of system dynamics, represented by the processes going on in a system. Process-oriented languages are dominated by the process perspective, although they usually also have very abstract constructs for modeling of static system aspects. A well-known example is the DFD language.

The behavior perspective covers events occurring in systems, and the possible ordering and cause-effect chains of events. This perspective is important for the modeling of control in systems, for initiating and terminating active phenomena as results of event occurrences. Harel’s Statecharts [52] is a good example of a language with behavior perspective.

The object perspective essentially combines the data and process perspectives. Some languages also include the behavior perspective to offer a language with multiple perspectives, all grouped into one construct, i.e. the object. Coad’s and Yourdon’s OOA [30] is a well known language for object-oriented analysis.

The rule or policy perspective is useful for systems where the phenomena are guided by laws and policies of an organization. Some rules may constrain the possible state space of the systems, while others specify the systems’ reactions and processing as response to external events. Hence, such languages may also integrate other perspectives. In knowledge-based systems, rules are used to capture the reasoning steps of domain experts directly. An example of a language with a rule perspective, although still with a research status, is the ERL language in the TEMPORA project [78].

Desirable Properties of CML’s

In order to fulfill their roles as described above, CML’s should have a number of desirable properties. Many of these properties are agreed upon to a large extent. The presentation here is based on those by Berzins et al. [12], Kung [67], and Roman [104].
2.2. Conceptual Modeling

Expressiveness We have already touched this theme above. The expressive power of a language determines which phenomena can be modeled, and at what level of detail. Preferably, the language should have a set of rather orthogonal constructs, each representing a different class of phenomena in systems. The relevant phenomena and the modeling perspectives to be offered, are supported through the chosen constructs. In the next section, we will discuss the expressiveness issue in more detail. For now, we will refine our division of constructs for modeling of static and dynamic aspects somewhat.

- *State components* define the state space of a system. State components are described through properties, and the property values define the state of the components.
- *State laws* place constraints on state components, defining their lawful states.
- *Events* may either be caused by the environment (*external events*) of a system, or through the dynamics of the system itself (*internal events*). The sequences of external events and outputs determine the (external) behavior of a system.
- *Dynamic laws* determine the systems' responses to external events, i.e. the series of state transitions which occur after an external event. Dynamic laws may also be organized into hierarchies with different *sublaw relationships* holding between laws at different levels in the hierarchy.

To deal with the inherent complexity of information systems, many researchers argue that languages should offer hierarchical decomposition of the phenomena being modeled, e.g Harel in [53]. Related to the division of constructs made above, hierarchies have particularly been studied for state components and dynamic laws. This will also be of importance to this thesis. State components may be organized in hierarchical relationships like e.g. *classifications, aggregations, generalizations, and associations*. In principle, the sublaw relationships are of the same kinds as those for state components.

Other properties related to expressiveness, are preciseness/unambiguity, and expressive economy. Expressive economy is achieved if the language is designed so that models can be expressed with as few and simple constructs as possible, avoiding to model the same information more than once.

**Understandability** The expressiveness of a language must balance with users' and developers' ability to understand it. As noted above, constructs must be user-oriented and problem-oriented, and not contain processing dependencies which will constrain the design space unnecessary. User perceptibility is enhanced with conceptual cleanliness, i.e. by using a few orthogonal constructs with little or no overlap in meaning, and by using easily distinguishable presentations (symbols) for different constructs. Using visual languages has been argued as a means for enhancing the comprehensibility of complex models (e.g. Harel [52] and Kung [68]).

**Analyzeability** If a model can be analyzed, it can be checked for various properties automatically, and deductions can be performed to infer consequences of model information.
The extent to which properties like syntactical correctness, consistency, and in part completeness can be determined automatically, depends on the level of language formality, i.e. the extent to which the language can be mapped to some logic and understood by a machine.

**Changeability** During the course of development, a model is likely to be modified many times. Hence, it should be easy to localize portions of models to be changed, and then perform the necessary changes. This is very much a tool problem, however the ease with which it can be done certainly depend on a well-defined language.

**Executability** As suggested by many researchers, CML’s should be executable. We very much agree with this opinion. Both Harel [53] and Brooks [20] hold executability as an important means to overcome problem complexity in systems development. A simple definition of model executability is the following:

*A conceptual model is executable if its state transitions can be computed automatically.*

Hence, for models to be executed, the CML must offer a sufficient detail level, and it must have defined a semantics which determine the state transitions in the models. Referring back to the 'what/how'-issue mentioned earlier, a CML can be executable even when only the external behavior of a system is specified. However, if more detailed functional properties are to be examined, lower level 'how'-aspects must be expressed.

The major motivation for executable models is that they can be exercised, and expose users and developers to their dynamic properties. These would otherwise have to be understood by model inspection and review. We return to the advantages of executability in the next section.

**Model Errors and Language Properties**

Referring back to the different types of errors which can be found in conceptual models, we can now relate these errors to the desirable properties above. On the one hand, language properties to some extent affect the probability of model errors. On the other hand, they are related to the ability to detect errors. Many of the problems in conceptual modeling relate to poorly defined languages. Such languages increase the likelihood of model errors, and at the same time decrease the possibility to detect errors made. Some researchers focus on a lack of formality, e.g. Roman in [104], and Bubenko who criticizes the use of 'fuzzy concepts' in [22]. Others focus on the expressiveness of languages, and point out that many languages currently used do not provide constructs for modeling of all relevant system aspects, e.g. Wand in [129]. Also, although there now exist quite a few executable CML's, they are not widely used in practice. We argue that many existing non-executable
languages could be made executable by relatively minor additions of detail and precision to the languages.

In the following, the strong relationships between model errors and language properties are commented.

**Syntax errors** are less likely to occur with understandable languages. They can only be detected if the language syntax is formally defined, and hence the models can be analyzed automatically.

**Model incompleteness** is more likely to occur with less understandable languages. Partly modeled constructs are detected if the model can be analyzed, while totally missing information can be detected during execution of a model, e.g. missing functionality. Note that if the language can not express all the relevant phenomena of systems, the models will most certainly be incomplete, and the missing information must be kept in the developers' minds until the system is designed and implemented.

**Model inconsistency** is more likely to occur if the language contain constructs with overlapping semantics, so that in a sense, the same information must be stated more than once. It is then important that consistency rules are specified as part of the language definition, and that the language is analyzable so that inconsistencies can be detected.

**Incorrectly stated information** in models is also more likely to be found when using less understandable languages. Model execution is proposed as an efficient means to detect such errors.

**Ambiguous models** are direct results of ambiguous languages, and hence they are impossible to detect without deep language knowledge. However, in some cases ambiguities can be resolved by modeling phenomena in greater detail. In these cases, ambiguities may be detected as model incompleteness.

Note that language properties alone can not assure error-free conceptual models. Additional techniques must be developed in order to detect errors in conceptual models. However, many of these techniques rely on the desired language properties listed above. Some of these techniques will be presented later on in this chapter.

### Language Specification

As indicated above, there are many advantages by having languages defined formally. They may then be analyzed and executed, and precisely defined semantics of language constructs will enhance communication. Also, a user-friendly presentation of language constructs will enhance communication.

In the following, we will describe shortly what aspects of a language need to be defined, and then we will focus on language syntax.
Syntax The syntax or grammar of a language determines what models can legally be expressed in the language, i.e. what models are syntactically correct. It defines the basic constructs, their properties, and relationships to other constructs through formation rules. It also defines constraints which must be satisfied.

Presentation The presentation of language constructs defines the visual appearance of models. Models may be presented textually, graphically, or in combinations of text and graphics. The specification of construct presentation is naturally closely connected to the language syntax. For instance, it is possible to annotate basic constructs with graphical symbols, as done in e.g. MetaEdit [108].

Semantics The semantics of a language determines the meaning of models, and hence how models should be interpreted by humans and machines. A formally defined semantics is provided through valuation functions which translate models into mathematical structures, or to models expressed in other, already formalized languages. In particular, executable languages may be given an execution semantics by translation to another executable language. We will discuss this issue in Chapter 4. Note however, that such translations have two roles: To define the semantics of the language, and to offer model execution as a means to detect model errors.

In addition to these aspects, it is also possible to specify methods formally, i.e. the tasks to be performed during modeling, their dependencies and relationships to conceptual models. Brinkkemper [19] uses the term metamodel to denote a model which express language syntax and its associated method, which he collectively calls a modeling technique. Other researchers have not included method modeling, but rather use metamodel to denote a model which express language syntax and presentation, e.g. Smolander in [108]. In this thesis, we will not focus on the method aspect either. A metamodel is taken to include all the three aspects of language specification listed above.

Metalanguages for Syntax Specification

The expressiveness of the metalanguage used for syntax specification determines which languages can be defined, but the metalanguage also determines the ease with which the specification is performed and understood. It has the same importance for the metamodel, as a CML has for its conceptual models. There are potentially many languages suitable for syntax specification. Some of these are grouped and presented below.

Backus-Naur Form This is a widely used language for specification of programming language syntax, but can obviously also be used for specification of CML’s. It exists in many variants, but common to these is that they can be used to specify a class of grammars called context-free phrase grammars. This is a subtype of phrase-structure grammars (Sowa [112]), which generate sentences being built from hierarchies of phrases. CFG’s are often used due to their simplicity, which makes parsing of sentences relatively easy. A weakness of CFG’s is that they can not express complex constraints. Although context-
2.2. Conceptual Modeling

sensitive grammars somewhat overcome this problem, for this thesis we assume CFG’s to be sufficient.

**Datamodels** These provide a natural way of expressing basic language constructs and their interrelationships and properties. Both Brinkkemper [19], Sorenson [111], and many others use datamodels for metamodelling. A great advantage of datamodels is the ease with which the language syntax is understood, at least when the datamodel is graphical. Also, datamodels often provide straightforward mappings to database schemas, which means that large parts of the storage structures for the tool supporting conceptual modeling can be easily derived. On the other hand, most datamodels offer limited expressiveness for constraint specification. Another disadvantage is that, although possible, datamodels are not that well suited to specify languages with complex phrase structures.

**Predicate Logic** Logic is a powerful language. Compared to datamodels, it allows also complex constraints to be specified. The capabilities of logic for metamodelling is demonstrated by Brinkkemper [19] and Falkenberg [40]. An obvious drawback of predicate logic is that it is less understandable, and also it is not well suited to specify phrase-structure grammars.

**Knowledge Representation Languages** Some of these languages combine the expressive power of datamodels, restricted versions of predicate logic, and the possibility to model on the instance level. In the DAIDA project [61], Telos [86] is used as a metalanguage, offering necessary modeling constructs to be defined through metaclasses. In the AMADEUS project [91], a frame language is used to represent a unified model of different CML’s. Also, Sowa’s conceptual graphs have been used for metamodelling [42]. Although these languages are very expressive, they are also unsuitable for specifying phrase-structure grammars.

**A Metalanguage for CML’s**

An executable CML often use expressions in accordance with phrase-structure grammars. At the same time, their models are often graphs with relationships between instantiated language constructs. Datamodels are hence suitable for specifying these constructs, their properties and relationships. We therefore propose an integrated metalanguage, which combines a datamodel with a language for specification of CFG’s. In addition, we suggest use of typed first order logic when the structural constraints of the datamodel are not sufficient.

More specifically, for this thesis it will be sufficient to use the datamodel depicted in Figure 2.2.

We propose to integrate an extended BNF with the datamodel by allowing the types of
construct properties to be strings generated by a particular non-terminal. In this BNF, alternative productions are indicated by ']', repeated productions (one or more, separated by commas) by \( \{X\}^* \), and possible productions by \([X]\). Terminals are indicated by the use of hyphens or boldface characters. Non-terminals placed within < > produces constants, most often text strings. We use a typed first order logic to express further constraints. Variables used in formulas range over constructs found in the metamodel, and the dot-notation is used in order to access values of construct properties. This integrated metalanguage is sufficiently expressive for specification of the syntax of executable CML's as treated in this thesis. It exploits the advantages of its sublanguages.

A Metamodeling Example

As an example, we want to define an executable CML which is a straightforward extension of the traditional DFD. DFD++ has the same basic constructs as the traditional DFD, but here, flows are allowed to have a numeric value. Rules can be used to specify the values of output flows of processes, as a function of the input flows to the process. These are very similar to entries of decision tables, where conditions guard different alternative output expressions. Flows from stores are assumed to be queries, and these are arithmetic expressions involving the input flows to the store. Flows from external entities are simply regarded as inputs of values from the user. Flows from processes to stores result in the store being updated with the new flow value, i.e. there is implicitly a variable corresponding to the flow in the store. It is assumed that all input flows of a process must have new unread instances in order to trigger the process. A read flag indicates whether the last flow instance has been read or not, for flows from entities and processes. A flow from a store is assumed to always be available.

Portions of the metamodel for this language is depicted in Figure 2.3. Some properties of constructs, e.g. names of processes, have been omitted. Also, some important constraints need to be added to make the metamodel complete. As an example, a requirement must be made to assure that all input flows to a store come from processes. This can be formulated in logic as follows:

\[
\forall f : \text{flow} \forall s : \text{store}(f \in s.\text{inputs} \rightarrow \exists p : \text{process}(f \in p.\text{outputs}))
\]
2.3 On the Expressiveness of CML's

In our opinion, a 'good' modeling language should have the desirable properties listed in the previous section. However, from the large number of existing languages, it seems as if there is disagreement on what a good language is. In this section, we will concentrate on the expressiveness of CML's, and review some analyses on this subject.

In the mid 80's, there was considerable interest in analyzing and comparing different modeling languages and methods, or methodologies, as exemplified by the IFIP conferences [94] and [95]. Several researchers proposed frameworks for methodology comparison, which usually divided the development process into phases, and set requirements to the components or models delivered from each phase. The focus was on what should be represented by the components for a methodology to be 'good'. Also later, several researchers have taken up this or related issues for various purposes. Therefore, rather than starting from scratch with a philosophical treatment of the subject, our discussions will be founded on work previously done by others.

The analyses undertaken have all been aimed at increasing the understanding of conceptual modeling, and of the expressiveness needed by CML's. In addition, the works we will refer to here have been carried out from other specific motivations:

- Wand and Weber have developed an ontological model of information systems (e.g. [125, 126, 127, 129]), which they among other things exploit for evaluation of the
expressiveness of CML's.

- IFIP working group 8.1 has developed a methodology framework from which components may be selected to define new languages/methodologies [93].

- Several projects have been aiming at developing multilanguage environments to let developers choose languages according to the problem or to personal preferences. Common to such systems is the use of a highly expressive internal language which serve as a bridge in the translations between different external languages. Examples which will be presented here are AMADEUS [13, 91], GDR [80], and ARIES [62]. Note that such systems are developed from the recognition that different languages are often very similar in the meaning of the constructs offered.

- Hull and King present a unified datamodel for a survey of semantic datamodels in [60].

The latter three all involve development of what we may call unified languages, and hence they are very interesting for our study. The approach we will take in the following is to represent the essential parts of the works listed above in metamodels, using the metalanguage introduced in the previous section. Although we only focus on the major constructs, the metamodels will serve as a basis for comparison of the different results, and give us useful knowledge about the needed constructs of CML's.

The Ontological Model of Information Systems

The ontological model has its origin in general systems theory. As an information system indeed is a system, it is assumed that systems theory can be used for analysis and design of information systems in particular. The term 'ontological' indicates that the model is concerned only with essential aspects of systems, those which convey their deep structures. An information system is considered to be a model of a real world system, and its goodness is measured by the extent it represents the meaning of the real world system. Deep structures are seen as opposed to surface structures which describe the system appearance for and interaction with its users, and to the physical structures which deal with technological aspects and implementation. The major constructs of the ontological model are depicted in Figure 2.4.

The most central constructs in the ontological model are thing, property, state, and transformation. From these, all other constructs can be derived. Things are what the world is made up of. Things may be composite, consisting of other things. Things are described by properties which map them into values. A kind is a set of things with two or more common properties. The state of a thing at a particular point in time is the vector of values of its properties. A state law restricts the states of a thing to a set of states which are deemed lawful in some sense. A system is a set of things which interact, i.e. their states affect the states of other things in the system. A system can be decomposed into subsystems. The environment consists of things which interact with the things in the system, in the way that they may directly change the state of a thing through an external event. Such an event
may lead the system to an *unstable state*, to which *transformations* respond by bringing the system back to a stable state.

The ontological model applies at all phases of system development. Models from different phases should preserve *invariants* for the final implemented system to be a good representation of the initial real-world system. The ontological model can be used to assess the *ontological completeness* of different CML's, to compare different CML's, and its foundation in systems theory has been exploited to analyze decompositions of systems [127].

### A Methodology Framework

Olle et al. present a comprehensive methodology framework in [93]. This framework is the result of joint work of participants in the IFIP working group 8.1, and builds on the authors’ knowledge of a large number of existing methodologies. As with other frameworks, this one also divides development into phases, of which *business analysis*, *system design* and *construction design* are considered in depth, and focuses on the delivered components from each phase. These should cover the three perspectives of data, process and behavior, as well as the integration of these perspectives.

The detailed descriptions of components give normative guidelines for what models of information systems should represent. In the following, we will focus on business analysis and the components delivered from this phase. In Figure 2.5, a simplified metamodel corresponding to the framework is depicted. The simplifications made should not exclude any essential components.

Static aspects of a business are described by *entities, relationships, attributes* and *constraints*. Entities are described by name, whereas relationships are described by name,
class (unary, binary, n-ary) and type (cardinalities). Attributes define the state space for entities, but can also describe relationships. Attributes may be organized into groups. Constraints are either value constraints or population constraints. Value constraints can be uniqueness constraints, referential constraints or general check constraints. Population constraints involve overlap of populations of different entity types.

In the process perspective, the focus is on business activities. These can be decomposed to a number of subactivities, and precedence relationships exist among activities at the same decomposition level. Activities receive flows of information or material, and produce flows as well. They can be started if certain preconditions hold.

The behavior perspective is covered through business events, which are events being perceived as pertinent to the business. They have an event name, and there may exist certain precedence relationships among them. Also, a precondition may need to be satisfied for an event to take place, and a postcondition may need to be satisfied after an event has occurred.

The perspectives are integrated in the following manner: Activities and events use entities and attributes, in the sense that they may refer and change their states. Conditions refer to entities and attributes. Business events may trigger activities.
The AMADEUS Metamodel

In the ESPRIT-project AMADEUS, an attempt was made to develop a unified metamodel for language integration. The approach taken was to analyze a set of ten well-known and representative modeling languages, identify the needed basic constructs in a unified metamodel, and then provide a general representation of this model. The surveyed languages included JSD, NIAM, SASD, SSADM, IE, SADT, and ISAC. In [91], the work is presented in some detail. Here, we only present the main result, i.e. the unified metamodel. It has the constructs shown in Figure 2.6.

As can be seen from the model, six main constructs are identified from the analysis of the languages; function/process, data flow, entity, event, state and store. Their relationships are also represented in the figure. For instance, a process is triggered by an event, it receives and generates data flows, manipulates entities and produces new events.

A frame based representation language (UMRL) is employed to represent the unified metamodel. The frame language has the standard frame-slot-facet constructs, and the mappings from the unified metamodel to UMRL are as described in the following. First, a construct in the unified metamodel is represented as a frame. No other frames are allowed. Second, a relationship in the unified model is represented as a slot in UMRL. In addition to these slots, slots to define part-subpart relationships, instance relationships and generalization relationships are allowed. Finally, any value-facet of UMRL refers to another frame. Other facets describe properties of slots like cardinalities, range, conditions etc.

Using this internal representation, principles for mapping rules between models are given, via mappings to the unified metamodel. Hence, an integration of CML's in CASE environments is facilitated.
The GDR Metamodel

GDR [80] is a design representation geared towards modeling of real-time systems, and can be used as a basis for defining languages. Examples of translations from Ward's Transformation Schema [130], from Statecharts [52], and from state transition diagrams, to GDR are given. As was the case with AMADEUS, GDR is based on a few simple, but powerful constructs. These design objects are organized in a class hierarchy, as shown in Figure 2.7. Each class has a set of predefined attributes associated. In the following, we briefly describe each main class.

Processes receive information on input ports and produce information to be transmitted through output ports. Processes may either be stores, where inputs may be stored and later produced as outputs on requests, or transforms, which compute outputs from inputs. The process construct is used to represent many phenomena, including program units, objects, states, files etc. Attributes of processes describe decompositions, and link each process to its ports.

Ports identify information transmitting locations of a process. Input ports and output ports correspond to inputs and outputs of a process, while ndports (non-directional) identify locations which are constrained in certain ways to other ndports. An example is when a constant relationship must be maintained between two pieces of information, e.g. for a constraint. All these ports are directly connected to transports, which link them to other ports. The indirect port is used for sending information by address, rather than through a single direct transport. Attributes of ports include associated process and transport, and the type of information which can be transmitted. In addition, output ports may transmit discrete or continuous data, while input ports may queue up incoming data or discard data when the process is inactive.
2.3. On the Expressiveness of CML’s

Figure 2.8: Excerpts of the ARIES metamodel

*Transports* correspond to communication channels. They connect ports and facilitate communication between processes by merging and distributing information on associated ports. Attributes identify the connected ports.

*Tokens* correspond to various types of information. *Control tokens* can be used to activate passive processes, deactivate active processes, or to signal occurrences of events. *Data tokens* carry information used for computations. They are described by attributes which give their structure, basic types, and representation. *Address tokens* contain a port identifier, i.e. the receiver of the address token sent through an indirect port. *Packet tokens* contain an address plus data.

The semantics of GDR is to a large extent given by Petri-nets. Roughly speaking, tokens correspond to Petri-net tokens, ports correspond to Petri-net places, and simple processes correspond to Petri-net transitions. A notable exception is data tokens, which do not have a Petri-net semantics.

**The ARIES Metamodel**

In ARIES, the intention is to provide a set of modeling languages from which developers and users can choose which one to use. The means for integration of models written in different languages is a highly expressive internal representation. Also, different presentations, both graphical and textual, can be defined from the internal representation, so that requirements can be presented in a readable manner. In ARIES, simulations of models are facilitated by translation to a database programming language (Benner [11]).
The most important constructs of the ARIES metamodel are depicted in Figure 2.8. The states of instances of types make up the system state. A type can have multiple subtypes and multiple supertypes. This means that multiple inheritance of attributes is supported. Furthermore, an instance may belong to more than one type at a time. Invariants are used to specify constraints which must hold in all states. Events are used to model dynamic aspects. An event may have a precondition and a postcondition, and it may receive inputs and produce outputs through parameters. An event is effected through a method which manipulates and refers to instances. Methods use traditional control structures (sequence, iteration and choice) to control data manipulation through statements. An event can be activated when its precondition holds, or when it is explicitly called from within a method of another event. Also, events may be organized into a generalization/specialization hierarchy.

A General Semantic Datamodel

Hull and King present a survey of semantic data models in [60]. From this survey one can derive a unified metamodel which covers the central constructs of newer semantic data models. This provides further insight in the requirements to expressiveness for modeling state spaces of a system. However, the metamodel focuses only on this perspective. The metamodel is given in Figure 2.9.

The main construct is that of a abstract data type (ADT). Instances of an ADT belong to the active domain of that ADT. An ADT may have several subtypes (isa relationships), and instances of subtypes may be derived through a membership formula. ADT's are characterized by attributes which may belong to printable types, which means that their values can be output. Attributes may also be aggregates or sets of printable types, and their values may be derived. Relationships between ADT's are represented by attributes as well, single or multivalued. In such cases, it can be stated that an attribute is the inverse
of another. *Constraints* restrict states of ADT's and their subtypes.

Although being focused on the data perspective of conceptual modeling, this metamodel provides useful ideas not covered by the other metamodels. In particular, the use of complex attributes like aggregations and groupings has been advocated in newer semantic data models. A similar survey of semantic data models by Peckham and Maryanski is found in [99].

### A Brief Comparison

Comparing the different metamodels, we find many similarities. On the surface, however, there may seem to be more differences than what is really the case. The differences stem from various sources. One obvious reason is different naming of similar constructs. Another reason is that sometimes, a property of a construct in one model is made explicit as a separate construct in another. Also, particular constructs may be completely lacking in one model, but exist in another. Finally, there is naturally the possibility that errors have been made in the metamodeling, since we in most cases have transformed a textual description into the graphical models.

We make a simple comparison by listing corresponding constructs in the different models, shown in Figure 2.10. Doing this, we also highlight the three former reasons for differences between models. We will use the ontological model as a basis for comparison, since it contains a few, basic constructs, and since it has already been used for the purpose of analyzing CML's. The constructs Environment, System, and Value&Time are omitted, since these are not found in the other models. Note the close correspondence with the four groups of constructs listed in the previous section (state component, state law, event, dynamic law). From the table, we see that Wand and Webers ontology, the methodology framework, and AMADEUS all have separate constructs for the data, process, and behavior perspectives. In ARIES, the event construct covers both processes and events. The unified semantic datamodel only covers the data perspective, while GDR emphasize more on modeling of dynamics than on modeling of data.

For representation of hierarchies in state components, the unified semantic datamodel offers classification, aggregation, generalization, and association. For representation of hierarchies among dynamic laws, all except the unified semantic datamodel has a 'vague’ control structure which resemble spontaneous activation when preconditions hold. As an example, for a business activity in the methodology framework to execute, a condition must hold. In ARIES, methods include control structures from procedural programming languages.

Of the six unified metamodels, only ARIES and GDR are executable. The others do not have the required detail level and a defined operational semantics.
2.4 Model Validation

There are roughly two kinds of techniques which have been developed to assure the correctness of conceptual models and other products of information systems development: Verification techniques and validation techniques. In this section, we first discuss their properties and relationships, and then we present some of the techniques developed to validate conceptual models.

Verification and Validation of Conceptual Models

Model verification aims at detecting errors by analyzing a model, using knowledge about the CML. Verification techniques are exploited to answer the question “Am I building the system right?” (Boehm [16]). As pointed out by Blum [14], verification is objective, in that model errors are detected formally and automatically, without the need to involve users. The types of errors which can be detected are syntax errors, inconsistencies, and in part incompleteness and ambiguities. Verification techniques can also be used to check the consistency between two models from different development phases. Whereas detection of errors can be done automatically, correcting them sometimes requires users to be involved. For instance, conflicting requirements may be resolved by asking users which one is correct.

Model validation techniques are exploited to answer the question “Am I building the right system?”, which can be rephrased as “Does the conceptual model represent the real requirements of the users?”. The types of errors to be detected are in part incompleteness (e.g. missing functionality), incorrect requirements, and also in part ambiguities. Validation techniques check the correspondence between a conceptual model and a real world system, and need the involvement of users. Validation is hence a highly subjective process, and one is never guaranteed that all errors are removed. Naturally, also to correct errors, user involvement is necessary.
The process of verifying and validating conceptual models has been depicted in Figure 2.11. Note that verification normally involves users to a little extent, while validation requires much user participation. The conceptual model is repeatedly refined and corrected, until it is sufficiently stable or the modeling process stops due to economic or other constraints. Some validation techniques depend on verification techniques to be applied. For instance, syntax errors in a model should be corrected before a model can be translated to an executable program.

Even if verification is an objective process, this does not at all imply that it is simple. On the contrary, verifying conceptual models can be a very difficult task, and is a separate research issue on its own. However, in this thesis we address the problem of model validation. In an companion thesis, Mingwei Yang [139] describes a theory for verifying conceptual models, in particular he addresses the problem of model constructivity.

We have previously mentioned some important causes of model errors. Model validation is difficult for the same reasons. We are particularly concerned with the inability to communicate the contents of conceptual models, and to make users understand them. We focus on two aspects of this problem:

System complexity For large and complex systems, the conceptual models will also become large and complex. It is difficult for any one person to have enough detailed knowledge of all parts of the system, and take decisions on behalf of the organization. Rather, many persons may be involved, with different views of the same system, so that conflicting viewpoints and interests will have to be resolved.

Language complexity To be able to represent complex systems, the modeling languages
must be highly expressive. Furthermore, we argue for sufficient formality in the languages, so that they should be executable. This tends to make the languages more difficult to comprehend. So even though CML's are supposed to have user-oriented constructs, they are not necessarily easy for the user to understand. Most CML's are intended for use on a wide range of systems, and consequently employ very general constructs independent of particular applications. Using more application-specific terminology probably increases the users’ ability to understand conceptual models.

In the following, we present and discuss previously proposed validation techniques. We will focus on techniques amenable to automated support. The traditional formal reviews [132, 140] have not been supported by tools so far, and have relied on the users ability to understand models from presentations done by developers. Due to the problems in model comprehension, new automated techniques are needed to support the validation of conceptual models. The techniques we present can be grouped into three categories: execution techniques, complexity reducing techniques, and presentation techniques. We also briefly discuss how they complement each other, and how they can be integrated to form more comprehensive validation techniques.

**Execution Techniques**

**Motivations**

The main idea of execution techniques, is to develop a prototype of an IS on the basis of a conceptual model of the system. As will be discussed below, the prototype may either be derived automatically from the model, or it may have a less formal relationship to it.

Regardless of the approach taken, the motivations for using a functional prototype arise from the fact that the prototype is an executing model of the IS which is developed. If the prototype is exercised in developer-user cooperation, useful feedbacks are supposed to be provided for model refinement and change. The dynamic properties of the models are made more explicit than through traditional reviews, which enhances the users’ understanding of the language and of the particular model. By having a prototype, the users are likely to be more involved in the development process. Their remarks can result in quick changes to the prototype, and another behavior is observed immediately. The increased user participation may have the effect of making users’ attitudes towards the project more positive. Another point is that the prototype can be used as a training vehicle, and assist in a smoother transition when the final system is operational.

Prototypes can also be executed by developers alone to detect unwanted properties like e.g. unreachable states and deadlocks. This is what Harel calls 'specification debugging' in [54]. Finally, although we focus on functional requirements, it should be noted that non-functional aspects like response times can be modeled and estimated using execution techniques. An example is Zave’s PAISLey language [141], which has constructs for reliability and response time modeling.
2.4. Model Validation

Figure 2.12: Alternative ways to develop prototypes from conceptual models

Two Approaches to Prototype Development

As noted above, there are basically two approaches for developing a prototype from a conceptual model. These have been depicted in Figure 2.12. One is what we may call a tightly coupled approach, where the prototype is derived automatically from the conceptual model, or it is the conceptual model itself. The former alternative requires a translation to another executable representation, i.e. a kind of compilation. The latter alternative corresponds to a direct interpretation of the conceptual model. Both these alternatives require that the CML is executable.

The other approach we may call a loosely coupled approach. Here, the prototype is expressed in a separate prototyping language, and the prototype is only based on the conceptual model in an informal way. The CML used is not executable. In some cases, the sole purpose of a prototype is to develop the conceptual model, and hence identify the functional requirements to the system. Early proponents of prototyping tended to favor the loosely coupled approach, like e.g. Gomaa [46]. Also, some propose the integration of conceptual modeling and a loosely coupled approach to prototyping involving only user interfaces, e.g. Morrison [85]. Prototyping languages sacrifice computational efficiency for efficiency in development. Typical prototyping languages for loose coupling are 4GL’s and very high level programming languages like PROLOG and SETL. Normally, the goal of the prototypes developed this way is to identify requirements. When this has been achieved, the prototype is ‘thrown away’. An exception to this is user interface prototypes, from which some parts may be kept for the implementation.

Rothenberg [105] explains the loosely coupled approach by referring to the vagueness of conceptual models. As we have described, this vagueness can be removed by introducing more formal and executable CML’s. In our opinion, there are important problems asso-
associated with the loosely coupled approach. First, it may be problematic to preserve the meaning of a conceptual model when developing the prototype. Secondly, when changes are made to the prototype based on users' responses, it may be difficult to account for this in the conceptual model afterwards. Furthermore, this process requires more work than with an executable CML. Finally, with an executable CML, the knowledge embodied in the model is the same as that of the prototype, and nothing is thrown away afterwards. Rather, the models can be used as a broader basis for more automated design and implementation phases. Hence, by removing the vagueness of conceptual models and using executable CML's, there is less need for the loosely coupled approach.

System Aspects and User Interaction

All aspects and perspectives embodied in a conceptual model can be validated through the use of execution techniques. For instance, the state space of a system can be modeled using constructs like entities, attributes, relationships, and constraints. From this information, it is possible to derive operations for creations, updates, retrievals, and deletions of instances. The constraints can be checked as the operations are performed. Languages like ERT and ERL [78], and CPL [38] are exploited in this manner. Harel's Statecharts [52] has most emphasis on behavioral system aspects, and in the STATEMATE environment [54] input and output events and their temporal ordering can be validated through model execution. The languages in PPP [48] to be presented in a later section are executable, and here the emphasis is on the processing of inputs to produce outputs. As a final example, prototyping of the user interface should be mentioned, since it is the most wide-spread type of prototyping supported by tools. Normally, this kind of prototyping is only loosely coupled to the conceptual model, however it is certainly a vital aspect of a system which should be given attention. It can also be exploited to validate functional aspects of a conceptual model as well.

Interacting with a prototype is much the same as interacting with an ordinary program, except that the user interface may be simpler for a pure functional prototype. In order for the developers to get most feedback, the ideal situation is that the prototypes are exercised in cooperation with users. All kinds of users may be involved, however the basic functionality requires the participation of a domain expert. In order to provide structure to the validation process, scenarios can be set up which reflect typical usage of the system to be developed. If the user interface is prototyped, the execution will provide end-users an impression of how the system once finished, will support their daily tasks.

Problems with Execution Techniques

Although prototyping has many proponents, it is associated with some problems which should be known:

Lack of method Many executable CML's and prototyping languages have been developed, but detailed methods for the use of these validation techniques are rarely
2.4. Model Validation

found. Use of the prototypes should be carefully planned, in order to test all relevant aspects of the conceptual models and to gain confidence in that the model indeed represents the requirements of the users.

Unrealistic expectations Interacting with the prototypes, the users will get expectations to the IS which is being developed. If the final system deviates significantly from the prototype, this may cause some frustration. As an example, if user-interface prototyping is done mostly to capture knowledge about the functions to be supported, the interface to the delivered system may be different from that of the prototype. Since deviations are likely to occur, it is important to state the limitations of the prototypes.

Error free models Most often a conceptual model can not be executed unless it has been checked to be consistent and complete in the sense that all constructs are fully specified. Also, it may not be possible to execute only portions of the model. This would be helpful when a particular function or other aspect of the system is to be tested in isolation.

The prototype as a black box When a prototype is executed, usually only its external behavior can be observed. The user reports external events and initiates operations, the prototypes request inputs and produce outputs, all this in a temporal order compliant with the conceptual model. In normal operation, this is exactly what we expect from a system. However, in a validation process, the conceptual model, and hence the prototype, will contain numerous errors. As the effects of these errors are observed as unwanted external behavior, they have to be detected and corrective actions must be taken. The first problem is then to be able to relate the external behavior to the internal structures of the model, i.e. to the way the model is built. Stated another way, the problem is to open up the black box, and follow paths of computation to account for the external behavior. For large and complex models, this is a difficult and crucial task.

From the introduction to this thesis, it should be clear that the latter problem will be addressed. We therefore briefly describe some supporting techniques which have been developed to overcome this problem.

Supporting Techniques

Model animation In the case that the prototype is derived automatically from a conceptual model, information about the current status of execution can be depicted in the conceptual model. This is most effective when the language used is graphical. Active elements of a model is highlighted in some way, e.g. to show which processes are executing, which control states are active etc. Statemate is a commercial environment which provide animation of Statecharts referred to above [54]. Another example is animation of rule-based models described by Tsalgetidou and Loucopoulos in [122]. Animation reveals the paths taken during execution of a model, and hence it narrows the search for model errors. However, it only displays the paths, it does not explain why they were taken. This is up to developers and users to find out by inspecting the relevant parts of the conceptual model.
Execution tracing and explanations  If information about the internal computations is made available, the reasons for the observed behavior can be found on a very detailed level. A simple form of tracing is to display values of state components or initiations and terminations of dynamic laws with their actual parameters. This kind of tracing is found in e.g. REMORA [77]. A more sophisticated approach is to store the traces on a file or in a database, which can be inspected later on. Depending on the detail level of the stored information, the traces may be very difficult to understand. Furthermore, they may be very large, and hence it is difficult to search for relevant information in them. Moreover, the use of an execution trace requires that a hypothesis is made to account for the observed behavior, and then the relevant portion of the trace is searched in order to establish or reject the hypothesis. This process must be repeated until an explanation is found. Tracing techniques have so far mostly been used in programming environments, as an aid in program debugging. However, as we recognize a prototype to be a special kind of program, we are lead to believe that these techniques can be used for model executions as well. This has already been proved in the support environments for Statecharts [54] and Gist [115]. The weaknesses of tracing techniques can be overcome by providing effective mechanisms for searching in a large trace, or by combining the techniques with model animation or explanation generation. Explanation generators let users ask questions about the observed behavior, and create focused descriptions which include information from the conceptual models and the execution traces which explain the interesting phenomena. Explanation generation is really a presentation technique, and will be presented below.

Complexity Reducing Techniques

These techniques reduce the complexity of conceptual models by adjusting the detail level or the part of the model which is in focus. The result is that the developers can tailor the presentation of conceptual models to the needs and capabilities of the users involved in the validation process. There are basically two categories of techniques, composition techniques and abstraction techniques.

Composition techniques  These remove details from a model by exploiting constructs of the modeling language. Typical examples include aggregation and generalization found in many datamodels, and the composition of processes into a super-process in a DFD. Hiding details in this way, the complexity of a model can be vastly reduced. Obviously, the complexity should not be hidden to every user. Composition techniques do not require much extra tool support, although it would perhaps be beneficial to record the detail levels appropriate for users involved in the validation process. The 'walkthrough' of the conceptual models could then be supported simply by displaying the models at the specified detail levels.

Abstraction techniques  These reduce complexity by creating views of conceptual models to focus on selected portions, or to see the model from a particular perspective. The detail level of the presented model remains the same. As an example, a particular process and its input and output could be selected for presentation. By offering views of the
2.4. Model Validation

conceptual model, the validation process can take into account who the users are, in the same way as with composition techniques. However, it requires some more tool support. Views must be specified and interpreted, the specified information retrieved, and finally presented. By relating views to users before a validation session, the walkthrough of a model can be specified, in the same manner as for composition techniques.

Currently, complexity reducing techniques are addressed by Seltveit in a companion thesis [106, 107]. In her work, additional complexity is added to the techniques in that specified views are allowed to be updated, and must be merged in a later stage. It should be clear that composition techniques and abstraction techniques can be successfully integrated, as they complement each other in the way that model complexity is reduced.

Presentation Techniques

The basic idea of presentation techniques is that, given a model, its content can be presented to users and developers in ways more comprehensible to them. This can be done by changing the language, the presentation medium, the perspectives, or the layout of the original model (Gulla [47]). Changing the language to one which the users are familiar with has obvious advantages. The new language may be natural language, or another CML. If the original language is textual, it may be useful to specify a graphical presentation of it. Also, the dominating perspectives of a model can be changed, so that the model is seen from a new viewpoint. Finally, the perceptibility of models can be enhanced by improving their layout, e.g. by removing crossing lines.

Using presentation techniques, the content of the original model is not changed, only its presentation. Hence, they are particularly useful when the users know what the models should express, but are unable to see that they really do.

We will focus on three categories of techniques, paraphrasing, explanation generation, and intermodel translation. The former two are used to present portions of a conceptual model in natural language, while the latter is used to translate models from one CML to another.

Paraphrasing These techniques produce sentences which are in one-to-one correspondence with instantiated constructs found in the model. Each sentence produced is normally independent of the others. The user may decide which parts of a model to paraphrase, but has otherwise no influence on the text which is generated. Examples of paraphrasing techniques are found in the Gist environment [114, 115], and in its successor ARIES [62]. Paraphrasing is also suggested as a means for validating formal requirements expressed in ERAE [39]. In the Gist environment, not only models, but also execution traces can be paraphrased.

Explanation generation An explanation is a description which enables one to understand a certain phenomenon. In our context, the challenge is to explain phenomena rep-
represented in conceptual models. As opposed to paraphrasers, explanation generators offer users a set of questions which can be asked to increase their understanding. Corresponding to each question, explanation strategies can be formulated, which capture knowledge of what information to include in order to make the model comprehensible. The included knowledge naturally depends on the phenomena to explain, and the final explanation may be presented as a combination of text and graphics. Another difference from paraphrasing is that the explanation can take into account who the users are, and the context in which the questions are asked.

Explanation generation may be combined with execution techniques as indicated above. Explanation generators can accept questions related to the external behavior observed, and give focused explanations which provide the rationales for the behavior. The explanations must then include information from an execution trace. This kind of integration has so far been mostly exploited in expert systems (Swartout [116], Chandrasekaran [24]), both in order to validate the experts’ knowledge, and to convince the users of these systems that the advice given are plausible. Without an execution trace, explanations of model dynamics would be very general, including only the type level information found in the conceptual models.

Another useful integration is to combine explanation generation with complexity reducing techniques, in particular to exploit the view generation routines to select portions of a model to be included in an explanation.

Gulla has worked out a general explanation generation component that will introduce explanation generation in conceptual modeling environments on a broad scale [47, 49]. It is integrated with both execution techniques and complexity reducing techniques as sketched above.

**Intermode translation** Given that two languages have some overlapping semantics, it is possible to translate models between the two languages. To preserve the meaning in the translations, the languages have to be equally expressive. If the users are more familiar with the target language, these translations are likely to increase their understanding of the conceptual models. Note that the techniques are applicable even if not all the content can be translated, as long as the limitations are well-known. As mentioned in the previous section, presentations can be defined declaratively in ARIES [62].

**Concluding Remarks**

We will not go into more detail on the different validation techniques at this stage. The emphasis made on execution techniques stems mostly from our interest in the topic and their relevance to our work, but also from the fact that most work on validation techniques so far has been on execution. We will return to execution techniques in Chapter 4 and Chapter 5. In the latter chapter we discuss in more detail the integration of model execution, tracing, and explanation generation to form a more comprehensive validation technique.
2.5 Modeling Support in CASE Environments

CASE environments offer a set of integrated tools for support of IS engineering. The objectives behind CASE were to increase the productivity of the development process, to reduce maintenance costs, and to increase the quality of the developed systems (Bubenko [22]). In spite of a rapid growth of the number of available environments with increasingly sophisticated functionality, so far there is no firm evidence that these objectives are being met on a broad scale. Empirical investigations tend to show that the quality of systems documentation, i.e. the products of the lifecycle phases, has increased with CASE (e.g. Kusters and Wijers [70]). There is no agreement that the quality of the developed systems has increased significantly.

There may be many reasons for the delayed success of CASE. Some of these are organizationally, and others technologically related. On the organizational level, many leaders are still reluctant to commit themselves to CASE, which results in poor training and preparation for introducing CASE technology in the organizations. Also, learning how to use CASE is a time consuming process. On the technical level, some necessary functionality is still missing in most environments. Of course, this may also be a reason for why many are reluctant to introducing CASE in the first place.

We believe with most others that CASE has a future. The technology will gradually improve, and it will be more widespread. We are concerned with the technological aspects of CASE, and in particular with support for modeling in the early phases of the system lifecycle, i.e. so called upper-CASE. In the following we will discuss the functionality of such CASE environments, and point out some weaknesses. Much of what is said on functionality is applicable to CASE in general.

CASE Functionality

Based on the presentation by Bubenko [22] and Hewett & Durham [57], CASE functionality is divided into the following categories:

Model editing and storage Most CASE environments provide modeling support for a set of languages, of which many are graphical. The editors facilitate retrieval, creation, manipulation, deletion, and storage of models. A striking feature of these tools is their user-friendliness, being based on graphical user interface technology. Editing a model is often syntax-directed, so that many errors are avoided. Models are stored in CASE repositories. These range from simple, file-based systems, to powerful specialized systems built on top of a DBMS. The more advanced repositories provide access to model information via a high level datamodel, e.g. a variant of the ER model, multi-user support, versioning etc. Repositories also provide a means for integrating different tools through data sharing. As noted above, a language metamodel can be used to derive the model storage structure, i.e. parts of the repository schema. Some additional administrative information must usually be included in the schema as well.
Model verification As we have described before, models can be analyzed to detect various types of errors. Syntactical errors, inconsistencies, and in part incompleteness can be detected if the language is sufficiently formalized. Besides the mere identification of errors, the tools should point them out and explain them to the developer, and support an efficient modification to repair them.

Model validation In the previous section, we presented some validation techniques which could be included in CASE environments. Current commercial CASE environments focus on user interface prototyping to support validation. A small number also provide executable CML’s.

Documentation A report generation tool is often found in CASE environments. The reports generated may follow predefined standards, or the tool may have a separate specification language which can be used to produce customized documentation. Textual explanations of models must be added manually.

Translations and transformations Some environments offer support in language translation, for instance to translate between models in subsequent development phases, or to generate code from models. Also, models may be transformed to improve them in some sense, e.g. to improve layout, or to make them otherwise more optimal according to some criteria.

Project management CASE environments are turning from single-user to multi-user systems, to account for the increasingly complex systems which are built and requires the coordinated efforts of many developers. Project management tools provide support in project planning and control. Versioning and configuration management are also needed.

Existing CASE environments offer this functionality to a varying degree. Traditionally, support for model editing, model verification and documentation have been emphasized. There is now a tendency to integrate tools across development phases, and extend the functionality with project management in particular, in order to provide more complete environments, e.g. Chen in [26]. Combining upper-CASE with lower-CASE can be achieved through repositories or agreed-upon interchange formats.

In the next section, we focus on what is missing in CASE environments with respect to support for conceptual modeling. We will not go into a broad discussion on this issue, rather we address some weaknesses directly related to our research.

Relevant Problems with CASE

The first problem we focus on is the use of inadequate languages. We have previously discussed the importance of languages. Recognizing this, when a CASE environment is to be developed, the choice of languages to support is among the most vital decisions made. We have argued for expressive, user-oriented, sufficiently formalized, and executable languages. Unfortunately, such languages are rarely found in commercial CASE. From the list of the most supported languages for system analysis given by Hewett and
Durham [7], we find that the point made by Bubenko [22] that most CASE environments support languages of the mid seventies, is indeed true. The list includes languages like DFD, process dependency diagrams, textual process narratives, state transition diagrams, entity-relationship models, and entity-life histories. Their popularity may be because of their simplicity, and due to the fact that they had some tradition even before CASE was introduced.

Another aspect of the inadequacy problem, is that the chosen languages may not fit the current practice in organizations, which still would like to exploit CASE technology. Often organizations have developed their own particular languages and methods, which are not supported by any CASE environment. This means that if they want to introduce CASE, they have to change. At a level below this, individual developers and even users may be familiar with particular languages, which are not all supported by the chosen tools. These problems are addressed by multi-language environments anticipated in the AMADEUS project [91] and in the research system ARIES [62].

The second problem we address, is the lack of validation support. This problem is recognized by Norman et al. in [90], in particular the need for functional prototypes in addition to user interface prototypes. They also see a potential of more domain-specific modeling languages for enhancing users ability to validate requirements. Although we have seen that many validation techniques exist, most of these are still only found in research prototypes. Apart from user interface prototyping, the execution approach is rarely supported. This may come as a surprise, since after all, leading researchers in the field have emphasized the role of rapid prototyping and executable languages. Also, investigations made by Lubars et al. [79] and Kusters et al. [70] show that developers find that validation support is lacking, in particular support for functional prototyping. Some reasons for the lack of executable CML's may be the slightly increased level of detail and complexity often required, the cost to develop the necessary support, and the fact that model execution is not sufficiently supported by methods.

The final problem to be addressed is the lack of flexibility or customizeability. We have touched this problem already, as we discussed problems with unsuitability of languages with respect to problem, organization, or individual developers. Enforcing a new practice in an organization will reduce the acceptance of CASE, and increase the costs associated with the introduction of a new technology. The lack of flexibility concerns more than only language aspects, for instance methods, user interface and software platforms. Next we present what we may call meta-environments or meta-tools, which offer CASE to be adapted to satisfy particular needs.

**Meta-CASE**

There are very few commercial environments of this kind. They sometimes go under the name CASE-shells (e.g. Smolander [108]), or metasystems (e.g. Sorenson [111]). Common to these systems is the ability to customize a CASE environment to some degree, ranging from simple layout changes to specification of the most vital properties of an environment, including its languages and methods.
In principle, every part of a CASE environment could be customizeable, however the focus in research prototypes and commercial systems have been on the following areas:

**Languages and repository schema** This makes it possible to define the syntax of languages, and the corresponding repository schema with necessary constraints. Given this information, the metatool can create modeling editors with checking of syntactical constraints. The actions to take on syntax violations may also be specified declaratively.

**Language presentation** The more advanced systems can let both the textual and graphical presentation of languages be specified. A simpler feature is to allow presentations to be changed, but not the underlying language.

**Methods** Tools for process modeling have been integrated to a little extent, and is still mostly in the research realm.

**User interface** Many CASE environments offer some customizeability on the user interface.

**Technical platforms and interfaces** Customizeability in this area means that a CASE environment can be used on different platforms of hardware and software. If interfaces to other tools can be easily made, the customer can include these tools in a more complete environment.

Note that the latter two areas do not require any sophisticated metamodeling, and can be offered by an ordinary CASE environment as well.

The development of a customized CASE environment is in many ways similar to ordinary IS development. After all, a CASE product is just another information system. A meta environment can be regarded as a special kind of application generator, where large portions of the system to be developed can be declaratively specified. The development process is depicted in Figure 2.13. *Environment developers* cooperate with the future users of the environment, and develop a metamodel of it using some metalanguage. The metamodels are then used to create a new, customized CASE environment. Meta CASE provide a very productive way of developing CASE environments. It would certainly be possible to use an ordinary CASE environment to develop another, but with the current technology it would certainly be more costly. A major reason for this is that current environments provide little high level support for developing advanced graphical user interfaces of the same kind as they have themselves.

The use of meta-CASE to develop CASE environments gives both benefits and risks to vendors and customers, as summarized below.

**Benefits to the customers:**
1) Support for local practice should give more rapid acceptance of CASE. 2) The environment is changeable and extensible, and opens up for creative use of CASE. 3) It gives a larger degree of vendor independence.
2.5. Modeling Support in CASE Environments

Figure 2.13: A simple architecture for meta environments

Risks to the customers: 1) A higher cost must be expected before the product is operational. 2) There are risks of lower robustness and lower performance. 3) Less support from vendors in evolution, which may be problematic if 'old' models are to be kept compatible with the modified environment.

Benefits to the vendor: 1) One product may serve a broader section of the market, and customized packages can be offered in niche markets. 2) Up to a point, the product can evolve with competition.

Risks to the vendor: 1) The 'metaness' adds to the technical challenges in development. 2) It is difficult to guarantee that the customers will have continuing support for their customized system.

Table 2.1 summarizes the main properties of some research and commercial meta environments with respect to the different areas of customization. The table shows that method modeling is still at a research stage in CASE, and that the commercial systems naturally put more emphasis on the technical platforms, interfaces, and user interface aspects. Also, so far the research systems exploit somewhat more complex metalanguages, which offer more complex objects and constraints to be modeled.

Another observation which we have made, is that although languages can be defined from scratch, there is almost no support for building environments with executable languages. The RLP system by Davis [32] is an exception, since all domain specific languages which can be defined have a basis in an extended finite state machine language. Also, the IPSYS Toolbuilder provides a high level programming language to develop code generators, but this language is not much different from existing programming languages like C, Pascal.
<table>
<thead>
<tr>
<th>System</th>
<th>Metalanguage</th>
<th>Presentation</th>
<th>Method</th>
<th>UI</th>
<th>Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLP [32]</td>
<td>BNF</td>
<td>Textual</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Metaedit [108]</td>
<td>Graphical DM</td>
<td>Graphical</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Socrates [120, 134]</td>
<td>DM</td>
<td>Graphical</td>
<td>Task model</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DAIDA [61]</td>
<td>KR (Telos)</td>
<td>Textual</td>
<td>Telos</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IPSYS [7]</td>
<td>DM</td>
<td>Both</td>
<td>Prototype</td>
<td>Yes</td>
<td>Many</td>
</tr>
</tbody>
</table>

DM = Datamodel
KR = Knowledge representation language
- = Not considered

Table 2.1: The major properties of some existing meta environments

etc. In general, the meta environments provide little support for specifying and implementing translation tools.

### 2.6 Conceptual Modeling in PPP

This section presents the CASE environment PPP\(^2\) which supports conceptual modeling. It has been described in various papers including [48, 76, 138], and in Lindland’s PhD-thesis [75]. We include the description here in order to illustrate many of the points made in this chapter, and because the results of our work will be integrated in the PPP environment.

#### Introduction to PPP

PPP is an experimental CASE environment. It provides modeling support for a set of four integrated languages, covering different aspects of the analysis and design stages of systems development. PrM (Process Model) is an extension of the traditional DFD, and hence can be used to describe functional aspects of a system. PhM (Phenomenon Model) is an extension of the entity-relationship model. PLD (Process Life Description) is a block-structured, visual program design language, used to describe computations of undecomposed processes. UID (User Interface Description) is used to model graphical user interfaces.

In this thesis, the modeling languages are of most interest. The functionality and architecture of the environment are described elsewhere, and will only be described briefly here. First, we present the languages PhM, PrM, and PLD, since they are the basis for prototype generation, and since support for the UID language has not been firmly established and integrated in PPP yet. Then we briefly look at the functionality of the environment, ending

\(^2\)PPP is an acronym for Phenomena, Processes, Programs
up with an emphasis on prototype generation. For a presentation of UID, see [101]. This presentation builds to a large extent on the presentation in [49].

The PPP language is described with respect to the following banking domain:

A customer of the bank can have several accounts and loans registered. All customers are identified by an id, but the bank also keeps the customer's name, address, and classification. The classification is the bank's judgement of the credit worthiness of the customer, and is assigned one of the values good, neutral, or bad. Customers are divided into two categories, institution customers and person customers. Associated with an institution customer is a set of users, i.e. names of persons authorized to use the customers' accounts. The salary of person customers is also kept. The accounts have an identifying number, a balance, and an interest rate, while a loan is characterized by a loan id, interest rate, date granted, initial loan amount, and balance. Associated with loans is a payment plan that says how they are to be paid back in terms of smaller regular payments. A payment is registered with an id, a date of payment, a due date for the payment, and an amount. Transactions on the customers' accounts contain information about the transaction id, the amount to be deposited or withdrawn, the date, and the transaction type (either deposit or withdrawal).

There are four functions in the systems:

- The customer can apply for a loan,
- open an account,
- deposit and withdraw money, and
- monthly statements of accounts are sent to all customers on the first day of each month.

Applying for a loan, the customer must first have opened an account in the bank and cannot have been 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.

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. If she is already registered, the new account is just added to the existing 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 then checked to see if the balance exceeds the requested amount. If so, the customer gets a notice, and 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 the customers account. A deposit is processed directly. In both cases, information of the processed transaction should be given to the customer, and stored for later use.
The **PhM** language

Phenomenon models describe static aspects of a problem domain. Basically, the PhM language is an extension of the well-known entity-relationship model. It has the same basic constructs **entity class**, **relationship class**, and **attributes**. The symbols for these constructs in PhM are shown in Figure 2.14, which depicts static aspects of the bank system. There are three main extensions compared to the basic ER model: It is possible to specify different types of **attribute relations**, **coverage** of relationships, and **subclass relationships**.

An attribute is a relationship between an entity class or relationship class and a datatype. There are four different types of attribute relations. An entity class must have an **identifier** attribute, which uniquely identify an instance of the class. An identifier is indicated by the abbreviation ‘id’, as for account_id for entityclass Account. A **repeating group** relates an entity class to a set of values of the same data type, as with users of Institution_customer.

A **quality** is an attribute of the entity class as a whole. Any other properties are simply called **attributes**.

The coverage of a relationship may be **full**, meaning that every instance of an entity class must participate in the relationship, or **partial**, meaning that participation is permitted, but
not obligatory. In the example, we see that each customer must have an account, but not necessarily have taken up a loan.

A subclass represents a subset of an entity class, of a relationship class, or of another subclass. It can have only one superclass, and it inherits the identifier of this superclass, along with all other attributes. An instance of a subclass is always member of the set of instances of its superclass. The two classes Person_customer and Institution_customer are subclasses of Customer.

An extension of the PhM language is now underway [139]. Major characteristics of this language are complex attributes (tuples and sets arbitrarily combined) and the use of methods to retrieve and update entities and relationships.

The PrM Language

PrM is used to record functional requirements. As will be shown, it has some characteristics in common with Ward’s Transformation Schema[130], and other languages like Kung’s Behavior Network presented in [68]. A top level PrM model for the bank system is shown in Figure 2.15. The most important constructs of the PrM language will be illustrated through this model.

Processes have the same meaning here as in a DFD, i.e. that of a transformation of input flows to output flows. They can be decomposed in a way similar to decomposition in DFD. The decomposition of P1:Process transaction consists of five subprocesses, P1.1 through P1.5, as shown in Figure 2.16.

Flows and stores also have the same meaning as in a DFD. However, the PPP language allows flow contents to be specified by datatypes or by linking flows to parts of PhM models. External agents have the same semantics as external entities in DFD. The name difference has been made to emphasize the dynamic aspects of entities. In the model, Customer is the only external agent.

Control flow is 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 legal combination of terminating flows. Non-triggering and non-terminating flows can be sent/received any time while the process is active. In the example, process P3 can be activated if it receives the triggering input flow Customer\(^3\), and will terminate by sending the terminating flow Account.

To define logical relationships among input flows and among output flows, PrM offers input ports and output ports, respectively. There are three basic kinds of ports corresponding to the three logical connectives: Conjunction (AND), disjunction (OR), and exclusive disjunction (XOR). From the example model in Figure 2.16, we see that the input port of

\(^3\)Marked with a 'T'.

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 output port of the process is an XOR port.

Moreover, a port may be conditional, repeating, or both in any combination. A conditional 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 New\_withdrawal indicates conditionality\(^4\).

Now, we can describe a single execution of, for instance 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.

\(^4\)Repetition is indicated by an unbroken line.
Timers can act as delays or clocks. In the former case, they have an input flow which determines the length of the delay. After the delay has passed, an output flow is sent. A clock sends output flows regularly, as does the timer T1 in the bank model. This clock sends an output flow on the first day of each month. Both clocks and delays can be turned off.

Resources contain a number of items which may be needed for processes to execute. A flow from a resource to a process specifies how many items are needed, and a flow from a process to a resource release items. No resources are found in the bank model.

For more detailed descriptions of PrM, see [75, 96].

The PLD Language

This language can be characterized as a block-structured, program design language [121]. It is used to specify process logic, i.e it can define the behavior of a single sequential process. Constructs for receiving assignments, iteration, and choices are defined. In addition, two constructs for receiving and sending data provide interprocess communication and communication with users and databases. Figure 2.17 shows the process logic of process P1.2 represented in a PLD model.

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_transaction. The receive construct holds this information, together with names of local variables and types of the data received. Here, we see that the information includes an account identifier,
account balance, customer name, amount, and date.

A choice construct follows, to take action if the requested amount is larger than the current balance. This 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 requested amount is too large, a withdrawal rejection is sent to the customer, with an error message and the available amount. The send construct identifies the data flow and the receiver involved, and contains one expression per data element to be sent on the flow. Next, a new amount is requested, and then an update of the variable amount is performed in an assignment construct. This holds the variable to be updated, and an update-expression which evaluates to the new value of the variable.

Then a new choice construct follows, to distinguish between an aborted and an accepted transaction. The withdrawal is rejected if the requested amount is not acceptable or is zero, and the flow Aborted_transaction is sent to the customer. Otherwise, the accepted withdrawal (Ok_withdrawal) is sent to process P1.4.

Loop constructs are not present in the model. They contain an expression to be evaluated
each time the loop body can start execution, and correspond to WHILE-loops and FOR-loops of high level programming languages. In addition, it is possible to declare local variables within a PLD diagram.

**PPP functionality at a glance**

The functionality of PPP can be divided into 1) modeling editors (a UID editor is not developed yet), 2) verification support to check model consistency, 3) report generation, 4) various translations between the different languages to facilitate more rapid developments and consistent models, and 5) validation support by prototype generation.

Central to the development of PPP models is model integration through the use of translations and transformations. Due to the tightly integrated languages, initial models in one language can be generated from information contained in other models written in other languages. Users of the environment then avoid the tedious work of modeling the same information twice, but this also increases the likelihood of arriving at consistent models more quickly. The following are examples of these translations and transformations:

- Initial PLD models can be generated from the ports and flows found in PrM models. The structure of the generated PLD model then directly reflects the port structures, and the types of send and receive constructs are fetched from the flow types. In Figure 2.17, the non-shaded parts of the model are generated automatically, while the shaded parts indicate developer-supplied information after the initial model was generated.

- Ports of a process can be abstracted from ports of the process’ decomposition. Extensions to this work is presented in [139].

- Initial SQL table definitions can be generated from PhM models.

- Reports can be automatically generated from models.

As will be described in the next section, the generation of prototypes is a translation as well.

**Generation of Prototypes in PPP**

Prototype generation has been a central activity within the PPP project. This has been motivated from the need for model validation. Several target languages have been used, including Ada, C, Simula, and TEQUEL/C:

- PLD's can be translated to Ada tasks which communicate through a rendezvous mechanism [48]. It is assumed that data types and expressions in the PLD's are
compliant with a subset of the Ada syntax, so these can be directly put into the generated code. Since there may be many PLD’s in a realistic system, performance can only be expected at a 'prototype level'. Weaknesses of the generator are that database manipulation is not included, and that the user interface is very simple.

- If the PLD’s are populated with expressions from a simulation language, SIMULA, simulation models can be generated [50]. Using simulations, non-functional requirements like response times and reliability can be validated. A weakness of the generator is that manipulation of databases is not included.

- In [6] the generation of C code from PLD’s is presented. The concurrency in PrM models is overcome by 'gluing' PLD’s together at corresponding send and receive constructs. The result is large, complex program with no modular structure. However, from the user’s point of view, the generated program is acceptable as a prototype. Also here, no database manipulation is included.

- Finally, in cooperation with members of the ESPRIT project TEMPORA, generation of programs in a temporal, rule-based language, TEQUEL, has been supported [66]. The rules can also include calls to routines written in C. The generated program can handle database operations, though in a somewhat restricted form. This can be overcome by extending the expressions of the PLD language. The semantics of PrM and PLD is preserved to a large degree, but the generation assumes that processes are atomic, and can not communicate with other processes except at the start and end of their executions.

The different prototype generators have proved their usefulness in providing facilities for model validation. However, their shortcomings may be summarized as follows:

- They all generate code outside the PPP environment. Hence they rely on tools over which we have no control. This may be a problem if special execution techniques are desired, and it may also be more problematic to intervene the execution to start up other validation tools. Furthermore, the feedback loop is then longer, as modeling and execution are not tightly integrated within the environment.

- They do not integrate prototypes of the user interface. These can be generated from UID models [100].

- They have no or limited ability to manipulate instances of datamodels. Adding expressions which manipulate a relational database to the PLD language would make it possible to generate embedded SQL. It would be more appropriate to manipulate data on a datamodel level for prototyping purposes. The extended PhM language [139] has a defined algebra and a semantic data language to manipulate instances. A mapping to SQL will have to be defined if a relational database is used as the target in prototype generations.

- Finally, they have been developed in a rather ad-hoc manner, although they are built using many of the same principles.
Figure 2.18: The major constructs of the PPP metamodel

The PPP Metamodel

Figure 2.18 show metamodels for the PhM, PrM, and PLD languages. We have used the simple datamodel introduced earlier, and the metamodels are built from the repository definition found in Andersen’s thesis [5]. Note that we have omitted some layout information, and no construct properties are given. The syntax of expressions in some PLD constructs varies with the target language for prototype generation. We assume that they are similar to expressions found in high level programming languages. Figure 2.19 shows an example syntax for these expressions. Also, to simplify the metamodels, the isa relationship is used. This is not found in the repository schema based on PROLOG facts.
Expression ::= Expression Addop Term|Term
Term ::= Term Mulop Factor|Factor
Factor ::= <Variable>|<Constant>|'(Expression)'
Addop ::= '+'|'-'
Mulop ::= '*'|'\'
Condition ::= Condition Conn Condition|'NOT('Condition')'|'(Condition')|Comparison
Comparison ::= Expression Rel Expression
Conn ::= 'AND'|'OR'
Rel ::= '='|'|'>|'|'\geq'|'|'\leq'

Figure 2.19: Syntax for conditions and expressions in PLD

2.7 Conclusions

In this chapter, we have presented some important topics related to conceptual modeling in IS engineering, and discussed the problems which are relevant for our study. We agree with proponents of user-oriented and executable modeling languages, recognizing the many advantages these properties give. In particular, these properties enhance our ability to involve users to validate conceptual models.

Unfortunately, existing CASE environments are most often rather inflexible in the languages and methods they offer support for, and also validation techniques are rather poorly supported. In order to combine a flexibility in language with executability, we envisage the need for three basic components: 1) A metamodeling component which can be used to define the syntax of languages, their presentations and storage structures for models, 2) a translation component which offer powerful translation mechanisms to translate models to an executable representation, and 3) an execution component which provides a target language for the translations and a means for executing the resulting models. We have seen that there exist CASE environments with variants of the former component included. In our study, we focus on the latter two components. Analyses reviewed in this chapter indicate that many languages are very similar. Hence it should be feasible to find an expressive internal language which cover a wide range of CML’s.

We also recognize the problems with using model execution as a validation technique on its own. Explanation generation is proposed as a technique to overcome this problem, but it needs effective access to an execution trace. We therefore consider this problem in connection with model execution.

Some of the problems associated with the execution of conceptual models in the PPP environment have also given motivations for our study. In particular, the plans of extending the PPP languages with a more powerful datamodel, and of using different languages to express process logic call for a more expressive target prototyping language than those used today. We therefore intend to demonstrate the feasibility of the general mechanisms we provide by showing how they can be integrated with the PPP environment.
Chapter 3

Translation Facilities in CASE Environments

In this chapter we primarily discuss the use of translations in CASE environments. First, we go through a number of tasks which fall into the category of syntax-directed tasks. Second, we state some requirements to a general translation facility which will support developers in the specification and implementation of translators. Then we present the main principles followed in the specification and implementation of translators. We conclude by giving some examples of environments which use translations with different degrees of sophistication.

3.1 Syntax-Directed Tasks in CASE Environments

There are a number of tasks in CASE environments which can be characterized as syntax-directed, i.e. that the syntactical structures of the input guides the goal structure of the tools performing these tasks. By automating these tasks in CASE, the tedious and error-prone manual work that would otherwise be needed is eliminated. Although we are particularly concerned with translations, these are of the same nature as other syntax-directed tasks, and hence can be built using the same kinds of support. We use the term translation facility to denote a system which allows translations to be specified and at least partially be implemented automatically. The need for this kind of support has long been recognized for programming environments, as seen e.g. through the development of parser generators or compiler-compiler. However, such facilities are rarely used in general CASE environments which support conceptual modeling. Rather, ad hoc procedural approaches tend to be used, where translators are dedicated to a single task and where the source and target languages are fixed.

We now briefly describe some of the syntax-directed tasks which are undertaken in CASE, with an emphasis on translations.
Multilanguage Environments Examples of work in this direction are the AMADEUS project [13], Lubars’ general design representation GDR [80], and the ARIES environment [62], which were all introduced in the previous chapter. Another example is given by Delugach in [35]. He uses Sowa’s conceptual graphs as an internal language to translate between the ER model, dataflow diagrams, state transition diagrams, and requirements networks in SREM [4]. Of these works, only ARIES uses general translation facilities. Also, many CASE environments provide a set of integrated languages which cover different aspects of a system. It is often the case that these are semantically overlapping, so translations can be used to avoid having developers state the same information more than once. This is the case with e.g. PPP, as explained in the previous chapter, and with PRISMA as explained in [89].

Phase Integration Support Some environments aim at providing automated support for transitions between different phases of the systems lifecycle. This can be done by translating models resulting from one phase to initial models for the next phase. Usually this can only be done semi-automatically, since the transition involves design decisions, and there is a wide range of models in the latter phase which are consistent with models from the previous phase. Hence the developers must often interact with the environment, to select an alternative if many alternatives exist. Both the DAIDA prototype [28, 61] and the IPSEN prototype [74] give support along these lines. PPP gives support in the transition from PrM modeling to PLD modeling, as explained in the previous chapter. However, in PPP, only one alternative is given as a suggestion from the system, and the developer can change this at wish later on. Also, many commercial CASE environments provide some support for phase integration.

Code Generation Code generation can be seen as a special instance of phase integration, in that it involves translation from some intermediate model to a prototype or the final implementation. Although not widely used in practice, prototype generation is found in many research environments. Examples of environments supporting code generation include TEMPORA [78], STATEMATE [54], PROTO [64], and PPP. Also, commercial environments like IEF use translations, for instance to port implementations to different target platforms. The majority of code generators are developed procedurally, closely tied to the source and target languages. This leads to less adaptable generators, which is not a problem as long as the environment relies on a stable set of languages. However, in more unstable situations more flexible approaches are needed.

Documentation Documentation is tedious, but yet important work which can be partly automated by extracting and structuring information in information systems models to form textual/graphical documents. Usually this involves little translation, but still it is a syntax-directed task as output is produced during a kind of parsing of the models. Some environments offer the contents and structures to be explicitly specified e.g. STATEMATE, whereas other rely on a fixed specification, e.g. PPP. Many commercial environments support documentation.
3.2. Requirements to Translation Facilities

Transformation/Restructuring  When the syntax-directed task involves changes to a single model, we say that the model is transformed or restructured. The need to transform models arise for several reasons. First, models may be transformed to improve the efficiency. In Gist [8] and ARIES [62], an initial operational specification gradually evolves into the final implementation by a continuous replacement of real-world modeling constructs with more efficient constructs from the programming world. Second, models or programs may have improved readability through use of transformations. An often used example is the removal of 'gotos' from program code, replacing them with explicit loop constructs. As a final example, models or programs need to changed if the underlying language grammar changes. A semi-automatic method based on the grammar changes is presented by Garlan et al. in [43]. For a set of simple grammar changes, transformation rules are derived automatically. For more complex changes, developers have to write specialized transformation procedures.

Formalization  An often used method for formalizing a language, is to define its constructs through the translation to another, already formal language. Falkenberg [40] and Lubars [80] use this principle in order to define the semantics of languages for modeling of dynamic system aspects. Falkenberg uses an extended Petri-net formalism to define semantics of DFD's and action diagrams, while Lubars uses Petri-nets to define parts of the semantics of his general design representation.

Reverse Engineering and Reengineering  Reverse engineering means finding patterns in program code or database schemas, and produce higher level abstractions from these in order to get an understanding of the underlying design and conceptual model. These abstractions are then used in the maintenance of the program, which probably was not developed using CASE technology. In a sense, reverse engineering can be seen as phase integration in the 'opposite' direction. It should be clear that when analyzing the existing code, the parsing structures of the input play a key role. Reengineering means porting the produced abstractions into a CASE environment and thereafter go towards a new implementation, at least partly automatically.

3.2 Requirements to Translation Facilities

Requirements to translation facilities resemble many of the requirements to software in general. Here, we will review some important requirements from the viewpoint of the translation task, and point out some particular requirements for translation.

Separate Specification Level  It has proven advantageous to have a separate specification level for translations, or at least to be able to formulate translations declaratively (Rich and Waters [102]). For this purpose, separate translation specification languages are developed. Purely procedural approaches tend to lead to less maintainable translation specifications. If the specifications are closely linked to the grammar of the source
and target languages, they are more easily adapted to changes in the grammars. Note that as language grammars are specified abstractly in BNF or semantic datamodel form, the translation specifications should preferably refer to these rather than to the details of their repository representations.

**General** The translations should be easily expressed, and the specification language should be complete for the kinds of translations needed. Furthermore, they should be adaptable, and hence easily maintainable.

**Independence from Languages and Tasks** The translation specification language should be independent of source and target languages, and of the particular translation task. This way it can be used for the variety of translation tasks used in CASE environments.

**Mixed Model Representations** Translations may involve mixed representations of models, in particular datamodel and parse tree representations. We saw that in the PPP environment, model information can be stored as relations. However, expressions in PLD constructs must be parsed before models are translated. One can also imagine that such expressions are stored directly as parse trees. A particular case of mixed model representations, is when a translation is needed from a textual to a graphical representation, or vice versa. Furthermore, it should be possible to port the translation implementation to different repositories. In fact, this can be considered to be a translation task in its own.

**Predefined Semantic Actions** The translation specification language should include a set of commonly used predefined *semantic actions* which are used in the production of outputs in the target language when certain patterns in the source models are found. Examples of such actions are actions to combine partial results to one overall result, to store and retrieve model information, and to parse strings according to a given grammar. It should also be possible to extend the basic set of actions as necessary.

**User Interaction** It is sometimes necessary to involve the developer when conflicting translations are applicable, and let her choose the most appropriate. This may for instance be the case in the transition from one phase in the systems lifecycle to the next.

**Automatic Implementation** Preferably, translation specifications should themselves be automatically translated to an efficient implementation. It should be possible to specify this translation using the translation specification language itself.

The requirements above are the most important identified from our study of translation tasks in CASE. They will guide the evaluation of existing approaches, and the development of our translation facility presented in Chapter 8.
3.3 General Translation Principles

For a translation between two languages, the languages need to be compatible to some extent. This means that the meaning of the source models can be preserved to some degree in the target model. The degree of compatibility must of course be well-known. If the target language can not express the information in the source model, some information will be 'lost' in the translation. On the other hand, if the target models need more information than what is found in the source models, it may be necessary to generate default information, or to involve the developer to get the necessary information. Another possible problem is that two-way translators may not end up with the initial model when this model is translated forth and back again. This may happen because different constructs in the source language are mapped into the same target language construct. When the reverse translation is to be done, it may be impossible to decide which original construct was used. These problems must of course be fully understood, and the use of the target models generated must take this into account.

Leaving these general issues, we now go into the main principles followed in translations. A general architecture of a translation facility is depicted in Figure 3.1. First, translations are specified referring to the source and target languages. From this information, the translator is derived. It accepts models written in the source language, and controls the use of the implemented translations to produce a model in the target language. We refer to the two models as source and target models, respectively.

Traditionally, automated syntax-directed translation has been much studied in the context of compiler systems (Aho and Ullman [3]). Without going in details, we will present some major ideas taken from these systems. They certainly have relevance to more general translation problems. During translation in these systems, basically three steps are performed: Parsing of the input to produce a parse tree, construction of a dependency graph which determines the order in which the translation results are produced, and finally the actual production of output.
Chapter 3. Translation Facilities in CASE Environments

**Parsing** The parser embodies knowledge about the grammar which presumably produced the input string. It constructs a parse tree which leaves are tokens supplied by a lexical analyzer. These correspond to terminals of the grammar. Interior nodes of the tree correspond to non-terminals of the grammar, and the children of these nodes correspond again to the non-terminals and terminals in the productions. It is not our intention to study the details and subtleties of different classes of grammars, and the parsing problems associated with these. Rather, in this thesis, it is sufficient to exploit well-known parsing systems, like e.g. YACC, which is described in the next section.

Translation specifications are generally given through *syntax-directed definitions*. In short, these are a context free grammar specifications extended with a set of attributes with semantic actions used to compute their values, and possibly additional semantic actions which can have side-effects as e.g. to print values or to update global variables. The semantic actions are linked to productions in the grammar. There are two subsets of attributes, *synthesized* and *inherited* attributes. Synthesized attributes have their values computed from values of attributes belonging to children nodes in the parse tree. Inherited attributes have their values computed from parent and sibling attributes in the parse tree, and hence these attributes provide a context for the translation. Note that terminals of a grammar can only have synthesized attributes, which values are taken directly from the parse tree.

The general form of a semantic action (or *semantic rule*) which computes the value of an attribute connected to the grammar production \( A \rightarrow \alpha \) is \( b = f(c_1, \cdots, c_n) \). If \( b \) is a synthesized attribute of \( A \), then all the \( c_i \)'s belong to the grammar symbols in \( \alpha \). Otherwise, if \( b \) is an inherited attribute of one of the grammar symbols in \( \alpha \), then the \( c_i \)'s belong to the grammar symbols in \( \alpha \), or to \( A \).

**Dependency graph construction and evaluation of semantic actions** Dependencies between different attributes can easily be found by inspecting the semantic actions which compute their values. In the formulation above, \( b \) depends on each of the \( c_i \)'s. The dependency graph is a directed acyclic graph where an edge between a node in the graph to another means that the latter depends on the former. A topological sort of this graph yields a valid evaluation order of the attributes. For translation tasks, attribute values may represent the results of translations. As noted above, semantic actions with side-effects may also be used to produce translation results.

For restricted classes of syntax-directed definitions, it is not necessary to construct a dependency graph, and the semantic actions are performed during parsing or by a fixed traversal of the parse tree. *S-attributed definitions* use only synthesized attributes, and the values of these attributes can be computed through a simple bottom-up traversal of the parse tree. *L-attributed definitions* include inherited attributes, but these may only depend on attributes of nodes to the left in the parse tree. Their values can be computed in a depth-first traversal of the parse tree.

In the following two sections, we will describe two translation facilities which are partly based on the general principles sketched here.
3.4 The Parser Generator YACC

YACC (Yet another compiler-compiler) is a widely available and much used parser generator [3], used to develop the front end of compilers. It accepts a file with syntax-directed definitions, and generates a parser for the grammar which executes the associated semantic actions when the different productions are recognized during parsing.

A YACC source program consists of three separate parts:

The declarations part: This contains C declarations to be used in the two latter parts. Also, different grammar tokens which are returned on a predefined format of an associated lexical analyzer are declared.

The translation rules part: Each rule consists of a grammar production and the associated semantic action. The semantic action is a sequence of C statements, i.e. arbitrary code can be used to update attribute values or other global variables, and to write results to a file. In particular, one synthesized attribute is associated with the non-terminal of the production. Its value is accessed through the variable $$, while the value associated with symbol i on the right hand side of the production is accessed via the variable $$. As an example, consider the grammar production

\[
Expr \rightarrow Expr \ '+' \ Term \quad \{$$ = $1 + $3\}
\]

Here, the semantic action computes the value of the attribute of the leftmost Expr, as the sum of the attribute values associated with Expr and Term on the right. Values of these synthesized attributes are passed implicitly up the parse tree. Additional attributes must be supported through global variables declared in the declarations part and additional statements in the semantic actions.

The supporting C-routines part Here, a lexical analyzer yylex() must be provided, as well as procedures used in the semantic actions, procedures to handle exceptions etc.

YACC provides a nice coupling between the parser generated and the semantic actions which can be used to produce the results of a translation. To some extent, there is a separate specification level of the translators developed this way. However, all semantic actions are C codes, and hence the parts which produce the results need to be developed from scratch. This may be costly, as noted by Howells et al. in [59]. No predefined semantic actions are provided, but naturally a library of such actions could be developed and used in later translator development. The use of syntax-directed definitions makes maintenance easier, but still it may be difficult to maintain the procedural codes of the semantic actions. Another problem with YACC is that it is only supposed to work on parse trees corresponding with context free grammars. Its features would have to be adapted to the more general repository representations in order to be generally applicable in CASE environments. Of course, one can imagine data from repositories being translated back and forth between
datamodel and parse tree representations, but this seems like an awkward solution. In fact, it also requires that both representations can be handled in an integrated way. It would be better to adapt the translator facility to work on more general representations.

3.5 Syntax-Directed Experts in POPART

Wile recognizes in [135] the similar nature of different syntax-directed tasks like translation and transformation, where the abstract syntactic structures of the input guide the tools performing the tasks. According to Wile, the performance of these tasks rely on a body of programming knowledge which needs to be captured, represented, and reasoned about. In his POPART system, this knowledge is represented in so called syntax-directed experts. These have a declarative nature and are organized in a flat structure. During the performance of a task, these experts cooperate in a goal-directed manner. Each expert is capable of performing a single task on some part of an abstract syntax tree, and for an overall task to be performed, subgoaling mechanisms decompose problems into smaller problems which can be solved by other experts.

The abstract representation of an expert is as shown in Figure 3.2.

\[
F(\text{tree}, I, O, G) = \\
P_1(\text{tree}) \Rightarrow \text{if exist } r_j \text{ so that } g_{1j}(d_{1i}(\text{tree}), r_j) \text{ then } C_1(r_1, \cdots, r_{k_1}) \\
\cdots \\
P_n(\text{tree}) \Rightarrow \text{if exist } r_j \text{ so that } g_{nj}(d_{ni}(\text{tree}), r_j) \text{ then } C_n(r_n, \cdots, r_{k_n})
\]

Figure 3.2: The general structure of syntax-directed experts in POPART

The input to an expert is an (part of an) abstract syntax tree. The input characterization \( I \) and output characterization \( O \) encode characteristics of the input and output trees, respectively. The goal achieved, \( G \), simply gives the name of the goal achieved when the expert has performed its task. Each expert may have a number of rules associated, where the left hand side of the rule denotes a particular input pattern, while the right hand side gives the decomposition of the goal into subgoals and combine partial results to an overall result. The \( P_i \)’s are predicates used to find matches with the input tree, and to bind variables which values are passed on to other experts. For each match found, the corresponding right hand side is a candidate for issuing subgoals to be achieved by other experts, and for combining partial results. Here, the \( g_{ij} \)’s are subgoals, while the \( d_{ij}(\text{tree}) \)’s are decompositions of the input tree. \( r_j \) is the result returned by the expert which performs the task needed to achieve \( g_{ij} \). A number of subgoals are issued, and if all of these are achieved, the partial results are combined to the overall result \( C_i(r_1, \cdots, r_{k_i}) \). Note that two rules may have the same left hand sides, but different right hand sides. Also, two left hand sides may both match the input tree, even if they are not completely identical. These properties lead to a need for a backtracking mechanism, in case a subgoal fails to be achieved by another expert.
3.5. Syntax-Directed Experts in POPART

Figure 3.3: Control of syntax-directed experts in POPART

Figure 3.3 shows the control of the cooperating experts. First, a formal representation of the experts are translated to a table representation, where each \( \text{lhs} \rightarrow \text{rhs} \) pattern gives a separate rule in the table. The initial problem consists of an input tree with its characterizations, and a goal to be achieved. First, an expert capable of achieving the goal is selected. Then, within this expert, a generated rule is found which left hand side matches the current input. Then the problem is decomposed according to the subgoals found in the right hand side of the rule. Each subproblem is described in the same manner as the initial problem, and then a new cycle of expert and rule selection is initiated. If all chosen rules achieve their goals, the process terminates when the initial chosen rule receives the results from its immediate cooperating experts, and the overall result is produced. However, if at some stage a chosen rule is not capable of performing its task, the system backtracks, and another untried rule matching the input is applied.

To give a simple example of an expert, consider the definition in Figure 3.4 of an expert which simplifies boolean expressions. Note that the rules are simplified. \( \%E \) is used as a shorthand for the simplification of \( E \), i.e. an implicit subgoal is issued. Also note that in the specification of an expert, the input tree is not given as an explicit parameter, but is still passed on in the background to be matched with the left hand sides of the rules.

The POPART system has been used successfully in the ARIES environment [62], both for interlanguage translation and for evolutionary transformations as mentioned earlier. The representation of ARIES requirements must conform with the requested format of inputs to POPART, i.e. syntax trees. This is achieved by using a repository which can have relations implemented as syntax trees. Hence the tree format is directly available for POPART in the repository.

The organization of programming knowledge into experts using a separate specification language for these experts is to a large extent in accordance with the requirements listed above. This makes translator specification easier, and makes the specifications more easily maintained. The approach is independent of source and target languages and of the specific task, as has been demonstrated through its extensive use.
def boolean expert (simple boolean, i = boolean, o = boolean, goal = simplify)
% excerpts of rules:
(1) not Expr: Expr = not Expr' => % Expr'
(2) not Expr => not % Expr
(3) Expr1 and Expr2: Expr1 = not Expr1' and Expr2 = not Expr2' => not(% Expr1' or % Expr2')

Figure 3.4: A syntax-directed expert in POPART

One problem with POPART, is that it does not support the use of attributes explicitly. One could imagine synthesized attribute values being returned as part of a result, but inherited attributes which provide a context for the translations can not be passed on in the opposite direction. A way to solve this problem would be to follow the same approach as suggested for YACC, i.e. to use global variables. Also, as is the case with YACC, the syntax-directed experts are developed to work on syntax-tree representations only, and not with the mixture of representations found in CASE repositories. Finally, little is said about predefined functions and predicates available to support the definition of an expert, except for the possibility to combine partial translation results to an output syntax tree.

3.6 Translation Facilities for Phase Integration

In this section we present the DAIDA and IPSEN environments, which give the possibility to explicitly specify dependencies and mapping rules between two subsequent phases of the systems lifecycle.

Transformations in IPSEN

In [74] Lefering presents how requirements and design (programming in the large) documents can be integrated. The generic framework depicted in Figure 3.5 has been developed as an abstraction of previous integration tools that had been developed from scratch in an ad-hoc manner.

The source and target documents belong to two subsequent phases. The integration document shows the links between components of the source and target documents, i.e. it shows instances of applied transformations\(^1\) which created target from the source. In addition, it keeps user decisions taken when alternative transformations could have been used. Finally, it may keep dependencies between different links to indicate that one link is depending on another.

\(^1\)The author uses the term 'transformation' rather than translation.
The transformations are encoded in rules which have a Pattern → Action structure. Pattern refers to the source document, to the integration document to check existing links, and possibly to the target document to check results from previous transformations. Action consists of insertions and deletions of information in the target and integration documents. The rules are divided into sections corresponding to the sections of the source document, in order to ease the search for applicable rules during the transformation process. At the time of writing the paper [74], the details of the rule formalism had not been established, and Pattern and Action consisted of calls to procedures written as interfaces to the documents involved.

The integrator controls the transformation process and the consistency checks. For transformations, the source document is scanned section by section according to a default pass. If information is found with incomplete/inconsistent links, it searches for applicable rules. If more than one rule is found, they are presented to the developer for selection. The chosen rules is then applied, and portions of the target document is stored, together with new links and decisions in the integration document. If a rule refers to links which are established by other rules, and these links do not exist, the corresponding rules are tried recursively to establish the links. This eliminates the need for a careful ordering of the rules before processing.

During modeling, inconsistent links may appear, e.g. if source or target information is deleted, or if rules applied earlier for some reason are no longer applicable. The integrator provides a consistency check which searches for inconsistent links, and either initiates a dialog with the developer, or tries to repair the inconsistent links automatically. In each pass, only the inconsistent links are involved in possible re-transformations, other links are left out. Hence the tool is called an incremental integration tool. In the paper, transformations from ER and DFD models to function modules, procedures, data object modules, data type modules, and type definitions are illustrated. However, the IPSEN framework is general, and can be used for other applications as well.

Although not completed, the ideas of IPSEN seem quite promising in that they fulfill many of the requirements established in the beginning of this chapter. At the time of writing, the procedures of the document interfaces are written manually, but the plan was to generate the implementation automatically from a separate rule language as sketched above. This
will give a separate specification level, and enhance maintainability of the translations. From the presentation, it is unclear what representations the rules will refer to, i.e. what kinds of repositories are assumed. If they are extended to deal with structures like syntax trees, they will have an even wider applicability.

The Mapping Framework of DAIDA

In DAIDA [28, 61], system development is broadly divided into requirements modeling, design, and implementation. It employs different languages for the different phases. For requirements modeling, the knowledge representation language Telos [86]. During design, the TaxisDL language is used, whereas the implementation language is a database programming language, DBPL. DAIDA supports transition between the phases through a mapping framework, which offers mapping rules, or dependency types to be specified. The developer plays an active role in choosing between different types and instantiating them. In [28], the focus is on mappings between requirements specification and design. A special feature of DAIDA, is the use of non-functional requirements to guide the mapping process. This will however not be described in the following presentation.

There are three categories of dependencies:

Classification dependencies define mappings of central constructs in the source and target models. An example is the mapping of activities into transactions or scripts in the design language.

Attribute dependencies determine how the attributes of the constructs above are mapped. For instance, non-null, single valued attributes are mapped into changing attributes in TaxisDL. Likewise, outputs of activities are mapped into produces of transactions in TaxisDL.

Isa dependencies and instantiation dependencies determine how the isa relationships and classification relationships of constructs are mapped.

Both the requirements and design languages are represented in Telos, and so are the dependency types. A dependency type has necessary attributes giving the classification dependency of the mapping. For each attribute of the construct to be mapped it has references to attribute dependencies, and it has possibly references to isa dependencies and instantiation dependencies. In addition, constraints on the mappings can be specified, e.g. to control the order of dependency type instantiations.

The developer interacts with DAIDA in the following manner during the mapping process:

1. Components which have not yet been mapped are displayed.
2. The developer selects a component to map.
3. The candidate dependency types are displayed.
4. The developer selects an available type, or defines a new one.

5. A template for the depending design entity is displayed.

6. The developer completes the design entity.

Note that the components to be mapped can not be chosen completely arbitrarily, since constraints of dependency types must be satisfied during the process.

The mapping framework of DAIDA seems to be quite powerful. In the same way as in IPSEN, the translations can be separately specified with the advantages for maintenance etc. A weakness as seen from our viewpoint, is the close connections with the knowledge base of DAIDA (Conceptbase), and that it is not designed to work with syntactical structures like syntax trees. As constructs of new source and target languages can be defined in Telos, as well as new dependency types, the framework is very general.

3.7 Conclusions

From the discussions after each section on translation facilities, we see that many, but not all of the requirements identified in the beginning of this chapter are satisfied. The most problematic requirement is the necessity to deal with mixed representations of models. Facilities from the 'programming world' usually assume translations to be formulated with respect to syntax or parse trees. Facilities from the conceptual modeling side assume database/knowledge base representations. Since executable models tend to need both kinds of representation, an integrated approach is necessary.

We have seen that many principles are shared for different translation tasks, and that a separate specification level seems appropriate for adaptability/maintainability of the translations. Also, some argue that rule-based approaches to translation specification captures design knowledge more explicitly than more procedural approaches.

If multiple translations are applicable at the same time, there may be a need to involve the user, depending on the task to be performed. We have seen that when design decisions are made, the alternatives are made available to the developer, who then chooses the most appropriate. For the task of translating a conceptual model to a prototype, the primary goal is to capture and preserve the semantics of the conceptual model. Hence translations should in these cases be made fully automatic. They should also be efficient in order to provide quick responses when changes have been made to the models.
Chapter 4

Executable CML’s in CASE Environments

In this chapter we elaborate on executable CML’s and their use in IS engineering. We first relate executable CML’s to other prototyping languages. Then we show the impact executable models have on the development process in terms of different lifecycle models exploiting prototyping. We describe different alternatives to develop a prototype from a conceptual model, and we also describe some execution mechanisms supported by the model executor to control and understand the executing prototype. Requirements to CASE environments supporting executable CML’s are stated, and then we present six selected languages and their supporting environments in some detail. In the conclusion, we summarize the lessons learned, in particular for the expressiveness of the executable CML’s studied.

4.1 Prototyping Languages

A prototyping language is simply a language in which one can model system prototypes. In Chapter 2 we discussed the use of executable CML’s for prototyping, but there certainly exist related alternatives. In this section we briefly discuss executable specification languages, operational specification languages, and very high level languages (VHLL’s), and their relationships to prototyping and executable CML’s.

Executable Specification Languages The term 'specification’ is usually taken to denote the 'what’ of a system at a particular level of abstraction. If requirements are expressed in an executable specification language having user-oriented constructs, these languages correspond exactly to what we have called executable CML’s. As such, conceptual models in e.g. TEMPORA [78] and BNM [68] are executable specifications.

However, the term executable specifications has also been used to denote specifications at lower abstraction levels, e.g. for specifications at the level of data types or subprograms.
Examples include algebraic specifications (e.g. van Horebeek [123] and Goguen [45]), and specifications expressed in logic (e.g. DeVille [37]).

**Operational Specification Languages** These were developed from the recognition that the what/how issues are intertwined, and that the division made should rather be on problem-oriented versus implementation-oriented concerns, as mentioned in Chapter 2. Two examples are the PAISley [141, 143] and Gist [8, 9] languages. In our classification, these are executable CML’s. However, as operational specifications are closely connected to a particular development approach\(^1\), we prefer to use the term ‘executable CML’.

**Very High Level Languages** These are most often general purpose programming languages. The very high level indicates that complex data types and operations are built in the languages. This results in more compact, more declarative, and less efficient programs than those written in a traditional HLL. There are two ways to exploit these languages for prototyping. One is to use them in a loosely coupled approach as described in Chapter 2. Another alternative is to use these languages as target languages when prototypes are generated from an executable CML. As the constructs offered by VHLL’s are usually more implementation-oriented than those found in CML’s, they are not that well suited for conceptual modeling.

Example languages include 4GL’s and database programming languages, PROLOG, and functional languages like MeToo [56]. 4GL’s are tailored to development of database applications, and give opportunities to specify tables, reports, screens etc. PROLOG provide lists as the primary data type, but arbitrary complex terms may be used. Its clause resolutions provide a powerful inference mechanism. Tavendale [118] and Weigand [131] both use PROLOG for prototyping from conceptual models.

### 4.2 Impacts on the System Lifecycle

Does the availability of executable CML’s and prototypes change the way we develop systems? To some extent it does, but the impact on lifecycle models vary a lot. While some exploit executable models to introduce minor improvements to existing lifecycle models, others propose radical changes based on prototyping. These issues have been treated by Agresti [2] and Boehm [17]. We review some major points here.

**Prototyping in the Waterfall Model** A simple way to improve the analysis phase of the traditional waterfall model would be to introduce prototyping to reduce the uncertainty of the captured requirements, and hence remove more requirements errors. This has been depicted in Figure 4.1, both for a loosely coupled and a tightly coupled prototyping approach. Note that for the loosely coupled approach, the final prototype is discarded after

---

\(^1\)To be presented in the next section
the requirements have stabilized. This is what corresponds to throw-away prototyping (Gomaa [46] and Carey [23]). With a tightly coupled approach, one is ensured that all knowledge captured in the prototype is available for the next phases through the conceptual model.

**Operational Specification Paradigm** The *operational specification paradigm* is based on the use of operational specification languages (Agresti [2]). Here, the emphasis is on first obtaining an operational specification which represents the users' requirements, and then gradually transform this specification into an efficient implementation. The transformations are automated as much as possible, and replace problem-oriented, non-efficient language constructs with more efficient constructs in the same or another language. Early proponents of the operational specification paradigm were the developers of Gist (Balzer) and PAISLey (Zave). The paradigm has been depicted in Figure 4.2. Note the similarity with the tightly coupled approach described above. The major difference is the emphasis on automatic transformations from specifications to the final system implementation in the operational specification paradigm.

**Prototyping in Boehm’s Spiral Model** Each cycle of the spiral model [17] brings system development closer to its completion. After having determined objectives for the part of the system to be elaborated, alternative means to accomplish them and their associated constraints are evaluated. In this part of the cycle, Boehm proposes prototypes to be used actively to resolve risks associated with each alternative. In the early cycles for requirements capture, executable conceptual models could obviously be used for this purpose.
Evolutionary Prototyping  Evolutionary prototyping contrasts throw-away prototyping in basically two ways. First, prototypes are never discarded, rather they are evolved into the final system. Second, whereas with throw-away prototyping the focus is on uncertain requirements first, evolutionary prototyping typically starts with the well-known parts of the system. Then this initial part is gradually extended to cover new functionality. It is exercised by users who provide feedback, and the prototype is revised until it is the final system. Evolutionary prototyping gives a managerial challenge, particularly in managing the many versions and configurations being developed as the prototype grows. Evolutionary prototyping does not fit very well with executable CML's as it is now, since the languages used typically are 4GL's for database applications, and little or no conceptual modeling is done beforehand.

4.3 Execution by Interpretation and Translation

If the conceptual model is directly interpreted, the model is the prototype, and no translations are necessary. If the model is translated to another executable language, the differences in abstraction level and styles of expression will affect the ease with which the translations are specified and implemented. Here, we discuss three different categories of target languages: Traditional high level programming languages (HLL's), very high level programming languages (VHLL's), and executable CML’s.

Execution by Direct Interpretation  Developing an interpreter is similar to specifying a language’s operational semantics. To each construct of the language, a semantic function is specified, which gives the computation to be performed when that construct is recognized in the model. Consider the simple example \( a := x \) \( \text{Op} \) \( y \), where \( a \) is a simple variable, and \( x \) and \( y \) may be complex expressions. When such a statement is recognized at run-time, it is
4.3. Execution by Interpretation and Translation

evaluated by first evaluating the values of x and y, and then the operation Op is performed on the resulting values. Finally, the value returned by evaluating the operation on the arguments x and y is stored for the variable a.

Although this example may indicate that developing interpreters is a simple task, it is not, for all but the simplest languages. We have previously explained that CML’s may be rather complex, both in the data structures and control structures offered. It may be a challenge to develop an interpreter with an acceptable performance. As an example, the interpreter developed for the Gist language was too slow to be used for practical prototyping, so translations to a more efficient language were introduced to get sufficient performance (Feather [41]).

However, when an interpreter has been developed, and the CML is relatively stable, certain advantages are offered. Naturally, translations are not necessary after each model revision, and hence the feedback loop is shorter. Changes to models can be made in the editors, and tested on the fly. Also, by controlling the execution through interpretation, it may be easier to link the model execution to tools which provide supporting validation techniques like e.g. explanation generation.

Examples of interpreted languages include Transformation Schemas in Teamwork [15, 130], Statecharts in STATEMATE [52, 54], SXL [73], PAISLEY [141], and structured analysis languages presented by Lea and Chung in [71].

Execution by Translation to HLL’s Although some programming languages have constructs for concurrent computations, for the most both data structures and control structures are relatively simple. They generally have a procedural rather than a declarative style of expression, and are characterized by high performance. As HLL’s usually are the ultimate implementation languages for systems, it would be attractive to generate as much as possible of the systems directly from their conceptual models. However, as the difference in abstraction levels and in styles of expression (procedural vs. more declarative) is quite large, translating a conceptual model into a program expressed in a HLL is not easy. As an example, consider the problem of simulating a database with instance manipulation and checking of constraints in a HLL.

Still, for certain aspects of a conceptual model, or for relatively simple CML’s, translations to HLL’s are feasible. As was described in Chapter 2, Ada prototypes are generated in the PPP environment [48], however without including database manipulation. Another example is REMORA [77] where prototypes are generated in PASCAL with embedded SQL.

Execution by Translation to VHLL’s As we described above, VHLL’s may be suitable target languages for prototype generation from conceptual models. Besides having built in more complex data types, these languages tend to have a more declarative style than HLL’s, e.g. PROLOG and 4GL’s. This is likely to make translation specification easier, but the cost is likely to be decreased performance. As these languages are primarily programming
languages, their constructs are not that of CML’s.

Examples of environments using VHLL’s include ARIES [62](a database programming language called AP5), and REMORA [77](SQL) as explained above.

**Execution by Translation to Executable CML’s** If another executable CML is used as a target language, it is assumed that a sufficiently efficient executor exist or can be developed for this language. Translations are likely to be easier to specify if the target language is expressive, and provides the dominating perspective(s) of the source language. As with VHLL’s, the price is usually slower computations. The target language may again be interpreted, or translated to yet another language for efficiency reasons.

If the executor of the target language has sufficient performance, the potential ease of translation specification and implementation gives a great advantage for multilanguage environments, and for meta CASE environments. Also compared to the task of developing interpreters for each new language, translations to an executable CML is easier.

We see examples of translations to executable CML’s in multilanguage environments like ARIES [62], where external languages are translated into the internal representation language, and with Lubars’ general design representation (GDR) [80].

### 4.4 Execution Mechanisms

An executable system model is in many respects similar to a program. It can be interpreted or compiled, it receives certain inputs and produces certain outputs during execution. The similarities induce a need for mechanisms comparable to those found in programming environments (Harel et al. [54]), particularly as those found in program debuggers. By supporting features like *step-by-step interactive execution, (programmed) batch execution, breakpoints, spypoints* etc., the developers and users achieve better control of the execution.

Step-by-step interactive execution lets the users or developers respond on behalf of the system environment, giving inputs as external events to the executing model. The model responds to the events according to its specified dynamics in a single step, which updates the model state. Further steps may be initiated until the model reaches some equilibrium. After each step the user can inspect the model state, and possibly report new events. Execution in this fashion is found in e.g. Teamwork [15] and STATEMATE[54]. The alternative to this interactive mode of operation, is to store or program all events on a separate file, and then run the models as batch jobs. This naturally limits users involvement to observation of outputs, but it may be useful if complex, varying scenarios of system use is to be set up. Programmed execution allows inputs to be represented as statistical distributions, as done when executing models written in simulation languages. In STATEMATE [54], *exhaustive execution* is proposed as a means to test critical components by generating all possible sequences of inputs, and check the model for unwanted properties like deadlocks.
or unreachable states.

Further control can be gained by inserting breakpoints or spypoints in the executable model, similar to their use in a program debugger. A breakpoint is a statement, which, when reached, stops the execution to let the system state be inspected, state components be updated etc. Spypoints are used to record events for later inspection, or to report to users or other tools about occurring events.

4.5 Requirements to CASE Environments Supporting Executable CML’s

Here, we are only concerned with the support for executable CML’s. We divide requirements into three categories. 1) Requirements to the CML’s used, 2) requirements to the model executor, and 3) requirements for facilities which help in understanding the model execution.

Language Requirements Here, we refer to the requirements listed in Chapter 2. The CML’s expressiveness should be such that any particularly uncertain aspects of the modeled systems can be investigated through model execution. Naturally, the languages should be executable, but we also require that they should not lead to intolerably slow prototypes. A significantly lower performance than for the final system is expected, though.

Executor Requirements The basic requirement here is that executors should take conceptual models as input, and either interpret them directly, or make use of a translator to map them into an executable representation which is then executed. During execution, inputs required and results produced should be presented in ways comprehensible to the users, although a realistic user interface is not expected from a prototype focusing purely on functional aspects of the modeled system. Another requirement concerns the integration with the modeling editors. To ensure a short feedback loop, it should be easy to make changes to the models and subsequently start executions again to observe the effects of model changes. Further, it should be possible to control the execution in various ways, for instance by providing debugging aids like step-by-step executions, breakpoints etc. Finally, during modeling, temporary errors like incompleteness and inconsistencies should not necessarily make model execution impossible. Rather, default informations could be added, or the users could be asked for necessary information during execution, or only parts of an overall model could be selected for execution.

Requirements for Understanding Executions There are various supporting techniques which can be used to enhance the understanding of the observed external behavior of an executing model. We touched upon some of these in Chapter 2. Ideally, the environment should offer a battery of techniques like model animation, tracing and explanation generation in order to cope with different execution situations.
4.6 Some Selected Languages and Environments

Selection Criteria and Presentation Scheme

In this section we present a selection of executable CML’s. We do this in order to make the general discussions on CML’s so far more concrete, and in particular to study the expressiveness of executable languages. We find many executable CML’s in the literature, of which many have already been referenced. The selection we have made is based on various criteria. First, we wanted to present languages with emphasis on different modeling perspectives, and languages intended for modeling of different kinds of systems. This illustrates the general applicability of executable CML’s. Also, the majority of the languages are developed by leading researchers in our field. Finally, we wanted to highlight some of the most common formal foundations underlying executable CML’s.

The selected languages are the following:

- The CML’s of the STATEMATE environment [54], i.e. Statecharts and Activity-charts, used for modeling reactive systems. This is the only language in the selection supported by a commercial CASE environment.

- Gist [9], which to a large degree forms the basis for the internal representation language used in the ARIES environment.

- The CML’s of the TEMPORA environment [78], i.e. ERT, ERL, and PID, emphasizing on representing business policies as rules.

- The Behavior Network Model (BNM) developed by Kung and Sølvberg [69], which builds on Petri net theory, and has a very tight integration of the data and process perspectives.

- TaxisDL, which is the design language in DAIDA [61, 83].

- An object-oriented language called eG-Nets developed by Deng et al. [36], which is also based on Petri nets.

In the presentations, we focus on relevant aspects connected to the three groups of requirements identified previously in this chapter. Hence we focus on issues related to language expressiveness, to the model executors, and to means for understanding model executions.

For language expressiveness, we emphasize on the constructs offered for modeling the static and dynamic aspects of systems, their dominating perspective(s), scopes, and formal foundations if they are made explicit. We illustrate the languages using the bank example introduced in Chapter 2, although we extend it somewhat if necessary, to illustrate particular constructs.

For model executors, we emphasize on the approach taken (interpretation or translations), and on any additional execution techniques offered. If the environments provide supporting validation techniques, we also mention these.
4.6. Some Selected Languages and Environments

Figure 4.3: The bank model in Statecharts

Conceptual Modeling in STATEMATE

STATEMATE [54] is an environment intended for modeling and development of reactive systems, i.e. systems which are characterized by complex behavior, and which models include complex sequences of events, actions, and information flows.

STATEMATE offers two languages for conceptual modeling: Statecharts [52] which is used for modeling behavioral aspects, and Activitycharts which are similar to DFD. The two languages are integrated in that each activitychart on each level of decomposition is controlled by a control activity which is described by a statechart.

Due to the nature of the systems being modeled, the models are dominated by a behavior perspective. However, data items can be modeled to represent the system state and values of information flows. As the language is not intended for data-intensive systems, data
items have simple structures, and there are no constructs for specification of constraints.

Statecharts are used to model the systems’ reactions to internal and external events, in terms of its execution states (subsequently called states) and its control of activities. Statecharts are based on finite state machines, but include among other things hierarchies of states to avoid the state-explosion problem encountered with traditional FSM’s. A higher level state may be an aggregate of simpler states, or it may be an exclusive disjunction of simpler states. In the example model shown in Figure 4.3, the state Banking system is an exclusive disjunction of the five states Idle, Transaction processing, Loan processing, Opening account, and Issuing monthly statement. This means that in any execution state of the bank system, the system may be in one of these five states. Also, the state Transaction processing is an aggregate of the states Processing and Help mode. Hence, when the system is processing transactions, it must be in both these substates concurrently.

State transitions are generally described by formulas of the form $\alpha[C]/\beta$, where $\alpha$ is an event, $C$ is a condition which must be true for the transition to take place, and $\beta$ is an action to be performed when the transition takes place. Events may be e.g. external events, states entered or leaved, activities started, or that a condition on a data item becomes true. The condition $C$ may require e.g a particular state to be active, or a particular activity to be executing in order to be true. The action is either related to control of activities (start, stop, suspend, resume), updates of data items, or scheduling of future events. In the example model, we see that when a new transaction is received, the activity Verify_transaction is started, and the state Verifying transaction is entered. Note that the same event may initiate several state transitions simultaneously, and that several transitions may take place in sequence before a new stable execution state is reached. Also note that we have not shown any activities in the model.

Execution in STATEMATE is offered both by direct interpretation, and by translation to a HLL (Ada and C). The motivation for the latter is to test prototypes close to their final target environments. STATEMATE supports most of the execution techniques described in the previous section, including step-by-step and programmed executions. An executing model may be animated by highlighting active states and activities. Also, an execution trace may be stored for later inspection. Note that unless activities are supported by subprograms, the execution of models only involve the behavioral aspects, excluding state changes made by these activities.

**Conceptual Modeling in ARIES**

In the previous chapter, we presented the translation facilities offered in ARIES[62]. This is a multilanguage environment which has a powerful internal representation language to integrate models expressed in the CML's offered. Here, we will briefly present this language, and illustrate it by showing excerpts of a bank system model expressed in Gist. Gist utilizes much of the expressiveness of ARIES’ internal language.

ARIES uses an extended ER model for modeling of state spaces of systems. Types, instances, and relationships may be modeled. Instances may belong to more than one type,
4.6. Some Selected Languages and Environments

Type specifications:

```plaintext
type customer( Classification | STRING, 
                  Address | STRING);
type account( Balance | REAL, 
               Owner | customer);
```

Invariant specification

```plaintext
always prohibited NEGATIVE_BALANCE 
exist account || account:Balance < 0
```

Action specification

```plaintext
action NEW_TRANSACTION ( new_transaction )
definition insert transaction = new_transaction
```

Agent specification

```plaintext
agent customer( ) where 
demon CUSTOMER_ACTIVITY( )
  trigger true
  response
    choose
      NEW_TRANSACTION((a new_transaction));
    ....
  end_choose;
```

Demon specification

```plaintext
demon VERIFY_TRANSACTION ( transaction )
  trigger exist transaction
  response
    begin
      if exist account || (transaction:Account = account:Id and 
                         transaction:Customer = account:Owner) 
        then
          if transaction:Type = 'deposit' then insert deposit = transaction 
          else insert withdrawal = transaction 
          else REPORT_INVALID(transaction)
        end;
    ....
```

Figure 4.4: The bank model in ARIES’ Gist language

and they inherit properties along generalization relationships to possibly multiple supertypes. Both types and relationships may be derived, and relationships may be of arbitrary arities. Finally, relationships are fully associative. State laws are expressed by means of invariants. In the bank model depicted in Figure 4.4, two type definitions and an invariant have been indicated. Note that properties of type instances may refer to other instances directly. The invariant NEGATIVE_BALANCE states that the balance of an account may never be negative.

System dynamics is expressed through events. In ARIES, an event changes the system state, and has a duration in time. In order for an event to apply, a precondition must be true, and after its completion, a postcondition must be true. Events may be associated with methods, which consist of procedural steps which create, update, and destroy instances. Events may activate other events directly, or they may be activated spontaneously as their preconditions become true. In the example, both agents, actions, and demons correspond to events. The agent customer offer different actions to be initiated, e.g. to process a new transaction. The action NEW_TRANSACTION then inserts a new transaction as an instance of the type transaction. This instance is then input to the demon
VERIFY_TRANSACTION, which is activated as it finds that there exists an unprocessed transaction. It then checks the existing accounts to see if there is an account with the given identifier and owner. If so, it either inserts the transaction as a verified deposit or withdrawal. If not, an action is called to give the invalid transaction back to the customer. Note that this demon corresponds to process P1.1 Verify_transaction in the PPP model from Chapter 2.

The execution approach taken in ARIES is to translate ARIES models to a database programming language called AP5. In the early Gist environment, symbolic execution traces could be parphrased, but this capability has not been transferred to ARIES. However, execution traces can be analyzed by posing validation questions, which are patterns to be searched for in the execution traces as they are generated. This can be used to search for desired (matched patterns) and undesired behavior (unmatched or partially matched patterns). Also, ARIES can execute models with errors like incompleteness and inconsistencies. The execution component in ARIES is described in [11].

### Conceptual Modeling in TEMPORA

The ESPRIT project TEMPORA [78, 119] aims at creating an environment for development of complex business applications. The underlying paradigm is that development of an IS should be viewed as the task of developing the policy knowledge base of an organization, which is used throughout the development process. Business policies are captured by rules which both constrain the information of an organization, and determine the responses to events by choice of operations to perform. Hence, the rule perspective is dominating in the conceptual models.

Three sublanguages are offered: ERT (Entity-Relationship with Time), ERL (External Rule Language), and PID (Process Interaction Diagram). ERT is an extension of Chen's ER model. It extends this language by offering attributes, which are relationships between entity classes and value classes, by offering generalization relationships of various kinds (total or partial, disjoint or overlapping subclasses), complex entities and value classes, derived entity classes and relationships, and finally time stamping of entities and relationships. Time stamping of an object indicates that the historical states of its instances are to be kept for later reference. Figure 4.5 shows parts of the bank system modeled in the TEMPORA languages. Note that Customer is time stamped, and that it has two disjoint subclasses Institution_customer and Person_customer, which give a partition of the class. It also has a derived subclass Bad_customer. Note how attributes are represented as relationships between an entity class and a value class, illustrated by the attribute has of Account.

ERTL is used to express derivation rules for derived objects in ERT models. The derivation rule for the entity class Bad_customer is given in the example model, and includes reference to past states. ERL is also used to model different types of constraints. These include value constraints (restrict values of attributes), population constraints (restrict number of entities with given properties), and uniqueness constraints (states that certain attribute values are unique within a class). Also, a general ERT constraint is offered if none of the
4.6. Some Selected Languages and Environments

Figure 4.5: The bank model in TEMPORA

predefined constraint types fit. This is expressed similar to relational calculus. In the bank model, a population constraint states that there can not exist more than 100 accounts with a negative balance.

PID is used to specify business processes. It is similar to DFD, but has process interfaces specified by ports (inspired by PrM in PPP), and dataflows are linked to views of the ERT model. Undecomposed processes have their process logic specified by action rules. These have the form WHEN Event IF Condition THEN Action. Here, event is a named external or internal event, or a time point reference which triggers the process. Condition is some condition on instances of the database, and Action is either a call to a procedure or direct manipulation of database instances. The bank model includes an action rule for the process Verify_transaction. It assumes that there exist a procedure with the same name which is to be executed when a new transaction is given from the user.

Different alternatives for model executions have been pursued. One alternative is to map the ERT models to a database schema of a relational DBMS, and have a rule manager to process the rules on top of this database. In order to be executable, all rules must be classified, and action rules must have their procedures implemented. In a given state, there may be several rules which can be executed, and bring the system from one state to the next. Also, all actions of rules are executed atomically. The rule manager can execute a model step-by-step, and it is also possible to view the historical states of the database, including the external events input from users. Finally, work on animation of TEMPORA languages has been presented in [122].

Another alternative has been to translate a restricted version of the ERT and ERL to an object-oriented language built on top of PROLOG [98].

1) Population constraint: "The number of accounts with negative balance should not exceed 100"
   number_of ( Account has Balance < 0 ) <= 100

2) Derivation rule involving time: "A bad customer is one who has had an account with negative balance"
   Bad_customer(C) is_derived_as sometimes_in_past Customer(C) has_acc Account has Balance < 0

3) Action rule initiating process: "The process Verify_transaction is started when a new transaction has arrived"
   WHEN Transaction THEN Verify_transaction
Conceptual Modeling Using BNM

The Behavior Network Model [69, 68] is intended for conceptual information modeling. It provides a tight integration of the data and process perspectives in a single language. Broadly speaking, it is based on the ER model for modeling static system aspects, and a process model with preconditions and postconditions specified for the lowest level processes for modeling dynamic aspects.

The extended ER model includes constructs for modeling subclasses and complex attributes of type list, set, stack etc. It can also express some structural constraints. In addition, a relational calculus language is used to specify both static and temporal constraints. The bank model expressed in BNM is shown in Figure 4.6. The right hand side shows parts of the static model for the bank system. Note that attributes are not modeled.

The dynamic aspects of a system are modeled through the use of processes which trans-
4.6. Some Selected Languages and Environments

form input flows to output flows, and may be decomposed as in DFD. In addition, process interfaces may be specified in a way very similar to ports in PPP, and control flows are distinguished from data flows. Processes are linked to entities and relationships which they reference or manipulate in some way. Undecomposed processes are described by a precondition and a postcondition expressed in relational calculus, which may include references to past states. They are hence similar to transitions in Petri nets. In the bottom level model, a place is associated with each flow, which is used to depict the individuals involved in transformations, and give them variable names which are used in the specification of preconditions and postconditions.

In the bank model, the process Verify_transaction will be decomposed. Its interface specification states that both New_transaction and Customer must be input, and that either Withdrawal, a new deposit, or an invalid transaction is output. The process which checks a withdrawal has been decomposed to four processes, which are specified through preconditions and postconditions. The specification of process Amount_too_large has been given. It states that the process is executed if the requested amount is greater than the current balance. If so, an error message is returned to the user, and the withdrawal is sent to be aborted or changed through the input of an acceptable amount.

The underlying formal foundation of the 'bottom level' BNM is Petri nets. BNM is similar to Predicate/Transition Nets [44] in the way transitions are specified, and similar to colored Petri nets in the way tokens have type specifications.

The execution approach taken is to translate a BNM model to an application on top of a relational DBMS. A prototype CASE environment is being developed which supports this translation, and also supports model analysis similar to what is found for Petri nets, e.g. by constructing reachability trees. Besides this, no support for understanding executions is reported.

Conceptual Modeling Using TaxisDL

TaxisDL is used as a conceptual design language in DAIDA[61]. Since in DAIDA, new languages can be defined using Telos, TaxisDL should be a highly expressive language which serve many of the same purposes as does the language we will develop in this thesis. So, even if TaxisDL is only partially executable, we found it appropriate to consider it here.

TaxisDL combines a semantic data model for modeling static system aspects with transactions and scripts for modeling of system dynamics. The semantic datamodel builds on entities, relationships, and attributes. Entity classes can be specified with generalization relationships, and instances of such classes have unchanging identities. Attribute values may belong to simple data classes, enumerated classes, or they may refer to other instances, i.e. represent relationships between entities. Attributes may be set-valued and aggregates of simpler labeled values. Attributes may also be given categories, i.e. they may be said to be 'CHANGING', 'UNCHANGING', or 'UNIQUE', with obvious meanings. In Figure 4.7, we see the definition of customers in a bank model. Note how enumerated classes allow the classification of customers to be specified precisely. Also note that constraints
TaxisDL DESIGN Banking_system IS

ENUMERATED CLASS Classification = { 'good', 'neutral', 'bad' }

ENTITY CLASS Customer WITH
   CHANGING name: Strings;
   CHANGING classification: Classification;
   CHANGING takes_up: SetOf Loan;
   .......... 
END Customer;

ENTITY CLASS Person_customer ISA Customer WITH
   CHANGING salary: Integer;
   INVARIANTS salary: This.salary > 0
END Person_customer;

TRANSACTION Withdraw WITH
   IN account_id: integer; amount: integer;
   GIVEN EXISTS a IN Account: a.id = account_id;
   GOAL a.balance = a.balance - amount;
END Withdraw;

END Banking_system;

Figure 4.7: The bank model in TaxisDL

(invariants) are specified in connection with entity classes. These constraints are specified in a typed first order language similar to relational calculus.

Transactions are used to query and update databases. A transaction is specified with parameters, entities that are consumed (deleted) and produced (inserted), a precondition and a postcondition, and possibly dynamic constraints. In Figure 4.7 a withdraw transaction is modeled. 'GIVEN' indicates a precondition, while 'GOAL' labels a postcondition. Since both preconditions and postconditions are expressed in relational calculus, transactions are generally not executable. However, if we place some restrictions on the types of postconditions allowed, transactions in TaxisDL may be executed. Hierarchies are not used in modeling of transactions, and algorithms can not be easily specified. The latter is made with the intention that implementation decisions should not be made in the design stages.

Scripts are used for modeling of long term processes. They are built around Petri net skeletons of states connected by transition arcs with condition/action rules. These rules may refer to the passage of time, they may initiate transactions, and they may exchange messages with other scripts. Both transactions and scripts are organized into classes, which can be related by generalization relationships according to their generality. They can also be considered as entity classes, with their different parts (parameters, preconditions etc.) regarded as attributes, which can be accessed in the same way as attributes of entities.

In [83], Meirlaen et al. describes an executor for the semantic datamodel and transactions. TaxisDL designs are mapped to Probe (PROLOG with object extensions) and PCL (PROLOG constraint language), and executed. There is some support for understanding executions, since transactions and scripts can be accessed after and during their executions.
4.6. Some Selected Languages and Environments

![Diagram of GSP(Account)]

Public:
- Newaccount(id, owner, interest), Dep(id, owner, amount), Withdraw(id, owner, amount)
Protected:
- id, Int, Bal, Owner

**Method: Deposit**

Id, Bal, Owner: Places for attributes Id, Bal, and Owner respectively.
L_D: Goal place for invalid deposit.
C_D: Goal place for completed deposit.
ISP(Trans): Call to method Store_transaction in class Trans.
Dep: Starting point for method Dep.
Ok_d: Place for verified, but not processed deposit.

![Diagram of Method: Deposit]

Figure 4.8: The bank model in eG-Nets

**Conceptual Modeling Using eG-Nets**

Extended G-nets is a language intended for modeling of complex information systems [36]. It is one of few executable CML's with an object-oriented perspective. Object-orientation is supported through constructs for object classes, inheritance hierarchies, and object interactions. This language is also based on Petri nets.

An object class is specified through its Generic Switch Place (GSP), which gives the class a name, and defines its private and public attributes, its private and public methods, as well as possibly a superclass. The upper part of the bank model in Figure 4.8 gives the GSP for the class Account. It has four private attributes, i.e. an account identifier, an interest rate, a balance, and an owner. Methods are provided to open new accounts, to make withdrawals, and to make deposits.

The internal structure of the object class specifies one net for each of the methods of the class. There are places for persistent storage of attribute values across method invocations, and places serving as starting points for the methods, which are filled with a token corresponding to the method's parameters. There are also so called goal places which, when they have tokens, indicate the end of the execution of a method. For object interaction, special places called Instantiated Switch Places (ISP's) are used. A token in such a place
means that a call to a method in another, or possibly the same object, should be made. The token specifies which object to interact with, which method to call, and the parameters of the method. When the call has been completed, the token has values from returned parameters, and the execution within the original net continues. Note that in general, tokens are tuples of values. These are indicated in the net, and used in the precondition specifications of the transitions.

In the bank model, we have modeled a net for the deposit method. First, the new deposit Dep is checked against the identifier (Id) and owner (Owner) of the account. One transition is for an invalid transaction and results in a token in the goal place l.D. The other transition accepts the deposit, and increase the balance subsequently. After that, a call to a method of an object in the class Trans is made, in order to store the new transaction and issue a statement to the customer. After that, a token for the completed deposit is unconditionally output to the goal place C.D.

Execution of eG-Nets is by interpretation. During execution, class instances are created as processes. As eG-Nets have a basis in Petri net theory, many of the analysis techniques are easily adapted.

4.7 Conclusions

In Chapter 2, we referred to studies showing that many CML’s are very similar. Through the presentations of concrete languages in this chapter, we find that this is indeed so.

For modeling of state spaces, we find that the most expressive languages offer semantic datamodels with many common constructs: Classification, generalization, aggregation, structural constraints, constraints expressed in sublanguages similar to relational calculus etc. We also find that some languages also offer instances to be modeled in a conceptual model (Statecharts, Gist). Statecharts and eG-nets have a somewhat weaker expressiveness for modeling of state components than do the other four. Note that only BNM offers modeling of arbitrary complex attributes, while ARIES is the only language which supports multiple inheritance. TEMPORA supports modeling of temporal aspects in the most powerful way. Only in TaxisDL general state laws can be explicitly connected to the entities and attributes which changes should lead to an evaluation of the state laws.

For modeling of system dynamics, we find that a wide range of control structures are offered, and that decompositions in hierarchical sublaw relationships are widely used. For instance, in Gist, we find the traditional control structures sequence, iteration, and choice. In PPP, PLD’s have similar constructs. However, in a PLD model, the receive construct may not be executed at the same time as it is encountered, since the input flow may not be available. In these cases, the execution will be suspended until a flow is received.

Another commonly found control structure is the spontaneous activation of dynamic laws as their precondition holds. In fact, this is found in some form or another in all the presented languages. This feature allows concurrency to be modeled in a natural way. For instance,
4.7. Conclusions

Petri nets have been used to model distributed and concurrent systems. However, there are some characteristics which distinguish different variants of this control structure. Whereas languages based on Petri nets only allow a single transition to take place at a time, other languages apply all dynamic laws which have preconditions met in a state. This is the case for action rules and derivation rules in TEMPORA, and for state transitions in Statecharts. In the former case, if more than one transition is applicable, which one should be chosen? Petri nets have a non-deterministic behavior which must be resolved somehow during execution. Different strategies may be followed. For instance, the user could be asked which transition should be fired, a transition could be selected based on some assigned priorities, or the selection could be based on random choice.

Yet another question related to spontaneous activations is: Should a dynamic law be allowed to be applied several times, i.e. each time it is not already executing and its precondition is true? For instance, during execution of a PrM model, a process may be triggered several times during the execution of its super process. On the other hand, one way of expressing process logic is to use production rules. It would be meaningless to execute a production rule repeatedly to obtain exactly the same result each time, or rather, it would have no effect on the state when it has been applied before.

None of the languages surveyed support all the control structures described above. For instance, ARIES and TaxisDL lack the expressiveness to model hierarchies of dynamic laws, e.g. process hierarchies. Even if such hierarchies can be modeled in BNM and TEMPORA, they are not incorporated in the execution of models expressed in these languages. Neither BNM nor TaxisDL can express algorithms like those found in e.g. PPP and ARIES (methods). In TEMPORA, non-determinism is not offered, as done in all the languages which are based on Petri nets. A general problem of languages based on Petri nets is that they can not express hierarchies in an explicit way, unless transitions may be decomposed.

Given our goal of providing very general primitives for executable CML's, how should we provide all these ways of specifying system dynamics? One obvious alternative would be to analyze a large set of languages, and simply provide all those control structures found there. Then, if a new language comes along, perhaps a new control structure would need to be added to the set of primitives. Another alternative is to analyze the control structures to find an underlying principle in which they can be specified and implemented in an executor. The latter approach would certainly be more powerful and flexible, and it is this approach we pursue in Chapter 7.

Besides language expressiveness, we have taken a brief look at model executors and techniques supported to understand model executions. We found that only a few environments support the different execution techniques, and even fewer have any additional support for execution understanding. We return to this latter issue in the next chapter.

Finally, it is very rare to see that executors which use model translations exploit general translation facilities.
Chapter 5

Execution Tracing in CASE Environments

In this chapter, the focus is on the use of execution traces for explanation of model behavior. We first present different utilizations of execution traces, and we state some general requirements to a tracing component. Then we review some major principles followed in what we can call a database approach to execution tracing. We briefly describe the question types which are made available to the users, and the general principles employed for generating explanations in response to these questions. Here, we draw much on experiences from expert systems research. Finally, we present some explanation generation systems, and some general tracing components. We conclude that the existing tracing components should be improved in order to accommodate the needs from explanation generators and adapted for tracing of conceptual models.

5.1 On the Use of Execution Traces

From the literature, we find that execution traces are used for various purposes.

Program Debugging When used for program debugging, execution traces of programs can be inspected to search for errors made. This is especially needed for concurrent programs which communicate with each other, and together form very complex behavior (McDowell et al. [82]). More advanced use of traces than mere inspection, are program visualization and replay of executions. Replaying of executions are necessary because of the possibly non-deterministic behavior of time-critical concurrent systems. The traced information may represent e.g. updates of variables, interprocess communications, procedure calls etc. When an error is detected, the execution trace is compared with the program to identify the cause(s) of the error. This search is normally manual, guided by some hypothesis made beforehand. Some tracing components provide effective means for
browsing through traces, or for retrieving trace information much like database queries.

**Paraphrasing System Dynamics** Execution traces tend to be very large and complex, and the search for useful information is difficult. One way of describing the model dynamics, is to execute the model, record the occurring events in a trace, and then paraphrasing interesting parts of the execution trace. In Gist [115], symbolic execution traces are paraphrased by a 'behavioral explainer'. It analyzes traces to find interesting fragments, and builds up an explanation structure which is subsequently paraphrased into English text. The main problems are to select relevant information from the traces, and also to paraphrase proofs. The former problem is solved by applying a set of heuristics, each which recognize a situation which is deemed useful for the developers or users to know about. The latter problem is approached by searching for instances of well-known inference rules like modus ponens etc. There are no possibilities for users to ask specific questions, rather, they must rely on the heuristics put into the explainer.

Another example of paraphrization of execution traces is presented by Kalita in [63]. There, natural language status reports are generated from a set of inter-related processes described by Petri-nets. This system is similar to the explainer in Gist, but it is based on concrete rather than symbolic executions.

**Validating Behavior** Benner [11] describes the use of execution traces in the ARIES simulation component. He uses *validation questions*, which are patterns of behavior, to search for desired or undesired behavior from the trace. The patterns are sequences of events to be searched for in the temporal order they are given. The patterns contain variables which are instantiated when the patterns are matched with the trace being produced. A fully instantiated pattern means that the specified behavior is observed, be it a desired or undesired behavior. If a desired behavior is only partially matched, or not matched at all, this means that the model has errors.

**Explanation Generation** Explanation generators were introduced in Chapter 2. They may be used both in the development phases for model validations, and in the final system in order to help the end-users. Most research on explanation generation relevant to our work has taken place in expert systems research, e.g. Basu et al. [10], Chandrasekaran [25], Clancey [29], and Neches et al. [87]. As we will briefly describe later on, these explanation generators are able to explain more than execution traces. However, we will focus on their ability to recognize and answer a range of questions related to the behavior of executing models.

### 5.2 Requirements to Tracing Components

In this section we sum up some major requirements to rather general tracing components. Many of existing explanation generators do not have separate tracing components, rather
some intertwine the problem solving status with their knowledge structures e.g. in RATIONALE [1]. However, these traces tend to be rather simple compared to traces from executing conceptual models.

**Language Independence** A general tracing component should be adaptable to different modeling languages and model executors. This can be achieved by either basing the execution trace on a single, but very general schema, or by allowing different schemas to be specified for each application. Note that schema here is used in a general sense, it simply gives an intentional description of execution traces.

**Expressiveness** Naturally, all information deemed necessary for later retrieval should be traceable. The complexity and detail level of executable conceptual models is comparable to that of programs. Hence, we should study the tracing techniques used in programming environments. The information stored in traces includes state changes, the reasons for state changes, and dependencies and relationships between state changes. The latter includes temporal and hierarchical relationships.

**Event Reporting** It should be easy to report information about relevant events to the tracing component, without disturbing the behavior of the executing model. Also, it is important that the temporal ordering of events is maintained in the trace.

**Information Retrieval** To make use of the stored execution trace, it must be possible to search for information in an effective way. For large and complex execution traces, there should be facilities going beyond mere inspection. As we noted above, different browsers and query facilities have been developed to study program traces. For explanation generation, particular queries should be made available, so that information to include in explanations can be easily retrieved. The form of queries supported should be based on the questions users may ask about model behavior, and it should be possible for a developer to use queries directly to understand executions based on trace information.

**Non-functional requirements** These concern the performance of the tracing component, including efficiency and reliability.

### 5.3 General Tracing Principles

The architecture depicted in Figure 5.1 represents what may be called a database approach to tracing. It is more close to the way execution tracing is performed in program tracers than to the way it is performed in most current expert systems.

In the architecture, a model executor reports information about occurring events as necessary, for instance by calling specialized routines for each event category. At the program level, events typically correspond to memory accesses, message interchange, object access, procedure calls, tasking activities etc. In an expert system, events may correspond to e.g. application of rules, conclusions reached, and hypotheses refined. In an executing
conceptual model, events reported may be about information flows sent, processes or activities applied, updates of state components etc. Usually, a fixed set of events is defined, but some systems provide event definition languages so that they can be more flexibly specified. This tend to create a considerable overhead, because events need to be detected at runtime.

Also, the detail level and the amount of information to be stored depends on the future use of the trace. For instance, to replay executions in non-deterministic situations, it is enough to store information which will resolve the non-determinism. However, in order to explain execution behavior, it is expected that very detailed information about the executions must be stored.

In any circumstance, the information to be stored must somehow be reported. Usually, this is done by inserting probes into the executing model. The probes are calls to routines which transfer information to a trace recorder, or they store information directly into the trace. To do this, they must be placed at appropriate locations in the model. If the models are translated to prototypes automatically, it is preferable to generate the probes as well. If the models are directly interpreted, then the probes may be integrated with the interpreter. In some cases, the execution of probes may change the behavior of the executing program or model. This may particularly be so in time-critical, concurrent systems, as noted by McDowell [82]. We will not study this problem here, in fact in our use of execution tracing we will avoid it completely.

Continuing towards the right of the architecture, the reported events are stored in the order they occur by the trace recorder. The information is stored according to a predefined schema, which includes attributes for temporal ordering of events. Many systems use a global clock in order to achieve the ordering, and mark starts and ends of intervals. This clock may represent the actual time, or provide a state numbering, e.g. each time an event is reported, the state number increases.

Finally, in the database approach, queries are used to specify the information to be retrieved from the trace. For explanation generation, these queries form the interface to the execution trace. Also other tools could use the execution trace through the interface provided with trace queries, and if they are made user-friendly, developers can use trace queries directly.
5.4 Principles for Explaining Model Behavior

Although we do not intend to develop an explanation generator ourselves, we will give a brief description of how they work. This will enhance our understanding of their interfaces to execution traces, and the requirements to retrieval mechanisms resulting from the query types offered.

Explanations and Question Types

In Chapter 2, we argued for the use of explanation generators in order to validate executable conceptual models. Broadly speaking, an explanation can be defined as

\[ \textit{a description which enable one to understand a certain phenomenon} \]

(Lindgreen et al. [92])

In our context, the phenomena to be understood are related to the behavior of executing conceptual models, and the explanations are most often textual descriptions. The receivers of the explanations may be both users and developers, and it is essential that these receivers can understand the given descriptions within a certain \textit{framework of explanation} shared with the explanation generators (Rolston [103]). A natural framework would be some basic understanding of the constructs of the modeling language used. If the receivers are unfamiliar with these constructs, information about their meaning must be included in the explanations.

A great advantage of explanation generators is their ability to relate observed behavior to the way the model is built, and point out those portions of the model which are responsible for the behavior. The explanations are produced automatically from users’ or developers’ requests, and the generators employ knowledge of what information is necessary to explain certain phenomena.

When it comes to the concrete questions which can be asked, these are adapted to the constructs of the modeling language used, and their semantics. As noted above, we had to turn to expert systems research in order to find systems which can explain behavior. Here, we also find different classifications of question types. Since the problem solving state of expert systems is comparable to the states of executing conceptual models, we can use these classifications as a starting point in order to illustrate the types of questions which should be offered.

There exist several classifications of questions in the literature, e.g. Abu-Hakima [1], Chandrasekaran [25], Neches [87], and Wick [133]. Here we use a slightly modified version of the one found in [1]. This is the most general classification, i.e. it subsumes the others with the addition made here. The following types of questions are mentioned:
Event questions These questions relate to the actual problem solving process carried out by the expert system. Typical events are conclusions given by the system, and requests posed to the users. More specifically, this type includes questions like 'Why must an input be given?', 'How was a conclusion obtained?', 'Why was a conclusion rejected?', and 'Why was a special strategy followed?'.

Hypothetical questions These refer to possible, but not actual states of the problem solving process. Questions of this type include: 'What conclusions would be inferred if a particular input is given?', and 'Which strategy would be followed if a particular input is given?'

Other questions Ability questions clarify for the user what the system is able to do, e.g. 'How can conclusion C be reached?'. Factual questions correspond to queries to the knowledge base, e.g. 'What are the preconditions of rule R?'. Finally, Knowledge justification questions ask for deeper justification of shallow, compiled knowledge which is used for efficiency reasons.

Of the above types, the former two are relevant to our work, as they address model behavior directly. The explanations generated in response to these questions should make use of execution traces.

We can easily translate questions of the two former categories into questions related to conceptual models. Consider the bank model represented in the PPP languages from Chapter 2. Typical event questions and hypothetical questions related to the decomposition of transaction processing in Figure 2.16 would be as shown in Figure 5.2.

Input justification: 'Why must New_withdrawal be given?'
Result justification: 'Why was Withdrawal_rejection generated?'
'Why was not Ok_withdrawal generated?'
Higher level strategy: 'Why was Verify_transaction executed?'
Hypothetical: 'What would happen if New_transaction was negative?'

Figure 5.2: Questions to execution of the bank model

Two Phases of Explanation Generation

A general architecture of explanation generation systems is depicted in Figure 5.3. It is primarily based on the descriptions of the EES system given in [87], which is one of the most comprehensive explanation generation system existing. The architecture has been somewhat geared towards explanation of conceptual models. Generation of explanations can be divided into two separate phases: Content generation and presentation. The former determines which information to include in the explanations, and the structural relationships between different pieces of information. The latter turns the resulting explanation structure into the final description which is presented to the users. This explanation will most often be textual, but if a graphical CML is used, it may have graphics included as
well. Although presentation is an important issue, we will not discuss this phase any further here.

The sources used for explanation generation, and the principles used in generating the contents of explanations, varies much from system to system. Most explanation generators in expert systems have so far been based on rather simple principles like canned text and template approaches. In the former approach, the content generator simply orders predetermined text strings connected to the conceptual model. For instance, an input justification could be given at a very general level, just stating what the input will be used to produce. Templates are structures of text and variables. By finding proper instantiations of the variables, an explanation is generated. Note that in both these cases, the presentation phase is reduced to simply displaying text on the screen. Examples of canned text and template approaches are found in e.g. Clancey's MYCIN system [29], and in the generic task based systems by Chandrasekaran [25]. The more sophisticated explanation generation systems are based on explanation strategies. Here, strategies are defined for each question type offered, and strategies at different abstraction levels are combined in a goal-oriented manner to produce a hierarchical explanation structure. This is a very flexible approach, since the strategies can take into account who the receiver of an explanation is, and the context in which a question is asked. These systems tend to use all the information sources depicted in Figure 5.3.

The execution trace provides information at the instance level about processes executed, flows produced etc. The conceptual model provides information at the type level, and the language metamodel should include semantical information about the constructs of the modeling languages used. The user model contains information about knowledge and preferences of the receivers of explanations. Using this knowledge, the contents can be adjusted to different types of users and developers. For instance, a developer is likely to be interested in more detailed explanations than would an end-user. Finally, contextual information gives information about the status of a dialogue taking place. This allows users to give follow-up questions to the explanation generator. The latter two sources are
Question: Why must New_withdrawal be given?
Possible answer: Verify_amount is a process which transforms input flows to output flows. New_withdrawal is an input flow which is needed to produce either Ok_withdrawal or Aborted_transaction. New_withdrawal must be received because the requested withdrawal amount was too large. The requested amount was 100, and the available amount was 80.

Figure 5.4: Answer to an event question

rarely used, and from the systems we surveyed, only EES by Neches et al. [87] had such capabilities.

Drawing upon the three former sources of information, we can give an example of a possible answer to one of the questions listed to the execution of the bank model above, although there exists no explanation generators available today which works with conceptual models. The example is given in Figure 5.4.

Note that in this explanation, the first sentence includes information from both the language metamodel and the conceptual model. The two next sentences include information from the conceptual model, whereas the last sentence includes information from an execution trace. If it is known from the user model that the constructs of the language is well-known, perhaps the first sentence is not necessary.

Some Explanation Generators for Expert Systems

Rule-based Systems In the typical rule-based expert systems, the execution trace shows the inferences made when the system tries to solve a problem. The conclusion of one rule is used as the premise for application of another rule. These structures can be inspected to explain to the user why certain inputs are needed by the system, or how a particular conclusion was obtained. The most cited system based on this kind of explanations is MYCIN [29], which was used in diagnostic problem solving. It also provided a general query language for simple factual and ability questions. MYCIN used a combination of canned text and templates to generate explanations.

Other rule-based systems have expanded the range of questions to also include Why not questions as well as hypothetical questions. Also, Wick introduces additional questions related to timing in [133], e.g. When was a particular piece of information given?.

Although very useful, these systems suffer from a number of limitations (Rolston [103]). First, only a very limited number of predefined question types are offered. Further, no knowledge about the context or the user is taken into account. Finally, since the rules represent compiled knowledge, no deep explanations can be provided.

NEOMYCIN [29] was an improvement of MYCIN intended to represent control structures explicitly. It used meta-rules to control the problem solving process, selecting tasks or
rather portions of the ordinary rule-base for execution. These metarules were used to explain problem solving strategies. The other weaknesses mentioned above still applied.

**Generic Task Based Expert Systems** Chandrasekaran [25] and Tanner [117] use *generic tasks* as building blocks in expert systems development, as well as to explain problem solving behavior. Several generic tasks have been identified, including hierarchical classification and design by plan selection and refinement. Common to these tasks is the hierarchical structure of 'specialists', which decompose problems into subproblems. For design, these specialists are called 'design specialists'. They contain knowledge to complete a portion of the overall design. They have a number of different plans available for the design problem. A 'sponsor' is connected to each plan, and evaluates the advantages of choosing the plan in the current situation. The specialists choose among plans by using a 'selector'. Each plan is a sequence of steps, each step being designed by a specialist at the next level in the hierarchy.

Explanations can be given to both event questions and knowledge justification questions. For event questions, the principle is to let each problem solving agent (specialist, plan, selector, sponsor) explain its own behavior. 'How'-questions are answered by referring to a context, i.e. from which agent the agent was called and for what purpose, values of any given variables, and the result. Follow-up questions are allowed, and answered by the agents which provided the values asked for.

Strategies are explained from plans. A standard plan has a structure consisting of a name, a sponsor name, a purpose, an achievement, and a sequence of steps which are performed to make its achievements. To explain a strategy, a plan explanation template is used. The structure of this template consists of three parts. The first part lists all achievements of the plan steps up to the current step, the next part simply states that the current step is being executed and its purpose, and the third part lists the remaining steps of the plan with their purposes.

**RATIONALE** RATIONALE [1] is a framework for building diagnostic expert systems with explanation facilities. Domain knowledge is represented as a hypotheses hierarchy, and a system reasons by activating and rejecting hypotheses based on observed symptoms. The term *explicit reasoning* is used to denote reasoning which refers to a causal domain model as well as to explicitly represented strategic or heuristic knowledge. A RATIONALE system reasons explicitly in this sense, since the strategy is basically that of top-down hypotheses refinement. Causal domain knowledge includes knowledge about what states suffice to activate or deactivate a hypotheses, what states constitute exceptions to activation, about what states lead to activation of alternative hypotheses etc.

We will not go into details of the reasoning mechanism here. However, every problem solving activity leaves behind a *reasoning trace*, which is simply an instantiation of part of the hypotheses hierarchy. All conditions about hypotheses activation, rejection etc. are traced, and so reasoning can be said to coincide with explaining. The trace is an *explanation tree*. The explanation tree is used to answer event questions like *Why is a symptom
needed? and Why was a hypothesis activated/rejected? Hypothetical questions like Which hypotheses will be activated if a particular symptom is observed? is answered by first generating a parallel trace, and then inspect it in the same way as for event questions.

RATIONALE uses templates to generate explanations. Its most distinguishing feature is the way it deals with question input and understanding. Unlike most other systems, it uses a graphical interface. The possible questions are adapted to a particular problem solving context. The user first chooses a hypothesis, symptom etc., and then a generic question which will refer to the chosen context. This is a nice way to deal with question understanding, as well as it informs the user which questions may be asked at any time.

EES  The main principle behind EES [84, 87] is that comprehensive explanations require that design knowledge used in developing an expert system must be explicitly represented. This means that not only the compiled knowledge used in the actual problem solving process can be accessed as with rule-based systems, but also the underlying deep knowledge which was used to derive the problem solving routines. A program writer creates what is called a development history, which is a hierarchical structure comparable to a proof tree. The leaves of the structure constitute the steps of the problem solving routine, and the interior shows the underlying knowledge structures. The root corresponds to the solution of a problem given to the expert system.

All question types except hypothetical questions are offered. Explanation generation is performed by a planner which uses a top-down hierarchical planning mechanism to achieve certain discourse goals, depending on the questions asked. Plan operators are formulated to describe how the discourse goal is achieved by direct inclusion of information from the knowledge base, execution trace etc., and possibly by achieving subgoals by means of lower level operators. Any way, it also determines the structure of the informations or subexplanations included. The resulting explanation is a hierarchical structure of plan operators, where the leaves correspond to information from the execution trace, the development history, and the knowledge base. In EES, both user models and dialog status are taken into account when the explanation is produced. The resulting structure is input to a text generation system which produces the presented explanation in natural language text.

A Database Approach to Explanation in Knowledge-based Systems  Most explanation generators access execution traces stored in main memory. However, if the number of facts in the trace becomes large, it may be better to store the trace in a database. This is the motivation for the work by Basu et al. [10], where traces are stored in a relational database. They assume a Horn-clause representation of knowledge, where fact statements are relations, and rules are clauses consisting of a single statement as the head, and a number of statements in the body.

The idea is that the resolution proof tree created during problem solving can be represented by a set of relations. The proof tree will then have the final conclusion as its root, and facts and failed statements as leaves. Interior nodes are statements found in rules. The schema contains a relation Trace which represents the proof tree. It has the following attributes:
5.5 Example Program Tracers

Stageno: A sequence number unique to each node.
Status: If the node succeeded and contribute to the solution, it is 't'. If the node failed, it is 'f', and if it succeeded, but not used in the final solution because of backtracking, it is 'a' (aborted).
Parent: The stage number of the head of the rule of which this statement is an antecedent.
SName: Name of statement.
RuleNumber: Number of rule which has statement as antecedent.

Further, to store data about each statement, one relation per statement in the knowledge base is defined. It also includes an attribute to link it to a statement in the proof tree. Finally, one relation is defined to link a solution to all the statements which contributed to it.

With this representation, different explanations can be given by specifying queries to the database. Since a rule may have different formulations, the children nodes of a statement may correspond to different relations, not known as queries are specified. This problem can not be handled by SQL, so an extended SQL was used for explanations (ESQL). As an example of an explanation, consider the problem of explaining why a conclusion is given. A possible way of explaining this, is to show that all antecedents of the rule with the given conclusion are proved. Starting from the root node, all children nodes are found, and they are joined with their corresponding statement relations.

As this tracing component follows a database approach, we can compare it to the requirements identified above. First, the approach is language dependent, since it is based on Horn-clauses. The statements relations can contain data from ground facts which are represented as relations. No other data structures are considered, and also it does not consider updates of relations as is necessary for tracing of executing conceptual models. The actual reporting of events is not discussed in the paper. ESQL provides a powerful retrieval mechanism, but can not be used on its own as an explanation generator, at least not for the end-user of the expert system.

5.5 Example Program Tracers

A Temporal Database Approach to Program Monitoring

According to Snodgrass [109], monitoring is a necessary first step in understanding a complex computational process, because it provides an indication of what happened during program execution. In order to record dynamic information, a uniform yet comprehensive representation is needed. The representation chosen in [109] is a temporal relational data model, i.e. users are presented with the conceptual viewpoint that the dynamic behavior is available as a collection of temporal relations. Having stored desired facts during program execution, a temporal query language is used to retrieve necessary information.
Example tables:

<table>
<thead>
<tr>
<th>Caller</th>
<th>Callee</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main</td>
<td>DoTypeDec</td>
<td>15</td>
</tr>
<tr>
<td>DoTypeDec</td>
<td>InsertSymbol</td>
<td>130</td>
</tr>
<tr>
<td>DoTypeDec</td>
<td>InsertSymbol</td>
<td>426</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Start</th>
<th>Stop</th>
</tr>
</thead>
<tbody>
<tr>
<td>DoTypeDec</td>
<td>15</td>
<td>923</td>
</tr>
<tr>
<td>InsertSymbol</td>
<td>130</td>
<td>158</td>
</tr>
<tr>
<td>InsertSymbol</td>
<td>426</td>
<td>466</td>
</tr>
</tbody>
</table>

Example queries:

- range of E is EXECUTING
- retrieve InsertExec
- where E.Procedure='insertsymbol'
- range of l is InsertExec
- retrieve DoTypeDec_Insert
- where E.Procedure='DoTypeDec'
- when E overlap l

Figure 5.5: Example tables and queries in Snodgrass' approach

In order to capture dynamic information, sensors are placed in the users programs. These are code fragments which transfer information to the monitor according to a defined schema, one schema defined for each sensor. There are two kinds of sensors, traced and sampled. A traced sensor initiates a transfer of information to the monitor when an event is detected, whereas a sampled sensor must be requested by the monitor to return information. In any case, information sent to the monitor includes system dependent information e.g. a sensor id, a timestamp for the event, and any information defined by the application in question.

There are two kinds of temporal relations, event relations and interval relations. An event relation records a particular state change at a particular point in time. An interval relation represents relationships which are valid during a period of time. An example of the former is the CALL relation which shows how and when procedures call each other during execution. The latter is exemplified by the EXECUTING relation which records when procedures are active. An event relation contains one attribute to represent the time of the event, whereas an interval relation must include two time attributes, one for the start and one for the end of the interval.

From the primitive relations connected to sensors, derived relations can be specified using a temporal query language. This language provides a powerful mechanism for specifying which information is wanted from the program execution. The language used in [109] has been thoroughly described in [110]. It is an extension of the Quel language used in the Ingres DBMS. For program monitoring, the when clause is particularly useful. This is a temporal version of the 'where' clause used in Quel and SQL. It specifies additional requirements to the information retrieved, based on the temporal attributes included in each relation. These requirements are temporal predicates which include operators like 'precede', 'overlap', and 'equal'.

We adopt an example given by Snodgrass for illustration. The example tables in Figure 5.5 are based on the relations CALL and EXECUTING introduced above. CALL is an event relation and has only one time attribute for the time of the event. From the table we see that
procedure 'Main' called procedure 'DoTypeDec' at time 15. The time attribute is not manipulable, it can only be used for temporal specification. Similarly, since EXECUTING is an interval relation, it must have two time attributes. 'Start' tells when the procedure started execution, while 'Stop' tells when the execution finished.

TQuel can now be used to retrieve derived information from these primitive relations. For instance, to find when the 'InsertSymbol' procedure was executing, the query to the left in Figure 5.5 can be given. InsertExec will be a derived relation with the two instances of EXECUTING which has 'InsertSymbol' as the procedure attribute. To determine when 'InsertSymbol' became active when called by the 'DoTypeDec' procedure, the query to the right can be issued. The last query illustrates use of the 'when clause'. It restricts the tuples retrieved to those which have overlapping intervals. Only those executions of InsertSymbol which overlap in time with executions of DoTypeDec are retrieved.

Comparing Snodgrass' approach with the requirements identified above, we find that it is general. It is language independent, since schemas can be specified as desired. The expressiveness of relations may be sufficient for many applications, although this issue is not discussed in the paper. The use of sensors seems to be a straightforward and reasonable way to report events. TQuel is a declarative and general purpose query language which can be used to retrieve trace information. Having said this, we find that no guidelines are given as to what program events should be traced. Only dynamic program information like procedure calls and executing procedures are recorded in the examples. How to record state changes of state components and relate these changes to dynamic program information are open issues. To fully understand behavior, such information should be stored. Finally, no guidelines are given as to what queries should be offered to enhance the understanding of programs.

Saving Traces for Ada Debugging

LeDoux [72] views the execution history of concurrent Ada programs as a stream of events. Figure 5.6 shows the important components of his YODA\(^1\) system. A program annotator parses Ada programs and instruments them with probes for recording of events. This can be done automatically based on a fixed set of events defined to be interesting for program debugging. These events relate to intertask communication, task initiation and termination, read and updates of variables etc. All events can be represented as relations. The annotated program is then run, and for each occurrence of a defined event, the inserted probes sends information to a program monitor. This monitor converts the received information to PROLOG facts, and records them according to a defined schema. At runtime, the events are linearly ordered by time. A global clock is used, and this clock is increased one tick for each event. An interval-based temporal logic is used as a basis for the query processor. All queries are written in PROLOG, with temporal relations like during, before and after included. This provides a flexible and powerful query facility.

Basically, YODA employs many of the same principles as in Snodgrass' approach, how-

---

\(^1\)Your Own Debugger for Ada
ever the application is restricted to tracing of Ada programs with concurrency. As such, the approach is language dependent, but it could be generalized to be used with other languages. The trace includes information about updated variables, but these are of simple types. The problem of tracing more complex data is not considered. Probes are inserted into the code automatically, which frees the programmers from thinking about how events should be reported. Finally, as PROLOG extended with temporal operators is used for trace querying, general retrieval mechanisms exist. It is however up to the programmers to define the queries deemed useful for program understanding.

5.6 Conclusions

In this chapter, we have studied the general principles and requirements to a tracing component. We have also reviewed explanation generation as a technique which enhances the understanding of executions, relying on trace information as a main input. This technique is however so far not exploited in CASE environments. Also, tracing components are rarely seen in upper CASE, although quite a few research environments support an executable CML. We find the most advanced tracing components in programming environments, and these often follow a database approach. They are however not directly applicable for tracing of conceptual models, for the purpose of explaining model behaviors. In particular, to determine what information to store, how to store it, and what queries should be supported for understanding executions, are open issues.
In Chapter 6, we describe an architecture of components which will support the modeling cycle framework introduced in Chapter 1.

In Chapter 7, an executable CML is proposed, which will serve as a internal target language for translations from a wide range of CML's.

In Chapter 8, we propose a new language for translation specification and implementation.

In Chapter 9, a tracing component is proposed, which can store execution traces and retrieve trace information from declarative queries expressed in a trace query language. Its interface to an explanation generator is briefly described.

In Chapter 10, we present the core algorithms for the interpreters developed for the proposed languages.

In Chapter 11, we propose an integration with the PPP CASE environment through a specification of translations from PPP models to the internal executable language which was described in Chapter 7.
Chapter 6

Support for an Integrated Approach to Conceptual Model Validation

In this chapter we propose an architecture of an environment which represents our answer to the research questions posed in the first chapter in this thesis. Basically, it involves components for conceptual modeling, model translation, model execution and tracing, and explanation of model behavior. It is clearly stated which of these components are addressed in the thesis, and which provide a context for the work. We describe how the architecture supports the modeling cycle introduced in Chapter 1.

6.1 Introduction

Having presented the potentials and limitations of current ideas in relevant research areas, we now present an architecture of components which together form the basis for an integrated approach to validation of conceptual models. It can be seen as an attempt to answer the research questions in this thesis on an architectural level. The architecture integrates components for conceptual modeling including a metamodeling component, components for model translation, components for model execution and tracing, and components for explanation of behavior observed during model execution. The former three groups of components and their integration are the means by which we want to support building CASE environments which include executable CML's, i.e. they constitute our answer to the first research question posed in the introduction of this thesis. The latter group of components and its interfaces to the three former groups are the means by which we explain observed behavior during execution of conceptual models, i.e our answer to the second research question.

The architecture is shown in Figure 6.1, and will be explained in the following sections. Some of the components are shaded, which means that they will only provide a context for
our work, and hence they will not be addressed in detail in the chapters to come. However, their interfaces to the components we focus on are important. The presentation of components will follow the modeling cycle introduced in Chapter 1, starting from components for conceptual modeling. For the next three groups of components we seek to attack some of the problems identified from the state of the art, however we consider the integration of components to be an achievement in its own right.

6.2 Support for the Modeling Cycle

Conceptual Modeling  We assume that the modeling languages used are specified by language metamodels. These may coincide with the repository schema, or at least they form the basis for this schema. The metamodel includes constraints to be satisfied, the presentation of language constructs and relationships etc., as described in Chapter 2. Although it is possible to support multiple or customized languages without a separate metamodeling tool e.g. ARIES [62], it is certainly easier with such a tool.

The tools for conceptual modeling need access to the language metamodels in order to verify the models. We assume a tool with the usual functionality for editing and verifying conceptual models. In order to test our ideas, we use the PPP environment. The results we will provide are still more general, and it should be possible to include the components in other environments.
6.2. Support for the Modeling Cycle

Model Translation  The translation specification tool supports the editing and checking of translations expressed in a translation specification language. Doing so, it needs access to the metamodels of the source and target languages involved in the translation, for syntax check of the specifications. The translation specification language we propose is rule-based, and integrates previous ideas in order to deal with the generation of prototypes from conceptual models. In particular, the mixed representation problem outlined in Chapter 3 is taken care of. We name the language TRL (Translation Rule Language) for ease of reference. The specifications expressed in this language reference the language metamodels directly. It is assumed that manipulation of the repository is directly included in, or may be automatically derived from the high level specifications.

A translator implements the translations either by direct interpretation, or by generation of a procedural translation program. It will be used to generate prototypes from conceptual models, in particular from PPP models. However, the principles used are general, so both the specification language and the translator can be used for a wider range of translation tasks. We develop a simple interpreter for TRL. In Chapter 8 we present TRL, while an interpreter for the language is described in Chapter 10. The use of TRL in PPP is presented in Chapter 11.

Model Execution and Tracing  The target language for prototype generation will depend on the semantics of the source language. We argue that the complexity of data and control structures in many CML’s calls for target languages with more abstract constructs than those found in high level programming languages. To this end, we define a general executable conceptual modeling language ECML. The constructs of the language are derived from a unified metamodel which is based on the analyses in Chapter 2 and 4. Motivated by the variety of control structures found in executable CML’s, ECML is particularly powerful for modeling system dynamics.

We develop an interpreter for the language, and we discuss how execution mechanisms presented in Chapter 4 can be supported. The events happening during execution of a conceptual model can be recorded for later inspection. We classify the events which can occur, and provide a reported events handler which can be called to store event information. This information is stored in an execution trace based on a general schema, and can be regarded as a specialized database.

ECML is described in Chapter 7, and its interpreter in Chapter 10. The tracing components are presented in Chapter 9. Potential use of ECML in PPP is described in Chapter 11.

Explanation of Model Behavior  Dynamic properties of the conceptual models are made more explicit when the models are exercised, and the external behavior with required inputs and produced outputs is revealed. We provide a means to enhance the understanding of the external behavior through focused access to the history of internal events which have taken place and are stored in execution traces. A query handler accepts trace queries, and searches a potentially large trace for relevant information. We propose a trace query language TRQL which provides a set of views into the execution trace to provide quick
Chapter 6. Support for an Integrated Approach to Conceptual Model Validation

explanations of model behavior. Furthermore, it provides an interface to an explanation
generator developed by Gulla [47].

The trace query language is described in Chapter 9, and a prototype of the query handler is
described in Chapter 10. The explanation generator and its interface to the query handler
is briefly described in Chapter 9.
Chapter 7

Constructs for Executable CML’s - ECML

In this chapter we develop a set of language constructs which will be used as a baseline for prototyping from conceptual models. We refer to the set as a whole by the name ECML - Executable Conceptual Modeling Language. For a given CML, a subset of these constructs will be selected in the translation to a prototype represented in ECML. We first discuss the general requirements to ECML. We develop a unified metamodel, from which the overall, abstract ECML constructs are derived. The presentation is rather informal, and the language is illustrated by examples from the bank system introduced in Chapter 2. Finally, we specify the important parts of the operational semantics of ECML, and we describe a principle for specification of control structures in executable CML’s.

7.1 Introduction

We now review the requirements to CML’s stated previously, and emphasize on some aspects which are particularly important, given our objectives. We name the language we are looking for ECML.

Expressiveness  ECML must be highly expressive, but still have constructs at a high level of abstraction, matching the type of constructs found in CML’s. As we have discussed previously, this will facilitate easier translations for prototype generation. Importantly, different modeling perspectives should be offered in an integrated way, either by offering explicit constructs for each perspective, or by straightforward combinations of other constructs.

Executability  Of course, ECML must be executable. Moreover, it should let models be expressed in an implementation independent way. Also, incomplete models, or parts of models expressed in a CML should be made executable by appropriate translations to ECML.
Styles of expression State components and state laws are usually expressed in a non-procedural way. State transitions may be expressed both in procedural and non-procedural ways. Some languages include both styles, e.g. both algorithms and rules can be expressed in TEMPORA [78]. We conclude that languages with both styles should be translatable to ECML.

Explanations A particular requirement to ECML is that each construct instance which cause state transitions should be uniquely identifiable. During execution, each state transition and its cause should be traced in order to be able to explain the observed behavior afterwards. The explanations will refer to the constructs which brought about particular changes. As an example, references to processes in PrM models or to PLD-constructs in PLD models may need to be included.

Technical considerations Our choice of language constructs should not lead to intolerable performance. One way to improve the performance is to allow calls to functions written in more efficient languages. Another approach is to organize dynamic constructs into hierarchies, to focus the search for possible state transitions, as discussed in [124]. Another one is to focus on limited portions of the models at a time, as described by Benner [11]. Also, execution with a small number of data instances at a time increases performance. Of these approaches, only the former two ways have implications for the choice of language constructs.

Another technical issue concerns integration with CASE environments. The tools built to support ECML must be easily integrated with a CASE environment, in particular with the PPP environment which will serve as a testbed for us. Constructs must be chosen so that ECML models can be stored in the PPP repository, by extending its schema.

To sum up, ECML should be expressive at a high level of abstraction, it should be executable with acceptable performance, and accommodate both procedural and non-procedural styles of expression. Furthermore, to provide explanations, construct instances must be uniquely referenced. The tool support for the language must be easily integrated in CASE environments. Many general requirements to CML's apply to ECML.

Note that we do not search for the 'universal' language. Rather, we will choose constructs based on an analysis of existing languages and on analyses previously made by others. Our intention is therefore to provide a rich set of constructs which can cover a large class of languages. CML's on the borders of this class could still be translated to ECML, but not with the same ease as those within. As an example of expressiveness not required from ECML, we only require that the language is a first order language without modal operators. Hence, we exclude the treatment of temporal logic. Still, even if this is not explicitly supported, many kinds of temporal expressiveness in source languages can be supported by appropriate translations to ECML.
7.2 A Unified Metamodel and Derivation of Language Constructs

The approach we take to meet the requirements stated to ECML, is to first develop a unified metamodel which includes all the basic constructs and the relationships we wish to support. From this metamodel we then derive the constructs that will be explicitly offered by ECML, and we decide on the syntax of these constructs.

Developing the new unified metamodel, we try to exploit the analyses done previously on CML expressiveness, which were described in Chapter 2. Together, the unified models presented in Chapter 2 represent the unification of a relatively large number of existing CML’s. The problem with most of these is that they either lack essential constructs as was clear from the comparison made in Chapter 2, or that an operational semantics has not been defined. The latter is partly due to an insufficient level of detail in the constructs. We have to be able to define an operational semantics, since ECML models are to be executable.

Existing executable CML’s like those surveyed in Chapter 4 lack some of the generality required from ECML. In particular, this is the case for the control structures among dynamic laws. We should open up for a wide range of such structures in the unified metamodel and in ECML.

The Unified Metamodel

The unified metamodel is shown in Figure 7.1. In the following, we first describe the constructs and relationships for description of static system aspects. Afterwards, a similar presentation is given for the parts associated with dynamic system aspects.

Constructs for Modeling of Static Aspects

The constructs are as follows:

**State component** A state component can be uniquely identified, and has an associated state. Excluding executional aspects, the state of a system is the aggregate state of all state components in the system. State components correspond to entities, the static part of objects, named variables etc.

**Class** A class has a name, and consists of a set of state components which have common properties.

**Property** A property is a function which maps a state component to a value. The value of all properties of a state component in a class constitute the state of that state component.
Figure 7.1: A new unified metamodel

**Type** A type is an intentional description of state components. It has a name and a definition.

**Type definition** A type definition specifies a type in terms of other types through type constructors like sets, aggregates etc.

**Condition** A condition is a logical sentence which refers to states of state components and is either true or false in a given state.

**State law** A state law includes a condition which should hold in all states. It further has a name, it specifies the situations in which the condition should be evaluated, and it specifies what action should be taken if it is violated.

The relationships between the constructs are described informally in Table 7.1

** Constructs for Modeling of Dynamic Aspects**

The constructs are as follows:

**Dynamic law** Dynamic laws causes state transitions in a prescribed manner. It covers constructs like process, rule, activities, method, etc.
### 7.2. A Unified Metamodel and Derivation of Language Constructs

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>member.of(s, c)</td>
<td>State component s is a member (instance) of class c.</td>
</tr>
<tr>
<td>isa(c1, c2)</td>
<td>All members of c1 are members of c2.</td>
</tr>
<tr>
<td>described.by(c, p)</td>
<td>All members of c have property p.</td>
</tr>
<tr>
<td>has.type(p, t)</td>
<td>The value of p is in the extension of t.</td>
</tr>
<tr>
<td>instance.of(s, t)</td>
<td>The state of s is in the extension of t.</td>
</tr>
<tr>
<td>hast(t, td)</td>
<td>Type t is defined through type definition td.</td>
</tr>
<tr>
<td>involves(td, t)</td>
<td>Definition td involves t. t may have a role in the definition.</td>
</tr>
<tr>
<td>has_sentence(sl, c)</td>
<td>State law sl requires c to hold in every state.</td>
</tr>
<tr>
<td>refers(c, s)</td>
<td>Condition c involves reference to s.</td>
</tr>
</tbody>
</table>

Table 7.1: Description of relationships between static constructs

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>precondition(d, c)</td>
<td>Condition c must necessarily hold for l to apply.</td>
</tr>
<tr>
<td>postcondition(d, c)</td>
<td>Condition c must hold in the state l terminates.</td>
</tr>
<tr>
<td>has(d, sr)</td>
<td>The complex dynamic law d has sublaws in relationship sr.</td>
</tr>
<tr>
<td>involves(sr, d)</td>
<td>Sublaw relationship sr involves sublaw d.</td>
</tr>
<tr>
<td>has(d, o)</td>
<td>Dynamic law d is simple, and o is executed as part of d.</td>
</tr>
<tr>
<td>input(d, s)</td>
<td>State component s is an input parameter to d.</td>
</tr>
<tr>
<td>output(d, s)</td>
<td>State component s is an output parameter from d.</td>
</tr>
<tr>
<td>precede(o1, o2)</td>
<td>o1 is executed before o2 as part of the same dynamic law.</td>
</tr>
<tr>
<td>inserts(o, s) etc.</td>
<td>Operation o inserts a state component s into a class.</td>
</tr>
<tr>
<td>precondition(ie, c)</td>
<td>Condition c holds in the state when ie happens.</td>
</tr>
<tr>
<td>postcondition(ie, c)</td>
<td>Condition c holds in the state when ie has happened.</td>
</tr>
<tr>
<td>associated_with(ie, d)</td>
<td>Event ie is caused by the execution of d.</td>
</tr>
<tr>
<td>trigger(ie, d)</td>
<td>The event ie may trigger the execution of d.</td>
</tr>
<tr>
<td>precede(ie1, ie2)</td>
<td>Event ie1 must happen before ie2.</td>
</tr>
<tr>
<td>specified_by(ee, o)</td>
<td>Operation o is performed when ee is reported.</td>
</tr>
<tr>
<td>precede(ee1, ee2)</td>
<td>Event ee1 must happen before ee2.</td>
</tr>
</tbody>
</table>

Table 7.2: Description of relationships between dynamic constructs

**Sublaw relationship** A sublaw relationship links dynamic laws in hierarchical law structures, i.e. it specifies control structures between dynamic laws. It has a name, and it must have a specification which determines its meaning. Typical control structures found in executable CML's were described in Chapter 4.

**Operation** An operation is a specification of a state change or a query. It may insert, delete, or update state components.

**External event** An external event is a state change imposed by the environment of the system. The state change is related to two subsequent system states.

**Internal event** An internal event is a state change imposed by a dynamic law in the system.
Condition  Conditions were described above.

The relationships between the constructs are explained in Table 7.2.

An Outline of ECML

All the constructs and relationships in the unified metamodel will be supported by constructs in ECML, either explicitly, or implicitly through straightforward combinations of other constructs.

For modeling of static system aspects, classes and variables may be declared. The properties of class members, as well as variables, may be of arbitrary complex types. Types have definitions which involve the use of mathematical structures like sets, aggregates etc. These are supported through predefined functions, predicates, and access mechanisms. Conditions are expressed in a typed first order language, where all bound variables range over classes or other finite associations of data. The language is similar to relational calculus. Terms of atomic predicates may be functional expressions which are evaluated applicatively.

State transitions are caused by execution of dynamic laws and external events. Dynamic laws are organized in law hierarchies, and the control structures in these hierarchies and their meaning can be specified in ECML. The operations of simple laws are executed atomically when the law is executed. Operations are specified using expressions similar to conditions, but these include free variables which are instantiated before the operation is performed. External events are also associated with operations which are performed when the events are reported. Internal events are not part of ECML, since they can be supported indirectly through a combination of dynamic laws and state components. For instance, the internal event stock_below_reorder_level is always associated with dynamic laws which decrease the number available in the store. The fact that the event has happened can either be detected by the same dynamic law, or by another dynamic law which continuously checks the number available, and updates a variable which flags that the event has occurred.

7.3 Constructs for Modeling of State Spaces

In this and the next section we will describe the primitives of ECML. Models in ECML have an internal representation influenced by the implementation language of its prototype interpreter, PROLOG. When convenient, we will take the liberty to depart from this representation in our presentation of ECML primitives. Of course, this is only for notational convenience, and does not at all influence on the semantics of the primitives. The complete syntax of ECML is given in the appendix.

We will describe the language using the same banking domain as in Chapter 2. We will mostly model the banking system by using mappings from the PPP models given there,
however we will sometimes extend the models to illustrate the expressiveness of ECML.

State Component Classes and Variables

Classes

From the bank domain description, we find that customers, accounts, loans, payments, and transactions are most naturally modeled as state component classes. Their different properties are represented by an aggregated structure of subcomponents, the type of each subcomponent corresponding to the type of the property, and the role name of this subtype corresponding to the property name. The general form of subcomponents in an aggregated structure of a class is

```
class(Classname,role(Rolename,Typename))
```

Typename may be the typename of any type, basic (integer, real, string, boolean), or constructed. The fact that the class customer has a property name can be represented as

```
class(customer,role(name,string))
```

The association of persons which can act as users of an institution account is represented by means of a constructed type as follows:

```
class(institution_customer,role(users,users))
type(users,set(string))
```

The general form of a type is the same as that of a class, however classes are always aggregated structures of simple or complex properties. An example of a complex property is the address of a customer. It can be represented by an aggregation of street address, zip code etc.:

```
class(customer,role(address,address))
type(address,role(street,string))
```

Other useful constructors at our disposal are mathematical structures like e.g. lists, bags, enumeration etc. They will not be described in detail here, but they can easily be accommodated within ECML. Note that the type constructors can be used freely to construct arbitrary complex types.

Class instances may be related to other instances in that property values may refer to other instances. Hence, an owner property of class account can be represented exactly as any other property, i.e. as
class(account,role(owner,customer))

The inverse property of a customer, has_account, is best represented as a set of such accounts, i.e. by

\[
\text{class(customer,role(has_account,accounts))} \\
\text{type(accounts,\text{set(account)})}
\]

The choice of a set type is made because of the cardinality of the relationship - a customer may have many accounts. Note that the coverage of a relationship is represented as a state law in ECML. Likewise with identifier properties. This will be illustrated below.

Generalization relationships among classes can be represented directly in ECML. The fact that a person_customer is a customer, is expressed as

\[
\text{isa(person_customer,customer)}
\]

Properties are inherited, so it makes sense to talk about the name of a person_customer, if this property is part of the definition of a customer. Additional assertions about the relationships between different subclasses of the same superclass can be expressed as state laws, as described below.

For another example, consider the representation of a data flow in ECML. This can be represented by a class as well, with the different types linked to the flow being represented as types of properties of the class. Because of the operational semantics of PrM, it is necessary to state the additional constraint that such a class may only have one instance at any time.

As a final example, consider the document class of monthly statements which are sent to customers. The information to be included in such a document results in a complex state component, as depicted in Figure 7.2.

All the information to be included in such a document can be computed from information about customers, their accounts, and the transactions of each account. This will be shown below.

**Variables**

Whereas class instances are identified through surrogates, variables are named state components. All variables must have defined types. The general form of a variable declaration is

\[
\text{variable(Typename, Variablename)}
\]
The type of a variable is defined using the same primitives as discussed for classes. As an example, consider a variable loanapplication which includes a customer name as a component. The type of the variable must be defined, e.g.

```plaintext
type(loanapplicationtype, role(name, string))
```

and then the variable can be declared to be of that type.

Another example concerns again the representation of data flows. Since the flow can only have one instance at a time, it would probably be more efficient to represent the flow as an aggregated variable, as loan_application above. A subcomponent to indicate whether the flow contents have been read or not would then have to be added. If we were to change the semantics of PrM, so that multiple instances of a flow could coexist, a flow variable in a list form would be appropriate. The list would act as an unbounded buffer which could keep the temporal order of instances as they were produced.

### Functional Expressions, Conditions, and State Laws

Each basic type and constructed type is supported by a set of basic functions to be used for evaluation of expressions on state components. A complete list of these functions is given in the appendix. Examples of such functions are the familiar arithmetic functions on numbers, and operations on sets such as union and intersection. Using the basic set of functions, complex functional expressions can be formed by having functional expressions as arguments of other functions. Hence, functional expressions are defined and evaluated applicatively. To access the values of subcomponents in aggregated structures, the dot-notation with role names is used. A simple example of a functional expression taken from an update of the balance of an account x is

```plaintext
add(x.balance, 100)
```
which evaluates to the sum of x's balance and 100. Another example is an expression
used to compute the maximum loan of a customer which is \(2 \times x.\text{salary} - \text{sumloan}\),
expressed in ECML as sub(mul(2,x.salary),sumloan). Here x is a person customer, and
sumloan is a variable with value equal to the sum of loans this customer has.

Conditions are expressed in a subset of first order logic, similar to the relational calculus
(Date [31]). It is a subset in the sense that

- quantified variables range over classes or finite associations of data,
- functions used as terms are functional expressions of the kind described above, i.e.
a set of predefined functions can be used,
- predicates used must also belong to a predefined set of predicates defined for the
various structures supported, and
- additional functions and predicates written in other languages may be used.

State laws are partly specified by a condition. An example of a state law is a constraint
which asserts that all salaries must be positive:

\[ \forall x : \text{person\_customer} \gt(x.\text{salary},0) \]

Here, we see an example of an atomic predicate, 'greater than', which is predefined. As
was the case with functions, a battery of predicates exist for the various mathematical
structures. Familiar arithmetic predicates are 'equality', 'greater than', 'less than' etc.,
and for sets we have 'equality', 'subset' etc. A complete list of predefined predicates is
found in the appendix.

Another example of a state law is the identifier property of a class instance. It is required
that the value of this property is unique. For the custno property of the class customer, this
is expressed as:

\[ \forall x : \text{customer} \forall y : \text{customer} \\
\text{equivalence}(\text{not}(\text{equal}(x,y)), \text{not}(\text{equal}(x.\text{custno}, y.\text{custno}))) \]

Now consider the disjointness of the two classes institution\_customer and person\_customer.
This is a state law, and the condition is expressed as:

\[ \forall x : \text{institution\_customer} \text{not}(\exists y : \text{person\_customer} \text{equal}(x,y)) \\
\forall x : \text{person\_customer} \text{not}(\exists y : \text{institution\_customer} \text{equal}(x,y)) \]

A state law includes more than a logical sentence. It should have an identifying name, a
specification of when it is to be evaluated, and a specification of what is to be done if it is
violated. Its specification is hence
7.4 Constructs for Modeling of State Transitions

\[
\text{stalelaw}(\text{Name, Operation, Statecomponent, Sentence, Action})
\]

Operation is one of insert, delete, and update. Statecomponent is a path specifying which state components are involved. The salary constraint can be specified as

\[
\text{stalelaw}(\text{salary, update, person\_customer\_salary, \forall x: \text{person\_customer}}
\text{gt}(x, \text{salary}, 0), \text{report})
\]

The state law is evaluated upon updates of a customer's salary, and any violations should be reported. Another possible action is to perform a rollback, or to call a special routine to be performed. Note that specification of the evaluation situation makes state law evaluation far more efficient compared to checking all state laws in every state of execution. Also note that state laws can be specified on variables as well. Thus it makes sense to state that e.g. the variable sumloan must be greater than or equal to zero.

As a last example of a state law, consider the referential constraint that all accounts of a customer must exist. This can be formulated as

\[
\forall x: \text{customer} \forall y \in x.\text{has\_accounts} \exists z: \text{account equal}(z, y)
\]

Here, the variable y range over a finite set of values, i.e. those references found in the set-valued property has\_accounts.

7.4 Constructs for Modeling of State Transitions

These primitives will be presented in two parts, one for operations of dynamic laws and external events, and one for law structures. Operations include insertions and reads, deletions, updates, and queries. Law structures for models of the banking system will be used to present dynamic laws and different sublaw relationships. The specification of the semantics of these relationships will be presented in connection with the operational semantics of ECML, in the next section.

Queries and Data Constructors

Queries

The ability to pose queries to an information system is crucial. The development of relational database technology and SQL has made it possible to specify information retrieval purely declaratively. We favor a declarative style of queries, and in ECML they are specified using an extension of conditions as described above.
The extensions to conditions involve the use of free variables in formulas. These can be used in combination with bound variables, however, they must then be declared first in the formula. In general, a query may be in one of three forms:

\[
\begin{align*}
\text{query} & (e(\text{Functional expression})) \\
\text{query} & (c(\text{Condition})) \\
\text{query} & (\text{List of expressions,Formula})
\end{align*}
\]

The first query evaluates a functional expression, the second evaluates a condition, and the last form of query retrieves values of expressions involving state components. The result of a query should be displayed on the screen. For instance, to retrieve the name and address of all customers, the query

\[
\text{query}([x\.name,x\.address],x\.customer)
\]

can be issued. To list the same information about customers who only have accounts with balances greater than K, the following query could be used:

\[
\text{query}([x\.name,x\.address],x\.customer \ \forall y \in x\.has\.accounts \ gtv(y\.balance,K))
\]

Any functional expression may be evaluated on the retrieved information before it is presented. For instance, to compute the amount of interest paid for all loans the first year for a person with name 'arne', the query

\[
\text{query}([\text{mul}(x\.amount,x\.interest)],x\.loan \ equal(x\.owner\.name,'arne'))
\]

can be issued. In this query we also see a more advanced use of the dot notation. As a relationship is just another property, the dot-notation can be used to easily follow links of relationships in the state components. Owner is a property of the class loan, and can take identifiers of instances of class customer as values.

**Data Constructors**

Data constructors are expressions which are used to compute temporary data structures used as terms of functions and predicates. Examples include set constructors and bag constructors. These will be described in the following.

A set constructor specifies the simple or complex elements of a set through a formula of the same kind as used in a queries. In fact, it can be thought of as a query which returns its information in a set. The general form of a set constructor is
Hence, the set constructor

\{x\text{.salary}|x:\text{person}\_\text{customer}\}\}

is evaluated to the set of all the distinct salaries of person customers. Note that since it is a set, duplicate elements are removed during its construction. If this effect is not desired, a bag constructor is used instead. It is basically specified in exactly the same way.

Using these constructors, more complex state laws and queries may be formulated. For instance, the assertion that the number of customers with negative balance should not exceed 100, can be expressed with the sentence

\text{lt}(\text{count}(\{x|x:\text{customer}\text{.lt}(x\text{.balance},0)\}),100)

Another example is the expression to compute the sum of loans of a particular customer 'john' (assuming it is the initial loan amount):

\text{query}(\text{e}(\text{sum}(<y\text{.amount}|x:\text{customer}\text{.and}(\text{equal}(x\text{.name}',\text{\textquoteleft}john\textquoteright),y\in x\text{.takes}\_\text{up} > )>)))

Note that a bag constructor is used to allow duplicate loan amounts to be included before the aggregate function \text{sum} is evaluated.

The final query example which includes data constructors is 'list names and sum of loan balance of all customers who have salaries greater than k and have at least two accounts with balance greater than c'. The query can be formulated as:

\text{query}([x\text{.name},\text{sum}(<y\text{.balance}|y\in x\text{.takes}\_\text{up}>)],
\text{\quad x:person}\_\text{customer}\text{.and}(\text{gt}(x\text{.salary},k),
\text{\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{gt}(\text{count}(<z|z\in x\text{.has}\_\text{accounts}\text{.gt}(z\text{.balance},c)>),1))))

\section*{Insertions, Deletions, and Updates of State Components}

\subsection*{Insertions}

Insertions of instances of state component classes can basically be performed in one of two ways: 1) by reading all property values through interaction with the environment, and 2) by computing all property values from the values of other state components. Also, the
insertion of subclass instances is treated specially, since they need to be related to their superclass.

The simplest form of an insertion is by reading property values of the new instance from the user. This operation is simply (assuming no generalization relationship is involved):

\[ \text{insert}(\text{Classname}) \]

Property values which refer to other instances are specified through a formula with free variables which are instantiated to the reference values needed. Such a formula can be given by the user as well.

When an instance of a subclass is to be inserted, then this instance must be specified as a specialization of an existing instance belonging to the superclass. The general form of this kind of insertion is

\[ \text{insert}(\text{subclass}(\text{Subclassname}), \text{Formula}, \text{Instance}) \]

As an example, consider the insertion of an instance of the class person_customer, which is a customer with custno 128:

\[ \text{insert}(\text{subclass}(\text{person_customer}), x: \text{customer} = \text{equal}(x.\text{custno}, 128), x) \]

All properties specific to person customers will then be read.

Sometimes it is possible to compute property values from values of existing state components. For instance, assume that a class bad_customers exists, with two properties: custno and balances. The latter property is a set-valued property with the balances of all accounts a bad customer has. Instances of this class can be specified by the operation

\[ \text{insert}(\text{bad_customers}, x, y: \text{customer} = \text{equal}(y.\text{classification}, \text{bad}), \left[ (x.\text{custno}, y.\text{custno}), (x.\text{balances}, z.\text{balance} \mid z \in y.\text{has_accounts}) \right] ) \]

Here, \( x \) is the variable to stand for the new instance inserted. The last argument is a set of assignment pairs, where the left arguments are state components, and the right arguments are functional expressions which are evaluated to the new property values of the state components.

The same principle is used if the new instance is a member of some subclass, or if only named variables need to be accessed in order to compute property values.
Deletions

Deletions of class instances are also specified using formulas with free variables. Instantiations which make the formula true, may be candidates for deletion. The general form of a deletion operation is:

\[
delete(\text{Formula}, \text{List of free variables})
\]

When an instance is deleted, all references to this instance are deleted as well. Furthermore, if a superclass instance is deleted, the instance is additionally deleted from all subclasses it is an instance of. Consider the example where a customer with customer number 128 is to be deleted:

\[
delete(x:\text{customer} \ equal(x.\text{custno}, 128), [x])
\]

Any property of other instances referring to this instance are nullified, and the instance is also removed from either person\_customer or institution\_customer, assuming that these constitute a partition of the customer class.

Updates

The general form of an update is as follows:

\[
update(\text{Formula}, \text{Set of assignments})
\]

As was briefly noted above, an assignment again consists of two parts:

\[
(\text{State component}, \text{Functional Expression})
\]

The former argument is either a name of a variable, or a specification of a subcomponent through the use of the dot-notation. The functional expression is of the same kind as presented previously. Since properties are inherited along specialization relationships, it makes sense to update an inherited property value.

Let us turn to some examples. Updating the balance of an account 1204 in accordance with a withdrawal transaction 103 can be specified as e.g.

\[
update(x:\text{transaction} \ y:\text{account}
    \text{and}(equal(x.\text{transaction\_id}, 103), equal(y.\text{account\_id}, 1204)),
    \text{[(y.balance, sub(y.balance, x.amount))]])
\]

To link the foregoing transaction (103) into the set of transactions of account 1204, the
update

\[\text{update}(x: \text{transaction} \ y: \text{account})\]
\[\text{and}(\text{equal}(x.\text{transaction}_id, 103), \text{equal}(y.\text{account}_id, 1204)),\]
\[\text{[(y.\text{transactions}, \text{union}(y.\text{transactions}, \{x\}))]]}\]

can be specified.

It should be noted that also non-autonomous data like numbers in a set, can be updated. If salaries is a variable of type set of integer, then the update

\[\text{update}(x \in \text{salaries}, [(x, \text{add}(x, k))])\]

can be specified to add \( k \) to each element of the set.

To round off this presentation of operations, we will show possible computations of some of the parts of the monthly statement to customers. Referring back to the class definition of monthly\_statement, the totalbalance property can be computed by the expression

\[\text{sum}(x.\text{balance}|x \in y.\text{has.accounts})\]

assuming that the variable \( y \) is bound to a particular customer.

The total insertion of instances of this class is fairly complex, and can be specified as follows:

\[\text{insert}(\text{monthly}\_\text{statement}, x, c: \text{customer},\]
\[\text{[(x.\text{name}, c.\text{name}), (x.\text{address}, c.\text{address}),}\]
\[\text{(x.\text{totalbalance}, \text{sum}(v.\text{balance}|v \in c.\text{has.accounts})})}\]
\[\text{(x.\text{accountsinfo}, \langle[\text{accountinfo}(\text{(account}_id, y.\text{account}_id}\]
\[\text{(interest.\text{earned}, \text{some formula})},\]
\[\text{(start.\text{balance}, \text{some formula})},\]
\[\text{(end.\text{balance}, y.\text{balance})}\]
\[\text{(transactionsinfo}, \text{[(transactioninfo}(\text{(date, z.\text{date})}\]
\[\text{(type, z.\text{type})},\]
\[\text{(amount, z.\text{amount}))}]\]
\[\text{z \in y.\text{transactions})})}\]
\[\text{y \in c.\text{has.accounts})})\}\]

Note that the type name of aggregates are given, e.g. accountinfo, and that the subcomponents are found in the list after the name of the aggregate.
Dynamic Laws and Law Structures

We have previously stated the general properties of dynamic laws. In this section we will present the concrete syntax of dynamic law primitives, and illustrate them through examples from the banking system. Sublaw relationships used are those needed to represent the dynamics of PPP models. Details of sublaw relationship specification are given in the next section.

Dynamic Laws

The dynamics of information system models is represented through an hierarchical law structure, in which a law is expressed as

\[
\text{dynamic_law}(\text{Id,Precondition,Body,Postcondition,Input,Output,Properties})
\]

The various parts are taken directly from the unified metamodel. Id is a unique name or other identifier of a law. Precondition is a condition, a logical sentence of exactly the same kind as those used in state laws. Body is either a sublaw relationship between a law and its sublaws, i.e.

\[
\text{sublaws(Sublaw relationship,List of sublaws with parameters})
\]

or a list of operations in the form

\[
\text{operations(List of operations})
\]

In the first case, the law is a complex law, whereas in the latter it is regarded as a simple law. Complex laws are found as interior nodes of the law structure, and simple laws as leaves. Sublaw relationship is a name of a relationship for which the semantics has been defined. The order of laws in the list of sublaws may be important for some sublaw relationships. The list of operations of a simple law are executed atomically from one state to the next, in the sequence given. Note that it is assumed that inconsistent operations across dynamic laws do not occur.

Postcondition is a formula of the same kind as the precondition. It is used when it is necessary to check a condition upon the termination of a dynamic law. Inputs is a list of parameter name and parameter type pairs. They are used to pass values to dynamic laws as they apply. Outputs is a similar list of output parameters, i.e. placeholders for values to be passed on as laws terminate. Properties is a list of property-value pairs which are needed in addition to those described. It may for instance be desirable to keep priority information with laws, and base the selection of laws to apply on these priorities.
Examples of Complex Laws

Complex laws are found in the PPP banking model as representations of PrM processes at various decomposition levels, and as representations of blocks of simpler PLD constructs. Figure 7.3 shows the law structure corresponding to the dynamic laws found in the banking model. The system law is here an artificial law which has all processes on the top level as sublaws. We recognize the decomposition of process P1, as well as the block structured PLD model for process P1.2. We have included the skeleton of the PLD model in order to show the correspondence with the law hierarchy. Note the introduction of law pld12, which is the null operation to be performed if the first alternative construct does not apply. Different sublaw relationships have been depicted in the figure. We will now describe each of the relationships needed for representation of PPP models.

**VCS** If \( l \) has a vague control structure of sublaws, \( l : \text{VCS}(l_1, \ldots, l_n) \), then \( l \) is executed by executing sublaws in a 'spontaneous' manner when their preconditions hold. When a sublaw terminates, it may be applied again in the same state. This relationship is used to represent decompositions of processes, and allows processes to be executed concurrently.

**SEQ** If \( l \) has a sequence of sublaws, \( l : \text{SEQ}(l_1, \ldots, l_n) \), then the sublaws are executed in the sequence given. Only one sublaw may execute at a time. If the precondition of a sublaw \( l_i \) does not hold, then \( l_i \) will be tried for execution in each subsequent state until it applies. \( l \) terminates when \( l_n \) terminates. The \( \text{SEQ} \) relationship is used in PLD models to represent sequences of constructs.

**EXCSEQ** If \( l \) has an exclusive-sequential structure of sublaws, \( l : \text{EXCSEQ}(l_1, \ldots, l_n) \), then \( l \) is executed by executing exactly one sublaw. If \( l \) applies, then at least one of the sublaws must apply in the same state. If the precondition of more than one law holds, the first law by sequential order is selected for application, hence the name \( \text{EXCSEQ} \). In PLD's, this relationship is found in choice constructs, each alternative being represented as a sublaw.
7.4. Constructs for Modeling of State Transitions

REPSEQ If \( l \) has a repeating-sequential structure of sublaws, \( l: \text{REPSEQ}(l_1, \ldots, l_n) \), then \( l \) is executed by executing the sublaws as in a SEQ relationship. However, as \( l_n \) terminates, and the precondition of \( l \) holds, the sequence is executed over again. The relationship is used in the representation of loop constructs in PLD models.

The use of these sublaw relationships was indicated in Figure 7.3. Note that the combination of VCS and SEQ makes it possible to represent communicating concurrent processes. The precondition of a law representing a process is 'a legal combination of triggering input flows has been received'. During execution of the process, it may be necessary to receive more flows. The sequence relationship has the effect of letting the process wait for the flows until they are available, and then continue on its execution.

We now give a few examples of complex laws. For process P1.2:Verify_amount, we get the following dynamic law:

\[
\text{dynamic\_law(P1.2,}\exists x: \text{withdrawal\_transaction,} \\
\text{sublaws(SEQ,}[\text{true},[\text{[true]}],[],[]])}
\]

given that the flow withdrawal\_transaction is represented as a class. This flow is the triggering input flow of the process. The second choice construct in the PLD for P1.2 is

\[
\text{dynamic\_law(pld7,}\text{true,sublaws(EXCSEQ,}[\text{true},[\text{[true]}],[],[]])}
\]

whereas the first alternative of this choice is represented by

\[
\text{dynamic\_law(pld8,}\text{or(equal(amount,0),gt(amount,account\_balance)),} \\
\text{sublaws(SEQ,}[\text{true},[\text{[true]}],[],[]])}
\]

Examples of Simple Laws

As a first example of a simple dynamic law, consider the update of the state component amount from the PLD for process P1.2 in the banking model. Assignments in PLD's are generally represented as simple laws. This assignment is unconditionally executed after the receipt of the flow New\_withdrawal. The law can be specified as

\[
\text{dynamic\_law(pld6,}\text{true,} \\
\text{operations([update(true,[[(\text{amount,new\_amount})]])],true,[[true],[],[]])}
\]

We here assume that the variables are unique within the model. If necessary, the locality of variables can be taken care of by defining an aggregated structure of components for each PLD.
Another example from the PLD model, is the sending of data flows. Let us assume that the flow Withdrawal_rejection is represented as a variable in ECML. This variable is of an aggregated type, the aggregate consisting of the three subcomponents message, account_balance, and data_read. The last component is a boolean component which indicates whether the flow contents have been read or not. It is used to avoid flow contents being overwritten by the producing process. The send construct which sends the flow can then be represented as

\[
\text{dynamic law(\text{pld5,equal(withdrawal_rejection.data_read,\text{true}), operations[[update(\text{true,}}}}
\]
\[
\left[\text{(withdrawal_rejection.message,'Error - Available amount is:')}\right]
\]
\[
\text{(withdrawal_rejection.account_balance,account_balance)}
\]
\[
\text{(withdrawal_rejection.data_read,\text{false})]])], true.[].[].[])
\]

If a flow is represented as a class, the precondition is a check that no instances of the flow exist. The operation is then to insert a new instance with properties error_message and account_balance as above. Receipts of data flows are simple laws represented in a similar manner.

**External Events**

External events is the means for the environment to report occurring events and affect the state of the executing models. An external event is therefore associated with a list of state changing operations. Often these will be inputs of data from the keyboard or other input device, but they may also be direct updates of state components as result of choosing the event to be reported. In addition they have a name which identifies them uniquely, and can be used to retrieve information about the source of the event. The general form of an external event is hence

\[
\text{external_event(Identifier,List of operations)}
\]

Depending on the execution strategy, external events may be reported in stable states only, or in both unstable and stable states. In the latter case, it is assumed that dynamic laws may execute for long times, during which new events should be reportable and initiate concurrent executions or interrupts of the ongoing execution. The semantics of dynamic laws as presented in the next section is not depending on this choice. Rather it is a choice of the interpreter of ECML, and both alternatives can easily be accommodated if necessary.

As a simple example of an external event, consider new transactions which can be given to the bank system. In the PPP model, this event corresponds to the triggering input flow Transaction from the external agent Customer. The operation associated with this event, is an input from the keyboard. Given that the input flow is represented by a class, the external event is given by:
externalevent(transaction,operations[[insert(transaction)]]))

7.5 The Operational Semantics of ECML

We will now present the operational semantics of ECML. The overall execution semantics which concerns application and termination of dynamic laws can be specified without referring to the details of operations etc. Most central to the operational semantics are the constructs dynamic law, precondition and sublaw relationship. We will not give a complete semantics for all the primitives, since we assume the semantics of predefined functions and predicates to be well-known. See the appendix for a complete list of the types, functions, and predicates taken into ECML at this stage.

First we describe the execution cycle for execution of ECML models.

The Execution Cycle

At a superficial level, we can describe the execution of an ECML model as follows, assuming that external events only can be reported in stable states:

1. Initially the system is in a stable state, i.e. a state in which it will remain unless forced to another state by an external event.

2. An external event occurs, either bringing the system directly into a new state, or to an unstable state, in which a dynamic law applies. This event is in some way reported to the system.

3. In the case of an unstable state, the system responds by applying dynamic laws in a top-down fashion down the law structure of the model, until the level of simple laws. Operations of simple laws are performed atomically, enforcing state transitions to take place. A series of state transitions and law applications take place until a new stable state is reached.

4. If any postcondition or state law is violated, appropriate action should be taken. At least the event should be reported, but it may also be necessary to halt the execution and reinstall the previous stable state, i.e. perform a rollback.

5. When no dynamic laws longer apply, the system is again in a stable state, ready to accept new external events.

This general execution cycle will be refined in the following. We will especially focus on the application and termination of dynamic laws.
Law Application and Termination

Parts of the motivation for organizing laws into hierarchical structures is to facilitate a more efficient computation. The idea is that the search for applicable laws under \( L \) above can be done in a more focused way than by a brute force search through a potentially large number of laws, although this would certainly lead to a much simpler operational semantics. Instead we divide the total set of dynamic laws \( L \) in a model into three disjoint sets:

i) the set of *executing* laws, \( E \),

ii) the set of *considered* laws \( C \), and

iii) the set of *not considered* laws, \( N \).

Note that we have \( L = E \cup C \cup N \), and \( E \cap C = E \cap N = C \cap N = \emptyset \).

In a stable state, \( N = L \). During execution, laws will be transferred between the three sets. For instance, a law initially in \( N \) may be transferred to \( C \), then to \( E \), and as it terminates, again to \( N \). The manner in which these transitions take place is directed by the semantics of the control structures or sublaw relationships which hold in the law structure. All transitions between the three sets are possible from one state to another as depicted in Figure 7.4. Basically, the search for applicable laws is initiated from those considered (i.e. members of \( C \)), and proceed downwards the law structure from them. The idea is to prune off large portions of the law structure, and hence narrow the search space for applicable laws.

In the following, we will first describe the transitions informally, with focus on law application and termination. Then we will formalize the transitions to make the operational semantics precise.

In any state, there are three ways in which a law may apply:

1. It applies if it
   - is a member of \( C \),
   - its precondition holds, and
   - it is *selected* to be applied from among the other sublaws of its superlaw in the law structure.

   This corresponds to transition 9 in Figure 7.4.

2. It applies if its superlaw in the law structure applies in the same state and
   - it is a *candidate* among the sublaws of its superlaw,
   - its precondition holds in the current state, and
Figure 7.4: Transitions between the sets $\mathcal{E}$, $\mathcal{C}$, and $\mathcal{N}$ during execution of an ECML model

- it is selected to be applied from among the sublaws of its superlaw which satisfy the two previous conditions.

This corresponds to transition 5.

3. It applies (reapplies) if
   - it has been executing, but terminates in the current state,
   - it is still considered,
   - its precondition holds, and
   - it is selected in the same fashion as above.

This corresponds (in part) to transition 1.

The choice of candidates and the subsequent selection of laws to be applied can be formulated as functions, as will be described below.

Likewise, there are also three different transitions for law termination. Associated with each sublaw relationship is a \textit{terminatecondition} which when instantiated with a particular law can be evaluated to determine if the law should be terminated. Any law satisfying this condition and currently a member of $\mathcal{E}$, will be transferred to either $\mathcal{E}$, $\mathcal{C}$, or $\mathcal{N}$ in the following manner:

1. It reenters $\mathcal{E}$ if it applies as under 3. above.

2. It terminates and enters $\mathcal{N}$ if
   - either its superlaw terminates and does not reapply, or
   - it is not considered any more.

This corresponds to transition 2.

3. It terminates and enters $\mathcal{C}$ if
its terminate condition holds,
its superlaw is still executing,
it is still considered, and
it is not reapplied.

This corresponds to transition 3.

Note that application and termination only cover five of totally nine transitions. Transitions 4 and 7 represent frame axioms, laws remain in these sets from one state to another unless transitions to other sets are made possible. The last two transitions are between the C and N sets, and correspond to laws being considered when their superlaws apply, and then being not considered as their superlaws stop execution. These transitions will also be formalized below.

**Formalization of Law Transitions**

In the following, transitions 1-9 will be formalized with the use of logic. The formalization is the basis for the development of an interpreter of ECML.

First we define some functions and predicates to be used:

- $\text{cand}_{sr}(l)$ is a function which for a law $l$ in a sublaw relationship $sr$ to its sublaws, returns the set of sublaws which are candidates for application as $l$ is applied.

- $\text{select}_{sr}(\{l_1, \ldots, l_n\})$ is a function which from a set of sublaws which are in a sublaw relationship $sr$ to their superlaw, returns a subset of the law set which are selected for application.

- $\text{addC}_{sr}(l)$ is a function which for a law $l$ in a sublaw relationship $sr$ to its sublaws, returns the sublaws which can be added to $C$ as $l$ is applied.

- $\text{taddC}_{sr}(l)$ is a function which for a law $l$ in a sublaw relationship $sr$ to its superlaw, returns a set of laws which can be added to $C$ as $l$ terminates.

- $\text{termc}_{sr}(l)$ is a function which for a law $l$ in a sublaw relationship $sr$ to its sublaws, returns the terminate condition of $l$.

- $\text{prec}(l)$ is a function which for a law $l$ returns the logical sentence which is its precondition.

- $\text{sup}(l)$ is a function which for a law $l$ returns its superlaw.

- $\text{holds}(c)$ is a meta-predicate which is true if the logical sentence $c$ is true.

- $sr(l)$ is a function which for a complex law $l$ returns an index which corresponds to the sublaw relationship between $l$ and its sublaws. This index corresponds to the index $sr$ used in the functions above.
The transitions we consider all involve two subsequent states, hence we use the temporal operator $\bullet$ to refer to the previous state.

Now we are in position to describe the transition rules formally. For each rule, we also give a short textual description.

**Law transition rule 1** A dynamic law remains in $E$ if its terminate condition does not hold, or it holds, but the law reapply in the same state:

\[
((\bullet(l \in E) \land \neg \text{holds}(\text{termc}_{sr(i)}(l))) \lor \\
(\bullet(l \in E) \land \text{holds}(\text{termc}_{sr(i)}(l)) \land l \in \text{taddC}_{sr(\text{sup}(l))}(l) \land \text{sup}(l) \in E \land \\
l \in \text{select}_{sr(\text{sup}(l))}(\text{\{}l'|\text{sup}(l') = \text{sup}(l) \land (\bullet(l' \in C) \lor (\bullet(l' \in E) \land \\
\text{holds}(\text{termc}_{sr(i)}(l'))) \land l' \in \text{taddC}_{sr(\text{sup}(l'))}(l'))\} ) \land \text{holds}(\text{prec}(l')))))) \\
\Rightarrow l \in E
\]

We recognize the parts of the formula from the informal description of law application above. Note that the selection is performed on the set of sublaws which are either members of $C$ or terminate in the current state and are still considered.

**Law transition rule 2** A dynamic law is transferred form $E$ to $N$ if its terminate condition holds, and either its superlaw is no longer executing or it is not considered for application any longer:

\[
(\bullet(l \in E) \land \text{holds}(\text{termc}_{sr(i)}(l))) \land (\text{sup}(l) \notin E \lor l \notin \text{taddC}_{sr(\text{sup}(l))}(l)) \\
\Rightarrow l \in N
\]

**Law transition rule 3** A dynamic law is transferred from $E$ to $C$ if its terminate condition holds, it is still to be considered for application, and it is not applied again in this state:

\[
(\bullet(l \in E) \land \text{holds}(\text{termc}_{sr(i)}(l))) \land l \in \text{taddC}_{sr(\text{sup}(l))}(l) \land \text{sup}(l) \in E \land \\
l \notin \text{select}_{sr(\text{sup}(l))}(\text{\{}l'|\text{sup}(l') = \text{sup}(l) \land (l' \in C \lor (\bullet(l' \in E) \land \\
\text{holds}(\text{termc}_{sr(i)}(l'))) \land l' \in \text{taddC}_{sr(\text{sup}(l'))}(l'))\} ) \land \text{holds}(\text{prec}(l')))))) \\
\Rightarrow l \in C
\]

For transition 1, we could have used a frame axiom, stating that a law will remain in $E$ unless it is transferred by means of formulas for transitions 2 or 3. Instead we chose to make the reapplication of dynamic laws explicit. However, for transitions 4 and 7, we formulate frame axioms. In the formulas, $A_{ij}$ stands for the antecedent of transition rule $ti$.

**Law transition rule 4** Frame axiom: A dynamic law remains in $N$, unless it is transferred by means of transition rules 5 or 6:
\( (l \in N) \land \neg (A_{18} \lor A_{19}) \Rightarrow l \in N \)

**Law transition rule 5** A dynamic law is transferred from \( N \) to \( E \) if its superlaw applies, it is a candidate, its precondition holds, and it is selected:

\[
\begin{align*}
\bullet & (l \in N \land \sup(l) \notin E) \land \sup(l) \in E \land \\
& l \in \text{select}_{sr(\sup(l))}(\{l' | l' \in \text{cand}_{sr(\sup(l))}(\sup(l)) \land \text{holds}(\text{prec}(l'))\}) \\
\Rightarrow & l \in E
\end{align*}
\]

**Law transition rule 6** A dynamic law is transferred from \( N \) to \( C \) if its superlaw applies, it is to be considered in the next state, but it does not apply in this state:

\[
\begin{align*}
\bullet & (l \in N \land \sup(l) \notin E) \land \sup(l) \in E \land l \in \text{addC}_{sr(\sup(l))}(\sup(l)) \land \\
& l \notin \text{select}_{sr(\sup(l))}(\{l' | l' \in \text{cand}_{sr(\sup(l))}(\sup(l)) \land \text{holds}(\text{prec}(l'))\}) \\
\Rightarrow & l \in C
\end{align*}
\]

**Law transition rule 7** Frame axiom: A dynamic law remains in \( C \), unless it is transferred by means of transition rules 8 or 9:

\[
\begin{align*}
\bullet & (l \in C) \land \neg (A_{18} \lor A_{19}) \Rightarrow l \in C
\end{align*}
\]

**Law transition rule 8** A dynamic law is transferred from \( C \) to \( N \) if its superlaw is no longer executing:

\[
\begin{align*}
\bullet & (l \in C \land \sup(l) \in E) \land \sup(l) \notin E \Rightarrow l \in N
\end{align*}
\]

**Law transition rule 9** A dynamic law is transferred from \( C \) to \( E \) if its precondition holds, and it is selected from those sublaws which are either members of \( C \) or terminates and are reconsidered:

\[
\begin{align*}
\bullet & (l \in C) \land \\
& l \in \text{select}_{sr(\sup(l))}(\{l' | \sup(l') = \sup(l) \land (\bullet(l' \in C) \lor (\bullet(l' \in E) \land \\
& \text{holds}(\text{term}_{sr(l)}(l')) \land l' \in \text{addC}_{sr(\sup(l'))}(l')) \land \text{holds}(\text{prec}(l'))\}) \\
\Rightarrow & l \in E
\end{align*}
\]

We can now conclude the formal treatment by reformulating the application and termination of dynamic laws with reference to the transitions above, and we state three general properties of the law transitions.
Application rule  A law applies if it was not member of $\mathcal{E}$ in the previous state, but is a member of $\mathcal{E}$ in this state, or it was a member, terminated, and then reapplied:

\[
\text{applies}(l) \iff (\bullet(l \not\in \mathcal{E}) \land l \in \mathcal{E}) \lor (\bullet(l \in \mathcal{E}) \land \text{terminates}(l) \land l \in \mathcal{E})
\]

Termination rule  A law terminates if its terminate condition holds and it is executing. A simple law always terminates after the state it was applied:

\[
\text{terminates}(l) \iff \bullet(l \in \mathcal{E}) \land (\text{simple}(l) \lor \text{holds}(\text{termc}_{sr}(l)(l)))
\]

$\mathcal{N}$ membership property  If a law is in $\mathcal{C}$ or $\mathcal{N}$, then all its sublaws are in $\mathcal{N}$.

$\mathcal{E}$ membership property  If a law is in $\mathcal{E}$, then all its superlaws in the law structure are in $\mathcal{E}$.

System law membership  If the system is in a stable state, then $l_s \in \mathcal{N}$. If the system is in an unstable state, then $l_s \in \mathcal{E}$.

We see how the semantics of sublaw relationships are captured in the functions $\text{cand}_{sr}$, $\text{select}_{sr}$, $\text{addc}_{sr}$, $\text{taddc}_{sr}$, and in the condition $\text{termc}_{sr}$. In the next section we will show how these can be specified, but we see that the operational semantics has been specified without any concern of the particularities of these relationships. Hence it should be possible to add these functions as necessary. For illustration, we now indicate how the VCS relationship found in PPP models is specified.

If we have $l : \text{VCS}(l_1, ..., l_n)$, then $\text{cand}_{\text{VCS}}(l) = \{l_1, ..., l_n\}$, since in a VCS relationship all sublaws may be applied when their superlaw applies. All sublaws for which the precondition holds, are selected. For the $\text{addc}_{\text{VCS}}$ function we have: $\text{addc}_{\text{VCS}}(l) = \{l'|l' \in \text{sublaws}(l) \land \neg \text{holds}(\text{prec}(l'))\}$, i.e. those sublaws which do not apply, are added to $\mathcal{C}$, and may be considered for application in the next state.

The terminate condition is that no sublaw is executing, i.e. member of $\mathcal{E}$, and that no sublaw member of $\mathcal{C}$ can be applied to enter $\mathcal{E}$. This can be formulated as $\text{terminates}(l) \iff \neg \exists l_i(\text{sublaw}(l, l_i) \land (\bullet(l_i \in \mathcal{E}) \land \neg \text{terminates}(l_i) \lor \text{holds}(\text{prec}(l_i))))$. Furthermore, $\text{taddc}_{\text{VCS}}(l_i) = l_i$, so sublaws may be repeatedly executed.

Note that concurrency occurs in a natural manner, i.e. all laws fulfilling the application condition are applied in a state. In general, all laws which are members of $\mathcal{E}$ at the same time, execute concurrently.

Now we have described the application and termination of dynamic laws in an ECML model. However, we have only considered the non-exceptional case when no postcondi-
tions or statelaws are violated. When such cases occur, at least two possible actions are possible: 1) reinstall the previous stable state and report the violation, and 2) just report the violation. We will not describe the rollback process in detail here. Appropriate rollback mechanisms is a topic of research on its own.

**Specification of Sublaw Relationships**

Inspired by the analysis of executable CML's in Chapter 4, we will now describe the specification of sublaw relationships in ECML. The basic idea is to specify the functions which are evaluated for law application and termination, and give a precise definition of the meaning of these in terms of the values returned by the functions. For each possible specification, we give a rationale for it, e.g. by referring to sublaw relationships of existing languages. The precise definitions given here together with the operational semantics presented above serve as a basis for developing an interpreter prototype for ECML.

**Specification of *cand*<sub>sr</sub>**

This function is used to select the sublaws of a law *l* which are to be tested for applicability as *l* is applied. For each sublaw returned, the precondition is checked to see if it holds in the current state. If this is the case, it is a member of the law set which is passed as argument to the *select*<sub>sr</sub> function. The possible specifications and their values are given in Table 7.3. The precise definition of the function is given in Table 7.4.

Note that this function is used at the time a superlaw is applied. For later states, the functions *addC* and *taddC* are used to determine candidates for execution. Finally, we require that the following constraints are satisfied:

\[
\begin{align*}
\text{cand}_{sr}(l)(l) &\neq \emptyset, \\
\text{cand}_{sr}(l)(l) &\subseteq \{l_i | \text{sublaw}(l_i, l)\}
\end{align*}
\]

**Specification of *select*<sub>sr</sub>**

The value of this function determines which sublaws are to be applied from a set of sublaws which are both candidates and have their preconditions fulfilled. Possible specifications are given and explained in Table 7.5. The precise definition of the function values is given in Table 7.6.

Specifications of the two functions *cand*<sub>sr</sub> and *select*<sub>sr</sub> can be combined at wish, however some combinations will produce the same result. This is the case for instance for the two combinations (onerrandom,first) and (onerrandom,onerrandom). We do not need to constrain the possible combinations because of this, as long as the combinations make sense. However, the specification of *select*<sub>sr</sub> must satisfy the constraints.
7.5. The Operational Semantics of ECML

<table>
<thead>
<tr>
<th>Specification</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>The first law in the sublaw sequence is a candidate. It is used for sequential sublaw relationships.</td>
</tr>
<tr>
<td>onerandom</td>
<td>A sublaw is picked at random to be a candidate. This may be used to pick random events from a set of possible events in the environment.</td>
</tr>
<tr>
<td>all</td>
<td>All sublaws are candidates. This is found e.g. if laws may execute in parallel.</td>
</tr>
<tr>
<td>condition,c</td>
<td>All sublaws which have certain properties values, are candidates. These properties may include e.g. priorities. In any case, checking the condition means checking values of law properties, and possibly comparing them with property values of other dynamic laws.</td>
</tr>
<tr>
<td>user</td>
<td>A user picks laws which are candidates, i.e. priorities are given interactively. This facility can be used to execute incomplete models.</td>
</tr>
</tbody>
</table>

Table 7.3: Explanation of $\text{cand}_{sr}$ specifications

<table>
<thead>
<tr>
<th>$\text{s(}\text{cand}_{sr}(l))$</th>
<th>$\text{cand}_{sr}(l)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>${l_1}$</td>
</tr>
<tr>
<td>onerandom</td>
<td>$\text{xor}({l_1}, \ldots, {l_n}), P(\text{cand}_{sr}(l) = {l_i}) = 1/n$</td>
</tr>
<tr>
<td>all</td>
<td>${l_1, \ldots, l_n}$</td>
</tr>
<tr>
<td>condition,c</td>
<td>${l_i \mid l_i \in {l_1, \ldots, l_n} \wedge \text{holds}(c(l_i))}$</td>
</tr>
<tr>
<td>user</td>
<td>${l_i \mid l_i \in {l_1, \ldots, l_n} \wedge \text{user gives } l_i \text{ as input}}$</td>
</tr>
</tbody>
</table>

Table 7.4: Specification of the $\text{cand}_{sr}$ function

$L \neq \emptyset \Rightarrow \text{select}_{sr}(l)(L) \neq \emptyset \land \text{select}_{sr}(l)(L) \subseteq L \land L \subseteq \{l_i \mid \text{sublaw}(l_i, l)\}$

Note that the order of the sublaws in the arguments of $\text{cand}_{sr}$ and $\text{select}_{sr}$ is kept in the returned sequence of sublaws.

**Specification of $\text{addC}_{sr}$**

This function is used the first time a law is applied, to determine which of its sublaws will enter $C$ for the next state of execution. The basis for specifying $\text{addC}_{sr}$ is the division of sublaws made by the functions $\text{cand}_{sr}$ and $\text{select}_{sr}$, which is depicted in Figure 7.5.

The specification of the function with explanations is given in Table 7.7, while the precise definition of the function values is given in Table 7.8. Note that the function is only used in connection with $\text{cand}_{sr}$ and $\text{select}_{sr}$, i.e. when the superlaw is applied. Later on, the $\text{taddC}_{sr}$ function is used to add candidates to $C$. 
<table>
<thead>
<tr>
<th>Specification</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>Only the first law in the sequence found in the argument sequence is selected. A typical example is structures like the CASE statement in programming languages.</td>
</tr>
<tr>
<td>onerandom</td>
<td>A sublaw is picked at random to be a applied. Used to resolve or simulate non-deterministic behavior, e.g. as found in Petri-nets.</td>
</tr>
<tr>
<td>all</td>
<td>All sublaws are selected. This is found e.g. if laws may execute in parallel, for instance to represent concurrent processes.</td>
</tr>
<tr>
<td>condition,s</td>
<td>A subset of the argument set is selected, based on a selection condition, similar to the candidate condition above.</td>
</tr>
<tr>
<td>user</td>
<td>A user selects a subset of the argument set to be applied. Can be used in the same manner as onerandom, but now the user determines the order in which laws are applied. This is also useful for the execution of incomplete models.</td>
</tr>
</tbody>
</table>

Table 7.5: Explanation of select_{sr} specifications

<table>
<thead>
<tr>
<th>( s(\text{select}_{sr}({l_1, \ldots, l_n})) )</th>
<th>( \text{select}_{sr}({l_1, \ldots, l_n}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>{( l_1 )}</td>
</tr>
<tr>
<td>last</td>
<td>{( l_n )}</td>
</tr>
<tr>
<td>onerandom</td>
<td>xor(( {l_1, \ldots, l_n} )), ( P(\text{select}_{sr}({l_1, \ldots, l_n}) = {l_i} = 1/n )</td>
</tr>
<tr>
<td>all</td>
<td>( {l_1, \ldots, l_n} )</td>
</tr>
<tr>
<td>selectcondition,s</td>
<td>{( l_i \mid l_i \in {l_1, \ldots, l_n} \land \text{holds}(s(l_i)) }}</td>
</tr>
<tr>
<td>user</td>
<td>{( l_i \mid l_i \in {l_1, \ldots, l_n} \land \text{user gives } l_i \text{ as input} }}</td>
</tr>
</tbody>
</table>

Table 7.6: Specification of the select_{sr} function

Figure 7.5: Division of sublaws made by the candidate and select functions
### 7.5. The Operational Semantics of ECML

<table>
<thead>
<tr>
<th>Specification</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>notcandidates</td>
<td>The sublaws which were not candidates are added to C.</td>
</tr>
<tr>
<td>notpreholds</td>
<td>The sublaws which were candidates, but for which the precondition did not hold, are added to C.</td>
</tr>
<tr>
<td>notselected</td>
<td>The sublaws which were arguments of the select function, but were not selected, are added to C.</td>
</tr>
<tr>
<td>none</td>
<td>No sublaws are added to C.</td>
</tr>
<tr>
<td>concat</td>
<td>Sequences may be concatenated.</td>
</tr>
</tbody>
</table>

Table 7.7: Explanation of \( add_{sr} \) specifications

<table>
<thead>
<tr>
<th>( s(add_{sr}(l)) )</th>
<th>( add_{sr}(l) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>notcandidates</td>
<td>( { l_i</td>
</tr>
<tr>
<td>notpreholds</td>
<td>( { l_i</td>
</tr>
<tr>
<td>notselected</td>
<td>( { l_i</td>
</tr>
<tr>
<td>none</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>concat</td>
<td>Concatenation of sequences with the usual semantics</td>
</tr>
</tbody>
</table>

Table 7.8: Specification of the \( add_{sr} \) function

**Specification of \( add_{sr} \)**

This function determines when a sublaw \( l_i \) terminates, which sublaw(s) may enter \( C \). After the initial division of candidates and non-candidates, this function serves to update the set of candidates considered for application. The possible specifications and their explanations are as given in Table 7.9.

The precise definition of the function is given in Table 7.10. Note that the specification must satisfy the following constraint:

\[
add_{sr}(\sup(l))(l) = \text{next} \Rightarrow \text{select}_{sr}(\sup(l))(\sup(l)) = \text{first}
\]

i.e. it is meaningless to add the next sublaw to \( C \) if it is not in order to execute sublaws in a given sequence.

**Specification of \( termc \)**

The terminate condition is used to determine whether a law should terminate in a given state or not. For a simple law, it always terminate after execution. For a complex law,
<table>
<thead>
<tr>
<th>Specification</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>No sublaw is added to $C$ as the sublaw terminates. In a way, this means progress towards termination of its superlaw.</td>
</tr>
<tr>
<td>sublaw</td>
<td>The sublaw just terminated is added to $C$, so it may be applied again.</td>
</tr>
<tr>
<td>next</td>
<td>The next sublaw in the sequence of sublaws is added to $C$. It is used for various sequence relationships.</td>
</tr>
</tbody>
</table>

Table 7.9: Explanation of $taddC_{sr}$ specifications

<table>
<thead>
<tr>
<th>$s(taddC_{sr}(\sup(l_i))(l_i))$</th>
<th>$taddC_{sr}(\sup(l_i))(l_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>sublaw</td>
<td>$l_i$</td>
</tr>
<tr>
<td>next</td>
<td>$l_{i \mod n + 1}$</td>
</tr>
</tbody>
</table>

Table 7.10: Specification of the $taddC_{sr}$ function

The decision is taken based on informations about the execution state of its sublaws. The specifications and their explanations are given in Table 7.11. A precise definition of these conditions is given in Table 7.12. Note that standard temporal operators are used in the specifications, i.e $\mathbf{\Diamond}$ refers to past or present state(s), while $S$ is the since operator.

We have not looked into how one can prove the soundness of the specifications. However, we can specify a 'non-progress' situation, which if it occurs, should lead to a halt of the model execution. A model halts if in any state, no simple law can be applied, i.e.

$$\text{terminates}(l_\ast) \iff l_\ast \in \mathcal{E} \land \neg \exists l (\text{simple}(l) \land l \in \mathcal{E})$$

Note that this situation does not have to be a result of incorrectly specified functions. It may be because the models are incorrect. For instance, compare this with the infinite loop situation which may occur during execution of a computer program.

Examples of Specifications of Sublaw Relationships

**Sublaw relationships in PPP** Table 7.13 summarizes the meanings of the sublaw relationships found in the current PPP language. With the means for specifying a class of sublaw relationships, we can extend this basic set as needed, for instance to offer new languages to express process logic.
### 7.5. The Operational Semantics of ECML

<table>
<thead>
<tr>
<th>Specification</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>true</code></td>
<td>A superlaw terminates unconditionally when the (only) executing sublaw terminates.</td>
</tr>
<tr>
<td><code>lastsublaw</code></td>
<td>A superlaw terminates when its 'last' sublaw in the sequence of sublaws terminates.</td>
</tr>
<tr>
<td><code>no_sublaw_executes</code></td>
<td>A superlaw terminates when no sublaw executes.</td>
</tr>
<tr>
<td><code>all_executed</code></td>
<td>A superlaw terminates if all sublaws have executed (at least once.)</td>
</tr>
<tr>
<td><code>no_prec_hold</code></td>
<td>A superlaw terminates if no precondition of any sublaw holds.</td>
</tr>
<tr>
<td><code>prec_not_hold</code></td>
<td>A superlaw terminates if its precondition does not hold.</td>
</tr>
<tr>
<td><code>and, or, not</code></td>
<td>The conditions can be combined using logical connectives</td>
</tr>
</tbody>
</table>

Table 7.11: Explanation of `termc` specifications

<table>
<thead>
<tr>
<th><code>s(termc(l))</code></th>
<th><code>holds(termc(l))</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>true</code></td>
<td><code>∃l_i(sublaw(l_i, l) ∧ terminates(l_i))</code></td>
</tr>
<tr>
<td><code>lastsublaw</code></td>
<td><code>terminates(l_n)</code></td>
</tr>
<tr>
<td><code>no_sublaw_executes (n_s_e)</code></td>
<td><code>¬∃l_i(sublaw(l_i, l) ∧ ∗(l_i ∈ ℰ) ∧ ¬terminates(l_i))</code></td>
</tr>
<tr>
<td><code>all_executed (a_e)</code></td>
<td><code>∀l_i(sublaw(l_i, l) ⇒ (∗ l_i ∈ ℰ ∧ ∗(l_i ∉ ℰ) ∨ terminates(l_i)) ∧ ∗(l ∈ ℰ S l_i ∈ ℰ))</code></td>
</tr>
<tr>
<td><code>no_prec_hold (n_p_h)</code></td>
<td><code>¬∃l_i(sublaw(l_i, l) ∧ holds(prec(l_i)))</code></td>
</tr>
<tr>
<td><code>prec_not_holds (p_n_h)</code></td>
<td><code>¬holds(prec(l))</code></td>
</tr>
<tr>
<td><code>and, or, not</code></td>
<td>Usual semantics</td>
</tr>
</tbody>
</table>

Table 7.12: Specification of `termc`

**Other examples** We will now illustrate the power of ECML to model system dynamics by defining some additional sublaw relationships. Most of these are found in existing languages, and are already proven useful. We have summarized the meanings of these sublaw relationships in Table 7.14.

- The non-deterministic firing of transitions in Petri nets are in VCSEXEC relationships.
- In *production systems* and other forward-chaining systems, rules are in PROD structures. New data are continuously derived from known data until a particular conclusion is obtained or there are no more inferences to be made.
- The EXC relationship may be used to choose one law non-deterministically from a set of laws, for instance to simulate the environment of a system.
- To execute incomplete models, or to resolve non-determinism, the user or developer may be consulted. In the first BNM-interpreter [51], the user could select
Table 7.13: Sublaw relationships in PPP

<table>
<thead>
<tr>
<th>Function</th>
<th>VCS</th>
<th>SEQ</th>
<th>EXCSEQ</th>
<th>REPSEQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>cand</td>
<td>all</td>
<td>first</td>
<td>all</td>
<td>first</td>
</tr>
<tr>
<td>select</td>
<td>all</td>
<td>first</td>
<td>first</td>
<td>first</td>
</tr>
<tr>
<td>addC</td>
<td>notprechecks</td>
<td>notprechecks</td>
<td>none</td>
<td>notprechecks</td>
</tr>
<tr>
<td>taddC</td>
<td>sublaw</td>
<td>next</td>
<td>none</td>
<td>next</td>
</tr>
<tr>
<td>termc</td>
<td>n.s.e∧ n.p.h</td>
<td>lastsblaw</td>
<td>true</td>
<td>lastsblaw∧p.n.h</td>
</tr>
</tbody>
</table>

Table 7.14: Other examples of sublaw relationships

<table>
<thead>
<tr>
<th>Function</th>
<th>VCSEXCE</th>
<th>PROD</th>
<th>EXC</th>
</tr>
</thead>
<tbody>
<tr>
<td>cand</td>
<td>all</td>
<td>all</td>
<td>all</td>
</tr>
<tr>
<td>select</td>
<td>onerandom</td>
<td>all</td>
<td>onerandom</td>
</tr>
<tr>
<td>addC</td>
<td>none</td>
<td>notprechecks</td>
<td>none</td>
</tr>
<tr>
<td>taddC</td>
<td>sublaw</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>termc</td>
<td>n.s.e∧ n.p.h</td>
<td>n.s.e∧ n.p.h</td>
<td>true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>VCSUSER</th>
<th>VCSPRI</th>
<th>AGG</th>
</tr>
</thead>
<tbody>
<tr>
<td>cand</td>
<td>all</td>
<td>all</td>
<td>all</td>
</tr>
<tr>
<td>select</td>
<td>user</td>
<td>pri=max(pri)</td>
<td>all</td>
</tr>
<tr>
<td>addC</td>
<td>notprechecks∪notselected</td>
<td>notprechecks∪notselected</td>
<td>notprechecks</td>
</tr>
<tr>
<td>taddC</td>
<td>sublaw</td>
<td>sublaw</td>
<td>none</td>
</tr>
<tr>
<td>termc</td>
<td>n.s.e∧ n.p.h</td>
<td>n.s.e∧ n.p.h</td>
<td>a.e</td>
</tr>
</tbody>
</table>

which transitions to fire from the set of enabled transitions. Let us denote such a relationship VCSUSER.

- In some rule-based systems, priorities are given to rules in VCSPRI relationships, so that some rules always take priority over others in case they all have preconditions which hold.

- When an aggregate structure of sublaws is executed, all laws must be executed exactly once. We call this relationship AGG.
Chapter 8

A Rule Language for Translation Specification and Implementation

In this chapter we propose a language for specification and implementation of translations in CASE environments, TRL. First, we review some of the major requirements identified in Chapter 3, which guided the development of TRL. The language is then described with respect to its syntactical and semantical aspects. The use of TRL is illustrated with three small, yet realistic examples. Finally, we discuss the generality of our approach, and indicate how efficient implementations can be derived.

8.1 Introduction

The requirements identified in Chapter 3 have guided our work. Before we present the details of the approach taken, we will first go through the requirements and briefly indicate how we intend to satisfy them.

First, an important requirement is that a specification level for translations is provided, i.e. translations can be specified at the level of source and target language specification. We intend to provide a rule-based language which can be used to make knowledge about translations explicit, where rules can achieve goals and refer to the metamodels of the source and target language. Thereby we hope that translations can be more easily expressed and modified.

The translation specification language is primarily intended for prototype generation, but as we have already shown, the same principles are used in many other syntax-directed tasks. We will illustrate this by showing how an existing translation in PPP can be expressed.
Prototype generation requires that the translation rules must refer to both datamodel and parse tree representations. We will integrate these representations within a single language. However, all translations will be initiated by access to instances of basic constructs in the language metamodels.

We provide a language with a number of predefined actions, where the most important is the ability to decompose a translation problem into smaller problems, refer to other rules which take care of these, and finally to combine the partial results. Some depend on the particularities of the data structures of the repository, while others are more general. An example of the latter is a function to return a parse tree from a string and its producing grammar.

We do not intend to spend much work on how to interact with users in case of conflicting, alternative translations, how to record decisions made etc. Rather, we assume that for prototype generation, a single translation can be specified. Still, we open up for specification of alternative translation rules, and provide a very simple selection from the alternatives. It will remain a topic for further research how to deal with alternative translations in a more sophisticated manner.

8.2 The Form of Translation Rules

We have seen previously that some rule-based approaches to translation specification satisfy many of the requirements to translation facilities for CASE. In particular, they give the opportunity to make designer knowledge more explicit, and they can easily be linked to the specification of the source and target languages. In this chapter we present a rule language which draws upon some of the ideas developed by others, as presented in Chapter 3. However, as its major objective is to facilitate translation of conceptual models into prototypes, it needs to deal with mixed model representations, i.e. data model and parse-tree structures. Hence, the language can be seen as an attempt to integrate and extend previous approaches in a useful manner.

We have found it appropriate to divide rules into two major categories:

1. Basic-construct rules: Rules which refer directly to constructs in the language metamodel, i.e. where patterns in the source model are identified by some sort of query to a repository.

2. Construct-property rules: Rules which translate properties of basic constructs. These may for instance have values corresponding to parse tree structures.

Figure 8.1 illustrates the difference between the two rule categories. All translations must be initiated by basic-construct rules, which identify the basic patterns to translate. If additional rules are needed to translate the property values, references to construct-property rules are made. In the figure, a basic-construct rule is shown to translate a pattern in the
source model to a resulting pattern in the target model. Doing this, it initiates the application of a number of construct-property rules, some of them working on intermediate representations like trees etc. In the following presentation, we describe each rule category in a separate section.

Basic-Construct Rules

Basic-construct rules are used to specify how patterns in the source models are translated into patterns in the target model. The general form of a basic-construct rule is as follows:

```
[ name          Name
  goal_achieved Goal_achieved
  source_model  Source_model_pattern
  precondition  Precondition
  generate      Generate
  subgoals      Subgoals ]
```

Each of the rule components are described in the following:

**Name** is a name or identifier of the rule. It is used to reference rules in cases where more than one rule can achieve the same goal.

**Goal achieved** is a characterization (name) of the goal achieved by a rule when it is applied.

**Source model pattern** gives implicitly or explicitly a query to the repository where the source and target models are stored. The better case is that the rule simply refers directly to the metamodel of the source language, in a high level query language.
defined for the datamodel used for metamodeling. Then it will be easier to specify translations, as an automatic mapping of these high level queries to lower level queries of the repository is done. For advanced repositories based on variants of the ER-model or object-oriented datamodels, this high level of specification is directly supported by the repository.

Anyway, the query binds variables which are used as the rule is applied. Note that the query may also involve references to results of previous translations, i.e. target models already generated.

**Precondition** is an additional condition on the input pattern which must be true in order for the rule to apply. In many cases, it is most natural to include conditions directly in the source\_model\_pattern. However, we have included it here to offer alternative formulations when explicit conditions seem appropriate or are necessary, e.g. to separate out conditions on results from previous translations.

**Generate (Target model pattern)** is a list of construct instances with property values to be inserted in the repository, referring to the metamodel of the target language. The same comments as for source\_model\_pattern apply here. Variables used in source\_model\_pattern may be included in the target model patterns which are generated. The insertions are performed as the rule terminates.

**Subgoals** is a list of subgoals to be achieved in order to achieve the overall goal of the rule. The subgoals must be achieved in the sequence given. They belong to different categories, of which the most important are subgoals which refer to other rules, through which a subgoaling mechanism is established. We will describe both the different categories of subgoals and the subgoaling mechanism in later sections. Other categories of subgoals include predefined functions to e.g. combine partial results to an overall result, to parse strings to produce parse trees, and to generate new identifiers. Also, as we have explained before, the set of functions should be extendible.

**Construct-Property Rules**

These rules have an input-output behavior similar to procedures in programming languages. They are given translation problems from basic-construct rules and construct-property rules as input, and return solutions as they terminate.

The general form of a construct-property rule is as follows:

```
[ name         Name
  goal\_achieved Goal\_achieved
  source\_model  Source\_model\_pattern
  precondition  Precondition
  context       Context
  return        Return
  generate      Generate
  subgoals      Subgoals ]
```
8.2. The Form of Translation Rules

_Name_, _Goal_achieved, _Generate_, and _Subgoals_ have the same descriptions as above. The other components are as follows:

**Source model pattern** is here used as a means to pass on translation problems. It contains elements from a source model which will be directly used in the translation to elements of a target model. The elements may be property values, or portions of parse trees constructed during the translation. For the latter case, it is desirable to use patterns directly from the grammar specification. As an example, if a subtree corresponds to the production _A_ → _B_ C, then this production should be used to recognize the tree directly, or at least some high level structure, hiding the details of the actual tree representation.

**Precondition** tests properties of elements found in the source model pattern, e.g. to check if they have particular values.

**Context** contains contextual information which is used in deciding on the results of translations, and which also may be included in the results directly themselves. Referring back to the general translation principles identified in Chapter 3, this context provides the inherited properties of a translation rule.

**Return** contains elements which are returned as a solution to the given translation problem. A typical result would be portions of a constructed parse tree for the target model.

**Two Variants of TRL**

So far, TRL has been presented in an abstract, framework-like way. To be useful for translation specification, the syntax for the various rule components has to be specified. In the examples to come later in this chapter, we will use two variants of TRL, both of which may be characterized as 'sugared' PROLOG. A major reason for this is that the PPP repository is based on PROLOG facts, and we want to experiment with real translation problems within this environment. Other reasons for choosing PROLOG are related to implementation issues, and are given in the beginning of Chapter 10.

The first variant of TRL uses a simple data manipulation language for the datamodel of the metalanguage introduced in Chapter 2. Variables are declared to range over instances of constructs found in the language metamodels, and property values are accessed using the dot-notation, in a similar manner to relational calculus and ECML. Properties are inherited in generalization hierarchies in exactly the same way as in ECML. The source model patterns and conditions include use of logical connectives and comparisons of property values. The generated data are lists of property-value pairs to be inserted for a new construct instance.

The second variant uses PROLOG terms more freely to access repository data through a specified schema structure. Also returned and generated data are PROLOG terms. Both variants are specified in the TRL grammar in the appendix.
Chapter 8. A Rule Language for Translation Specification and Implementation

Subgoals in TRL

The subgoals of a rule are organized in a sequence, and they are achieved in the order given. Additionally, a choice structure can be used, where alternative lists of subgoals are chosen on the basis of evaluation of conditions. As briefly mentioned above, subgoals can be of different categories. We will now go through each of these categories. Note that some of them are independent of the repository datamodel, while others are suited to the particular data structures found in the repository.

Rule references can be made through a subgoaling mechanism. The translation problem is passed on along with contextual information, and the particular goal to be achieved. When a selected rule has been applied, a result is passed back. There are two kinds of rule references that can be made, depending on whether a number of identical subgoals are to be issued or not. The expression

\[
\text{tgoal}(\text{Goal}, \\
\quad \text{Source\_model\_pattern}, \\
\quad \text{Context}, \\
\quad \text{Return})
\]

issues the subgoal \text{Goal} for the given source model pattern and context. It expects to have a result returned in \text{Return}. We also have the alternative

\[
\text{forall}(\text{Element}, \\
\quad \text{Set}, \\
\quad \text{tgoal}(\text{Goal}, \\
\quad \quad \text{Source\_model\_pattern}, \\
\quad \quad \text{Context}, \\
\quad \quad \text{Return}), \\
\quad \text{Result})
\]

which provides a way to issue the same subgoal for each element in a set. The partial results are combined into a list in the order they are produced (\text{Result}). Normally, the set element is included in the source model pattern given for each subgoal issued.

Parsing and unparsing. The function \text{parse}(String, Tree) produces a parse tree for a given string produced by a specified grammar. Likewise, the function \text{unparse}(Tree, String) produces a string from the parse tree given. Note that all terminals are assumed to be given in the parse tree. The parsing function will most often be used as a basic-construct rule issues a subgoal to translate a property which has as value a string which corresponds to an expression. Then construct-property rules will translate the produced tree in a recursive manner. The unparse function may then be used e.g. just before the insertions into the target model.

Combination of partial results. This can be done by constructing a complex structure from a number of components. The types of components and complex structures will to
8.3. Rule Semantics and Control of Rule Execution

A large degree depend on the data structures found in the metamodel. For instance, in PROLOG, lists are the major data type, so a function to combine lists is necessary. combine_lists(Lists, List) returns List as the concatenation of the lists found in Lists. construct_function(Functionname, Arguments, Function) constructs a function with the given name and the given arguments. Furthermore, distribute_function(Multiple argument function, Binary argument function) constructs a function of two arguments from a function with a number of arguments of the same type. As an example, add(2, 3, 4) is translated into add(2, add(3, 4)). Remember that only functions with two arguments are allowed in ECML.

**Text manipulating functions.** We have found it necessary to include functions which manipulate text strings. combine_texts(Strings, Concatenation) produces the concatenation of a set of text strings. It has been found useful when generating names of elements needed in ECML. uppercase(Text, Text in uppercase letters) converts a string to the corresponding string with only uppercase letters. lowercase(Text, Text in lowercase letters) converts a string to the same string with only lowercase letters.

**Other functions.** Generally, functions can be defined and used as needed. One that is often used is a function which can generate new identifiers. This is useful as we will see, before generating new instances in the target model. new_id(Category, Id) returns Id as a new identifier within the given category, e.g. within PLD's.

**Alternative subgoals.** Alternative subgoals may be issued depending on conditions as follows:

\[
(\text{Condition } 1 \rightarrow \text{Subgoals } 1, \\
\ldots
\]

else \rightarrow \text{Subgoals } N)

Here, a condition takes the same form as a precondition, and Subgoals \( i \) is a list of subgoals as explained in this section. Note that this construct is similar to CASE constructs found in programming languages like e.g. PASCAL. Also, the construct is not strictly necessary, as each alternative could be the basis for a separate rule. However, sometimes it is more natural to include the various alternatives in one rule and make it more compact, instead of spreading the decisions on several rules.

In the appendix, the syntax of TRL is presented, with a complete list of all subgoals which have been used in the translations in the PPP repository.

8.3 Rule Semantics and Control of Rule Execution

A complete translation from a source model to a target model is specified by means of a translation program. The program is simply a sequence of goals to be achieved for the
translation to complete successfully:

\[
\text{t\_program([Goal1, \ldots, GoalN])}
\]

All goals in the list must be achievable by basic-construct rules which have been specified. There may be more than one rule which is capable of achieving the same goal. If these rules have overlapping instances in their source model patterns, they are placed in alternative groups. During execution of the program then, the user must select one of the alternatives. Rules with non-overlapping patterns which satisfy a requested goal, are all applied. For execution of construct-property rules, conflicting rules are found during execution, and presented to the user for selection. Two rules are alternatives if they match on goal, source model pattern, and context, and in addition, both preconditions hold. Here, a match means that the different parts are unifiable.

Figure 8.2 depicts the overall execution strategy. At the translation program level, a \text{t\_program} controls execution in the manner explained above. All goals are achieved by the application of selected basic-construct rules. A rule is selected if it matches the goal issued, it is chosen from its alternative group (if it is contained in such a group), and its precondition holds. Basic-construct rules may in turn issue subgoals, which are achieved by rules at the construct-property level. In the same manner, construct-property rules are selected if rule goals, source model pattern and context matches, and in addition, the precondition holds. Conflicting rules must be resolved by the user through a selection from the applicable rules. Note that in the translations we use in this thesis, we do not make use of alternative groups, and generally only one rule can be applied.

Finally, rules at the construct-property level can issue subgoals which are achieved by other rules at the same level.

From the brief presentation here, it should be clear that no backtracking among rules is used. A 'commit' to a rule is made at the time it is applied. The non-backtracking solution makes a procedural implementation possible. The following paragraphs explain execution
FOR ALL Goal IN t_program DO
BEGIN
    FOR ALL Alternative group IN Basic-construct rules WHERE
        FOR ALL Rule IN Alternative group Rule.goal MATCHES Goal
    BEGIN
        SELECT Rule FROM Alternative group;
        EXECUTE Rule;
    END;
END;

Figure 8.3: Execution of translation programs

REPEAT
    FIND Instantiation OF Source_model_pattern;
    IF Precondition HOLDS THEN BEGIN
        FOR ALL Subgoal IN Subgoals FROM FIRST TO LAST DO
            EXECUTE Subgoal;
        INSERT Generate;
    END;
UNTIL LAST Instantiation

Figure 8.4: Execution of basic-construct rules

of translation rules through algorithmic descriptions.

Execution of Translation Programs A translation program is executed by executing
rules which can achieve goals listed in the program. If we had no conflicting rules in
alternative groups, it would be sufficient to just execute all rules which can achieve a given
goal, if their precondition holds. However, with the alternative groups, one rule must be
selected first. The algorithm is as shown in Figure 8.3.

Execution of Basic-Construct Rules First, the query of the rule is performed to find
an instance of the source model pattern of the rule. Then, if the precondition of the rule
holds, all subgoals are executed, and finally new instance are stored as specified. This is
repeated for each instantiation. The algorithm is given in Figure 8.4.

Execution of Construct-Property Rules A source model pattern and a context are pro-
vided as part of the translation problem to be solved. We assume that the precondition of
the rule holds, and that the rule has been selected for execution. Then all subgoals are
executed, and afterwards new instances are stored, and results are returned. The algorithm
is given in Figure 8.5.
GIVEN Source_model_pattern,Context.
BEGIN
  FOR ALL Subgoal IN Subgoals FROM FIRST TO LAST DO
    EXECUTE Subgoal;
    INSERT Generate;
    RETURN Result;
END;

Figure 8.5: Execution of construct-property rules

CASE
  Subgoal = Compound: FIND FIRST i SO THAT HOLDS Condition i;
  EXECUTE Subgoal i;
  Subgoal = Function: EXECUTE Function;
  Subgoal = tgoal(): FIND Rules IN Construct-property rules
  WHICH MATCH AND WHERE Precondition HOLDS;
  SELECT Rule FROM Rules;
  EXECUTE Rule;
  Subgoal = forall(): FORALL Elements EXECUTE t.goal
END;

Figure 8.6: Execution of subgoals

Execution of Subgoals Functions are executed in a straightforward manner. The compound conditional structure is executed by finding the first condition which holds, and execute the corresponding subgoal(s). If references to other rules are made through the goals they can achieve, t.goal(Goal,S,M,P,Context,Return), first a set of rules which match the components of t.goal and which precondition hold is found. Then a selected rule from this set is executed. A match involves a unification of the components with the corresponding components of an existing rule. Figure 8.6 shows the algorithm.

8.4 Use of TRL - Three Examples

To illustrate the translation specification language, we will apply it to three small, yet realistic translations. First, we specify the translation from Petri net models to ECML models. Then we specify the translation from SQL queries to ECML queries. Finally, we specify translations from PrM processes to initial PLD structures. The examples are not fully completed, but they should be sufficient to illustrate the main characteristics of TRL, and hopefully the ease with which one can translate conceptual models into ECML for execution. Additionally, through the last specification we show that the language has a wider applicability. We will use TRL further in Chapter 11, where a translation from PPP models to ECML is specified.
Translation from Petri nets to ECML

Petri nets were described in Chapter 4. Figure 8.7 shows a metamodel of the language. We will see how this metamodel is referred to in the specification of translations from Petri net models to ECML models.

The translation strategy is as follows:

1. The Petri net construct is translated to a system law in ECML. Its sublaws correspond to the transitions in the net. These are organized in an VCOEXEC structure, as discussed in the previous chapter.

2. A transition with no input places, i.e. a source, is translated to an external event in ECML. Its associated operations add tokens to the output places of the transition when the event is selected.

3. A transition with no output places, i.e. a sink, is translated to a dynamic law in ECML. Its precondition is that all input places have at least one token, and its operations remove one token from each input place.

4. A transition with both input places and output places is translated to a dynamic law. Its precondition is as described for a sink, while its operations are the operations for a sink plus the operations for a source.

5. A place is translated to an integer variable in ECML. For simplicity we assume that a place can not hold tokens initially, so the variable is initialized to zero. Alternatively, we could have translated places to classes, and then have tokens as instances of these classes. However, we do not lose anything at this abstract level of computation by the chosen approach. For instance, to do an animation of a Petri net, all we need to know is the number of tokens in each place.

In the following, we will formulate translation rules for 1., 2., 3., and 5. We will use the first variant for TRL described above, but for the generated ECML constructs, we use to the syntax presented in Chapter 7. 1. can now be formulated as the following basic-construct rule:
2. can be specified by one basic-construct rule which generates the external event, and makes use of a construct-property rule to create the update for each output place.

3. can be formulated by one basic-construct rule which generates the dynamic law, using one construct-property rule which produces the conjuncts of the precondition, and one construct-property rule to generate update operations.
8.4. Use of TRL - Three Examples

We omit 4., since it can easily be specified as a combination of rules found for 2. and 3. The translation in 5. can be specified through a basic-construct rule which generates the variables corresponding to places, and the initialization operation:

Finally, assuming that we have a rule which correspond to 4) with the goal transition, the complete translation can be specified through the program

\[ t_{program}([[\text{petrinet,source,sink,transition,place}]) \]

Translation from SQL to ECML

This example will illustrate translations using parse trees. In order to make the example comprehensible, we limit ourselves to a very restricted version of SQL. We focus on translation of queries of the following kind:

\[ \text{SELECT } \text{Expressions FROM } \text{Tables WHERE } \text{Predicate} \]

We further assume that all attributes must be prefixed by their table name. The grammar for the simplified query language is given in Figure 8.8.
Expressions ::= Expression|Expression Expressions
Expression ::= Attribute|Expression Op Attribute|(Expression)
Attribute ::= <Tablename>.<Attribute>
Op ::= '+'|'-'|'*'|'/'
Tables ::= <Tablename>|<Tablename> Tables
Predicate ::= Expression Relop Expression|Predicate'|NOT' Predicate|
            Predicate 'AND' Predicate|Predicate 'OR' Predicate
Relop ::= '='|'>'|'<'|⋯

Figure 8.8: A simplified SQL

We now make the assumption that tables are represented as classes in ECML, and attributes as class properties. The main steps in the translation are then as follows:

1. Expressions in the SQL query are translated into corresponding ECML expressions.

2. The list of tables in the SQL query is translated into the variables and classes used in the formula with free variables of an ECML query.

3. The predicate of the SQL query is translated into a corresponding predicate in ECML, in a way very similar to that used under 1.

Using the translation rules, the query

```
SELECT Person.Name, Person.Wage FROM Person WHERE Person.Wage > 120
```

would be translated into

```
query([PERSON.name, PERSON.wage], PERSON: person
gt(PERSON.wage, 120)).
```

First, given a query to translate, the overall strategy can be captured in the following construct-property rule:
Note the abstract representation of parse trees, where the grammar production is used directly. Non-terminals on the left of productions appear as names in terms which hold the subtrees of non-terminals on the right. These subtrees are accessed through variables, and terminals appear as quoted strings. The parse tree must be produced by the parse function, which may be implemented using YACC or similar tools.

For expressions and predicates, we generally formulate one rule for each production in the grammar. For instance, the production $\text{Expression} ::= \text{Expression Op Attribute}$ corresponds to the following rule:

\[
\begin{align*}
\text{name} & \quad \text{expressionrule} \\
\text{goal\_achieved} & \quad \text{expression} \\
\text{source\_model} & \quad \text{expression}((\text{E.O.A})) \\
\text{precondition} & \quad \text{true} \\
\text{context} & \quad [] \\
\text{return} & \quad \text{Expression} \\
\text{generate} & \quad [] \\
\text{subgoals} & \quad \{\text{L\_goal(expression,E,[],Expr)}, \\
& \quad \text{L\_goal(operator,O,[],Op),} \\
& \quad \text{L\_goal(attribute,A,[],Attribute),} \\
& \quad \text{construct\_function(Op,[Attribute.Expr],Expression)}\}
\end{align*}
\]

Note that each of the non-terminals on the right side of the production are translated first, by issuing the appropriate subgoals, and then the partial results are combined into an expression in ECML.

The leaves of the input parse tree for expressions are constants and attributes. The rule which corresponds to the attributes can be specified as follows:

\[
\begin{align*}
\text{name} & \quad \text{attributerule} \\
\text{goal\_achieved} & \quad \text{attribute} \\
\text{source\_model} & \quad \text{attribute}((\text{Tablename.,',',Attribute\_name})) \\
\text{precondition} & \quad \text{true} \\
\text{context} & \quad [] \\
\text{return} & \quad \text{Class.attribute} \\
\text{generate} & \quad [] \\
\text{subgoals} & \quad \{\text{uppercase(Tablename,Class),} \\
& \quad \text{lowercase(Attribute\_name,Attribute)}\}
\end{align*}
\]

This rule will translate e.g. $\text{Person.Wage}$ into $\text{PERSON.wage}$. In addition to these rules, we need a rule which goes through the list of expressions in the query, and translates each expression in turn. The rule is omitted from this presentation. Instead we turn to the translation of the formula of the query. First, the tables referenced are used to produce the first part of the formula, where variables are declared to belong to specific classes. Then, as the argument of the last free variable, the translated predicate is inserted. Translation of predicates is done in very much the same way as with expressions, so we omit this here.
Translation from Process Ports to PLD structures

As described in Chapter 2, it is possible to generate initial PLD structures from information about the ports of the corresponding process in a PrM model. The complete translation is described in [136], and was implemented as a set of PROLOG clauses, unfortunately in a more ad-hoc manner. The main strategy is to

1. Construct an input structure corresponding to the input port of the process. Here, triggering flows must appear first in the generated structure.

2. Construct an output structure corresponding to the output port of the process. Terminating flows must appear last in the structure.

3. For each input flow, generate a receive construct, and add the types of the flow to the generated construct.

4. For each output flow, generate a send construct, and add the types of the flow to the generated construct.

5. Produce a layout of the generated structures, i.e. determine the position of the constructs on the screen, their size and so on.
8.4. Use of TRL - Three Examples

Figure 8.9: Translation of PrM ports to initial PLD structures

An example of this translation has been depicted in figure 8.9.

In the following we will make some simplifying assumptions. First, we focus only on the structures, and not on the translation of contents of PLD constructs. Second, we ignore the details of diagram layout. We further focus on translation of input ports only, as output ports are translated in a similar manner. Finally, we assume that there is a single triggering subport of an AND-port, and that the triggering subport and triggering flows are placed 'first' in the list of subports.

We assume that information about input ports are stored as

\[
\text{inportnode}(Id, Type1, Type2, Level, Subnodes)
\]

Here \(Id\) is an identifier of a port, \(Type1\) is one of (xor,and), and \(Type2\) is one of (single, repeat, cond, condrep, repcond). The meaning of these values should be clear from the presentation of PPP. \(Level\) indicates the level of the port in the hierarchical port structure. If it is 1, then \(Type1\) is 'and', \(Type2\) is 'single', and \(Subnodes\) is a list of only one identifier of a flow input to the process. Flows are in turn assumed to be represented by facts flow(Id, Name, From, To, Types), and PLD constructs are assumed to be represented as pld(Id, From, To, Right, Type, Process). The three components From, To, and Right hold the identifiers of PLD constructs immediately before, after, and to the right of a construct. A major problem in this translation, is to establish these links between the constructs. This is due to the fact that we translate a hierarchical structure into a flat graph structure in PLD.

To translate the input port structure, the main strategy is

1. Recognize the port. Depending on the port, construct an initial structure, e.g. for a repeating port, a loop construct is generated in PLD.

2. Translate each of the subports of the port, one by one in the order given.

3. If a node corresponds to a flow, generate a receive-construct.

Since there are ten possible combinations of the two types associated with ports, there are ten different rules for 1. There is one rule for translating subports within an AND-port, one rule for translation of alternatives within an XOR-port. These correspond to...
Finally, there is one rule which translates an input flow to a receive construct. Here, we will present one rule within each category, and together, these rules are sufficient for the translation depicted in Figure 8.9 above.

Using the second variant of TRL (corresponding to schema for PPP repository), we first specify the construct-property rule _and_rep_rule_ which translates a repeating AND-port:

```
[ name and_rep_rule
  goal_achieved and_rep
  source_model [and_rep,Portids]
  precondition true
  context [This,Previous,Process]
  return [This,Next]
  generate [pld(This,Previous,Next,Right,loop,Process)]
  subgoals [new_id(pld,Next),
            new_id(pld,Right),
            t_goal(andsurports,Portids,[Right,This,Process],..)]
```

This rule first generates a loop construct. It gets as its contextual information the identifier of the loop construct, the identifier of the previous construct which ‘points’ at this loop construct, and the identifier of the process to which the generated PLD constructs belong. When it has been applied, it returns the identifier of the loop construct, and the identifier of the construct immediately below it. These are in turn used by the rule which issued the goal _and_rep_, for instance the following rule which goes through the subports of an AND-port, and translates each of them in sequence.

```
[ name andsurportsrule
  goal_achieved andsurports
  source_model Portids
  precondition (first(Portids,Id),
               inportnode(Id,T1,T2,Level,Subnodes),
               Level>1)
  context [This,Previous,Process]
  return [Last,After]
  generate []
  subgoals [t_goal([..,[T1,T2,Subnodes],[This,Previous,Process],[L1,A1])
            rest(Portids,Rest),
            t_goal(andsurports,Rest,[A1,L1,Process],[Last,After])]
```

The strategy is simple. First, pick out and translate the first subport. Then translate the rest of the subports. Note that only the identifier of the last generated PLD construct and the identifier of the next construct to be generated are returned. Also note that PLD constructs are generated as each rule terminates, since they are placed within the _generate_ component of the rule.

Finally, we specify the rule which translates an input flow to a receive construct as follows:
8.5 On an Implementation of TRL

There will be additional rules for the case that a subport is the last in a sequence, to terminate the translation of subports.

8.5 On an Implementation of TRL

Interpreting translation rules is a fairly simple task, as has been shown through the algorithms presented earlier. In Chapter 10 we will describe a prototype of an interpreter for TRL developed in PROLOG. The purpose of this section is to briefly explain how a procedural implementation of a translation program can be obtained from a specification in TRL. In fact, the task of obtaining such an implementation can be seen as yet another translation task. The source language is now TRL, and the target language is the chosen implementation language. The target language will probably be some database (repository) programming language, or a 3GL with repository manipulation embedded. Naturally, we should be able to specify the implementation in TRL.

We will only sketch an implementation based on the assumption that there is always only one rule applicable at a time (i.e. no alternative groups, never two construct-property rules applicable for the same goal), and that the matching (unification) of components of rule references with components of rules can be made at 'translation time'.

The main principles for an implementation are depicted in Figure 8.10. First, we present the translation of the overall program, then the principles for implementation of the rules, and finally for the execution of subgoals of a rule.

Implementation of Translation Programs A translation program in TRL is implemented as a program in the implementation language. For each of the goals in the translation program, a set of procedure calls are inserted. There will be one procedure call for each of the basic-construct rules which can achieve the goal in the translation program. This is illustrated at the top of Figure 8.10.
Implementation of Basic-Construct Rules  Basic-construct rules are implemented as procedures with no parameters. The name of a procedure is the concatenation of the rule name and the rule goal. Variables of the procedures correspond to the variables used in the translation rule. Their types must be inferred from knowledge about the data structures used in the repository. An alternative would be to extend TRL with explicitly declared variables.

The source model pattern is implemented as a query. If the repository is based on a DBMS, then some sort of cursor mechanism must be established to go through the retrieved data. A loop is inserted to advance the cursor. The body of the loop consists of a check correspond-
ing to the precondition of the rule. If this condition is true, the statements corresponding to the subgoals of the rule are executed. Finally, an insert is made into the repository, corresponding to the generate component of the rule.

Note that it should be fairly easy to exchange the language for queries, conditions, and insertions in TRL, and then specify new implementations. As mentioned above, TRL is also a rule framework, just as much as a concrete language. Another point to make is that we could define translations from TRL with PROLOG, to an implementation in a 3GL with relational database manipulation embedded. These topics remain for further research, however.

**Implementation of Construct-Property Rules**

Construct-property rules are implemented as procedures as well, named in the same manner as above. The source model pattern, context and return components of the rules are given as parameters to the procedure. Local variables corresponding to variables used in the subgoals are added. The body of the procedure consists of statements which correspond to subgoals and generate components of the rule. The precondition is not contained in the procedure itself, rather it is propagated to the point where the procedure is called, as will be explained below.

TRL has integrated the translation of parse trees, and these can be passed between different rule applications. In the implementation, a representation of parse trees must be achieved through a chosen data structure in the implementation language.

**Implementation of Subgoal Execution**

All functions occurring as subgoals are translated into functions and procedures in the implementation language. The alternative-structures can be implemented as CASE statements or IF THEN ELSE statements in the implementation language.

Translation of rule references through t.goal(Goal,Source,Context,Return) is not as straightforward. There may be more than one rule which can achieve Goal, and at the same time match the other components in the reference. It is the preconditions of the rules which decide which rule to apply. In the implementation therefore, the rule reference is made through a CASE statement, with one alternative for each rule matching the components of the reference. An alternative has a condition corresponding to the precondition of the rule, and a single statement which is a call to the procedure corresponding to the rule. The Source, Context, and Return components are supplied as parameters to the called procedure. We see that there is no need to include the precondition in the code for the rule which is referenced, since at the time the procedure is called, the precondition is known to be true.
Chapter 9

Execution Traces and Explanation of Model Behavior

In this chapter, we briefly review the requirements to a general tracing component for execution tracing of conceptual models. We define a system history concept, and use this concept to specify a trace schema. The interface between a model executor and the tracing component is described, i.e. how different events are reported and stored in execution traces. Afterwards, we describe how the trace information can be retrieved by means of a declarative query language which offers a set of predefined views into the trace. We show how these views can be exploited to provide quick explanations of model behavior, and finally we describe how the tracing component may be interfaced with a general explanation generator.

9.1 Introduction

In the development of the tracing component, we have been guided by the requirements identified in Chapter 5. We can exploit many of the ideas encompassed in existing tracing components, but adapt and extend them in order to be able to deal with executable conceptual models.

The architecture of the tracing component and its connections with other components was given in Chapter 6. The two major subcomponents were the reported events handler which receives information about occurring events from a model executor and stores this information in an execution trace, and the trace query handler which retrieves necessary information from given queries.

We try to achieve a high degree of language independence through the use of a fixed, but general schema. The schema and the traces stored will be expressive enough to capture
necessary informations about state changes and their causes in terms of different types of events. The reporting of events will be done through calls to a reporting routine in the reported events handler, and does not disturb the behavior of the executing model. Trace queries offer different views into the execution traces, and may serve as part of an interface to an explanation generator developed by Gulla [47].

The basic principles of the tracing component with the query language, and a possible coupling to the explanation generator is described in [49]. The overall integration of validation techniques within the PPP environment is the topic in [138].

9.2 An Expanded System History Definition

Usually, the term system history is used to denote the sequence of states that a system goes through, e.g. Dubois et al. [39]. In order to be a basis for specification of a trace schema for traces which again will be used to explain model behavior, we will expand the system history concept. Access to execution information about external events, dynamic laws, and state laws which brought about state changes is then needed. Collectively denoting these state changing constructs events, we can regard a system history to be a directed graph as described in the following.

A system history is a directed graph $G_H = \{V_S, E_E\}$, where $V_S$ is a set of vertices, one for each state. $E_E$ are edges $(S_f, S_t)$ where $S_f, S_t \in V_S$, $f$ and $t$ are state numbers with $f < t$, and the direction of the edge is from $S_f$ to $S_t$. Each vertex $S_i$ is associated with a state vector, which is a list of all state components in the system and their corresponding values $(sc, v)$. Since there are two kinds of states, unstable states and stable states, there are two kinds of vertices. Stable state vertices have only edges corresponding to external events leading from them. Unstable state vertices correspond to unstable states.

Edges are basically of three different kinds, since there are three categories of state changing constructs: External events, dynamic laws, and state laws. Dynamic laws at all levels in hierarchies are included, i.e. not only the simple dynamic laws which have operations associated with them. We distinguish between edges for simple dynamic laws and edges for complex dynamic laws. State laws are included for the case that a rollback is performed and the new state is a copy of the previous stable state. The edges are associated with a state change vector, which specifies what state components have been changed, their new values, the types of operations performed, any state component values which are referenced in the computation, and any values of state components which are referenced in the precondition of dynamic laws.

In the following, \textit{ev} is an identifier of an external event, \textit{dl} is an identifier of a dynamic law, and \textit{sl} is an identifier of a state law. Further, let \textit{prec, ref, inputs} and \textit{outputs} be lists of state component - value pairs, in the same fashion as the state vector. Let \textit{new} be a triplet $(sc, v, t)$, where \textit{sc} and \textit{v} are as defined above, and \textit{t} is an \textit{operation type}. Finally, let \textit{change} be a pair $(ref, new)$, and \textit{allchange} a list of \textit{change}. 
We can now define the state changing vectors (SCV’s) as follows:

- **External event edges** include information about the state changes and any references made in computing them: \( SCV = (ev, all\text{change}) \).

- **Simple dynamic law edges** include information about values referenced in preconditions, values referenced when computing new state component values, inputs and outputs, and the changes made: \( SCV = (dl, prec, inputs, outputs, all\text{change}) \).

- **Complex dynamic law edges** include information about values referenced in preconditions and inputs and outputs: \( SCV = (dl, prec, inputs, outputs) \).

- **State law violation edges** include information about values referenced in the evaluation of the state law: \( SCV = (sl, ref) \).

Figure 9.1 shows an example history for an execution of the bank model presented in Chapter 2. The system, starting from state 0, enters an unstable state through the input of a new transaction. The process P1:Process Transaction starts its execution, and at the same time, the subprocess P1.1: Verify_transaction is initiated. After it has completed, the flow Withdrawal_transaction has an instance, and process P1.2 is applied. The next events correspond to execution of complex and simple dynamic laws in the PLD model for this process. In this case, the requested amount is found to be too large, so a new amount is read from the customer. After a new amount is read, it is found to be ok, and the process terminates by generating the flow Ok_withdrawal. Finally, processes P1.4: Withdraw amount and P1.5: Issue account statement are executed, and as the latter terminates, so does their superprocess, P1. Note that the figure only indicates the external events and dynamic laws which have been applied, and does not show the details of state vectors and state change vectors.

### 9.3 A General Trace Schema

The system history definition given above may be seen as a conceptual model from which we now will derive a trace schema. Taking a database approach for the tracing component, we are eventually restricted by the capabilities of the DBMS we decide to use. We are faced
with the usual trade-offs between time and space when designing the schema, having to take into account such things as update and query types and frequencies. Since we have a somewhat limited knowledge of these factors now, we will concentrate on satisfying the functional requirements identified in Chapter 5. We will develop a prototype of the tracing component in PROLOG\(^1\), and this has influenced on the schema presented below to some degree.

Basically, our task is this: Find a trace schema which tuples will represent system histories as defined above. There are three issues of particular importance in accomplishing this. First, even with relatively small conceptual models with only few class instances, it is infeasible to store the state vector for every state. For some types of trace queries where it is necessary to find the value of state components in particular states, it is certainly advantageous to do this. However, we will rely on knowledge about an initial state, the current state, and the state changes in between, in order to find state component values.

Second, the graph structure will be represented through the use state numbers. A 'from' and a 'to' property must be included in the schema relations which represent edges in the history graph. Both of these are state numbers, representing the state at the tail and the state at the head of the edge, respectively.

The final issue concerns the representation of state components and their values. One problem is that these values are of different types. Another problem is to choose the appropriate granularity of the components and their values. Some languages, including ECML, offer the possibility of using complex types of variables and properties. Theoretically, it is possible to store the component-value pairs at the finest granularity in such cases, but the problem then is to identify the components through queries later on. As an example, consider a variable var which is defined to be of type set of integer. In an operation, all set members which satisfy a certain property are updated. If all updates are to be stored individually, what reference should be used for the set members? In this case, it would be far easier to store the set as a whole, identified through the name of the variable. We choose a granularity where it is possible to identify properties of class instances and roles of variables being aggregates. For most cases, this level seems to be sufficient, but we will have to examine this issue in the future as we get more practical experience with the tracing component.

We now define the trace schema as follows:

```
dynamic LAW(Id,From,To,Precondition values,Inputs,Outputs)
external EVENT(Id,From,To)
state LAW(Id,From,To,Referenced values)
state CHANGE(Id,To,Referenced values,State component,Value,Type)
```

We have separated out the actual state changes from the representations of the edges, i.e. the all changes part presented for the system history. Each tuple of state change corresponds to change. This choice makes changes to particular components easier to identify.

\(^1\)The choice of PROLOG will be explained in Chapter 10
i.e. one avoids having to search through a list of changes. Note that this relation has included both an identifier and a state, in order to e.g. distinguish between two executions of the same dynamic law. Also note that in the schema, we do not distinguish between simple and complex laws. This information can be found in the conceptual model, but it can also be found in the trace by searching for state change tuples which refer to the laws. If no tuple is connected to a law, it is complex.

The domains used in the relations above are as follows: Id is an identifier of an external event, a dynamic law, or a state law. From, To are state numbers as discussed above. Preconditionvalues, Referencedvalues, Inputs, Outputs are lists of component-value pairs. Statecomponent is an identifier. It may be a variable name, a class instance identifier, or these two combined with rolenames using the dot-notation. Value is a simple or complex constant. In the coupling to ECML, we use the ECML representation of values. Type identifies the operations involved in state changes. It may be insert, update, delete, query, or input, with obvious meanings. If an instance is the member of several classes in a subclass hierarchy, it may be deleted from one class in the hierarchy, but remain an instance of all the superclasses. In such cases, it is necessary to include the class name. As an example, if person has a subclass employee, and an instance is deleted from employee, Type is delete(employee).

Figure 9.2 shows parts of an execution trace based on the schema above. It represents the same execution of the bank model as was used to exemplify the system history concept, and the state numbers included correspond with those used in Figure 9.1. For ease of reference, we have shown the correspondence with the PLD model given in Chapter 2.
as well. The situation is that a new withdrawal transaction is received. A person named 'arne' tries to withdraw an amount of 100, but the balance of his account is 80. A new amount is then requested, and a value of 80 is read. Finally, the amount is accepted, and an accepted withdrawal is sent.

9.4 Interface to a Model Executor: The Reported Events Handler

In each execution state, we generally assume that several events may be reported. The diversity of events have already been discussed above. For the cases we consider here, both external event edges, state law edges, and simple dynamic law edges go from one state to the immediate next state, hence it is sufficient to report them after their completion. Complex dynamic laws will execute over a number of states, and their initiation and termination must be reported separately. The reported events handler basically stores information as it is received, but it is also responsible for adding the temporal information, i.e. the from-state and the to-state as described above. It updates the state number each time a list of events are reported. The reported events handler has a single routine which is called from a model executor in order to report events. The routine takes a parameter which is a list of items for the different events as described below (refer back to the domain descriptions above).

\[
\begin{align*}
\text{external\_event}(Id,\text{List of state\_change tuples}) \\
\text{dynamic\_simple}(Id,\text{Preconditionvalues, List of state\_change tuples, Inputs, Outputs}) \\
\text{init\_complex}(Id,\text{Inputs}) \\
\text{term\_complex}(Id,\text{Outputs}) \\
\text{state\_law}(Id,\text{Referencedvalues})
\end{align*}
\]

9.5 A Query Language to Retrieve Trace Information

The Need for Different Views

Execution traces soon become large and incomprehensible. Database approaches to tracing components therefore provide a query language to be able to select interesting information from the traces. If this query language is the one of the DBMS used, it is not tailored to the task of explaining model behavior, and does not give any clues as to what queries should be made in order to understand the behavior.

In order to provide such clues, we suggest to offer different views into execution traces. Each view is associated with one or more trace query types. These queries are designed to match with commonly occurring event questions and hypothetical questions mentioned in Chapter 5. The views are listed in the following.
9.5. A Query Language to Retrieve Trace Information

State component view In this view, the focus is on one or more specified state components, and on the changes made to them during an execution period.

External event view This view focuses on the occurrence(s) of external events, which are referenced by name.

Law application view The focus is here on application of particular dynamic law(s). The reasons for their application and their effects on the system state are retrieved.

Law context view The context of a law is defined as the law predecessors in the hierarchical law structure, and any laws being executed concurrently at the same level in the law structure. Context queries return information about the context a law was applied in, and the applicability reasons for the laws in this context.

Refers view There are two ways in which a law may refer to another: By referring to a state component changed by the other law either in its precondition, or in the computation of the new value of a state component. Possible refers-to relationships can be computed statically, but in a model execution, only some of them will be 'instantiated'. As an example, if the precondition of a dynamic law requires a variable to be greater than zero, there may be many dynamic laws which have updated this variable. Following refers-to relationships in the execution trace provides a set of causal links which give us the computation paths from input to output, or the other way around.

State law view In the case that a state law is violated, this view provide the opportunity to retrieve information about the state components which have been referenced during the evaluation of the law.

State view Here, the focus is on a particular state, which is specified by qualifying it with the application or termination of dynamic laws, occurrence of external events etc. If necessary, the state vector can be computed for a state.

Combined views Views may be combined to provide more economical specification of desired trace information.

In the following, we will briefly present examples of queries corresponding to these views, expressed in a trace query language TRQL. The examples presented are taken from execution of the bank model. Note that we have not considered in detail how the user interface to the tracing components should be so far. In the next section, we return to the question types described in Chapter 5, and discuss in a more general way how trace queries can be used to enhance the understanding of model executions.

The grammar of TRQL and a detailed specification of the retrieved data for each query are given in the appendix.

The State Component View

Changes made to state components can either be caused by the application of simple dynamic laws or by external events. During an execution, the same component may be
changed several times, and the query type for this view gives an opportunity to find all changes made, and their order. The general form of the query is

\[
\text{FIND Occurrence CHANGES FOR Statecompref [WHERE Changecondition]}
\]

Each query refers to a specified execution period. Occurrence may either be ALL, FIRST, LAST, PREVIOUS, or NEXT. If information about all changes within the execution period is wanted, the former alternative is chosen. The first and the last change can be retrieved using the next two alternatives. The latter two alternatives are used in order to navigate through a list of changes made to the specified component(s).

Statecompref is either the name of a variable, a variable with a rolemame, a class name, a specified instance of a class, or the property of a class instance. This corresponds with the granularity of the state component information stored in the execution trace. Changecondition restricts the changes which are retrieved. The condition may be on the type of operation performed in the change, e.g. insert or update, or a condition on the values of state components which were changed at the same time.

The retrieved information includes a tuple of the dynamiclaw or external_event relation, and a tuple of the state_change relation. The two tuples are joined on identifier and state number. The former tuple may give the reasons for the state change (precondition holds), and the latter gives the references made to other state components in the computation of the new state of the changed component.

To illustrate the query language, we now give some examples from the bank model execution. The query

\[
\text{FIND LAST CHANGES FOR Ok.withdrawal WHERE insert}
\]

retrieves information about the latest execution of the PLD construct which generates an instance of the flow Ok.withdrawal, i.e. of construct pld11.

\[
\text{FIND LAST CHANGES FOR \{X.balance,free(X.account,equal(X.id,123))\}}
\]

retrieves information about the execution of the last update of the balance of an account with identifier 123.

\[
\text{FIND ALL CHANGES FOR p1.2}
\]
finds all changes made to a variable named \(p1.2\). This variable is a record of all the local variables in process \(p1.2\). For one execution of process Verify_amount, this corresponds to execution of PLD constructs pld1 and pld6.

Finally, note that the state component view may be combined with other views which retrieve information about changed state components. Statecompref may be a subquery which e.g. retrieves information about a simple dynamic law which has been applied. When combined with the state component view, this subquery is evaluated to the state components which have been changed by this dynamic law.

**The External Event View**

A query on the form

\[
\text{FIND Occurrence OF Externalevent}
\]

is used to retrieve information about external events. Occurrence is the same as above, while Externalevent is the identifier of an external event. All information about the specified event is retrieved, i.e. both one or more tuples of the external_event relation and all the associated tuples of the state_change relation. As an example,

\[
\text{FIND LAST OCCURRENCE OF Transaction}
\]

would retrieve information about the last input of the flow Transaction, which triggers the process Process_transaction.

**The Law Application View**

Information about law applications can be retrieved through the query

\[
\text{FIND Occurrence APPLICATIONS OF Lawref}
\]

Occurrence is the same as before, while Lawref is either the identifier of a dynamic law, or a subquery which identifies one or more dynamic laws. For instance, Lawref may be a query from the state component view. It is then assumed that one is interested in the applications of the dynamic law(s) which changed the state component.
For a complex dynamic law, one or more tuples of the dynamic_law relation is retrieved. For a simple law, associated tuples of the state_change relation are retrieved as well. As an example, in order to find the applicability reasons and life span of all executions of the process Issue_account_statement, the query

\textbf{FIND ALL APPLICATIONS OF p1.5}

can be issued.

\textbf{The Law Context View}

Information about executed superlaws can be retrieved through queries on the form

\textbf{FIND Superlaw OF Law}

Here, Superlaw is either \texttt{SUPER} or \texttt{ALLSUPER}. In the former case, information about the application of the immediate superlaw is retrieved, i.e. a tuple from the dynamic_law relation. In the latter case, information about execution of all superlaws in the law hierarchy is fetched. Law is a subquery which returns the application of a dynamic law, i.e. the identifier of a law together with the state number of the state it was applied in.

Note that we have assumed that sublaw relationships can be found by inspecting the conceptual model. Otherwise, we would have to add this information to the execution trace by including a relation superlaw(Sublaw,Superlaw) in the trace schema.

Using the bank model again, the query

\textbf{FIND ALLSUPER OF (FIND LAST APPLICATION OF pld5)}

retrieves information about the execution of constructs pld2 and pld3, as well as about execution of processes Verify_amount and Process_transaction.

\textbf{The Refers View}

The refers view is used to follow causal links, or computation paths in the executions. These links can be followed both in a forward and backward direction. For a language like PPP, the refers-to relationships can in part be taken directly from a model inspection
simply by following the possible data flows. For other languages, these relationships can not be found that directly.

In the backward direction, a dynamic law (simple or complex) may refer to simple dynamic laws or to external events. The query

\textbf{FIND IS-REFERRED-BY Law}

can be used to find all links in the backward direction. Law is here a query which returns the application of a dynamic law. For each application, the returned data is the set of dynamic law applications and external events with corresponding state changes which are referenced by the application.

In the forward direction, one can find which laws (simple or complex) which refer to a simple dynamic law or to an external event. The query is on the form

\textbf{FIND REFERS-TO Laworevent}

where Laworevent is a subquery which either returns the application of a simple dynamic law, or the occurrence of an external event. In the bank model, the query

\textbf{FIND IS-REFERRED-BY (FIND LAST CHANGES FOR Ok\_withdrawal WHERE insert)}

retrieves information about the execution of the constructs with identifiers pld1 and pld6, i.e. about the retrieval of the flow Withdrawal\_transaction, and about the update of the variable amount.

\textbf{The State Law View}

The query

\textbf{FIND Occurrence VIOLATIONS OF Statelaw}

can be used to get information about state law violations. It retrieves one or more tuples of the state\_law relation, i.e. one finds out which state components were the cause of the violation.
The State View

Information related to states include the events initiated and terminated in the states, as well as the state vector. A query on the form

\[
\text{FIND Occurrence STATES WHERE Statecondition}
\]

can be issued in order to identify particular states. Statecondition may require that a particular external event or dynamic law was initiated in the state, that a specified state component was changed etc. In order to retrieve information about all events related to a state, the following query can be given:

\[
\text{ALLEVENTS Stateview.}
\]

To find values of state components or expressions in specified states, the query

\[
\text{FIND VALUE OF Statecompexpression IN (Stateview)}
\]

can be issued. The expression may be arithmetic, and may include hypothetical values of state components. This is useful for answering hypothetical questions, e.g. what would have happened if the components had other values than they did.

Finally, the state view is used to specify the execution period which the queries implicitly refer to.

\[
\text{FROM STATE (Stateview) and TO STATE (Stateview)}
\]

are used to specify the beginning and end of this period, respectively. The subquery is then required to retrieve a single state number.

9.6 Understanding Executions Through Trace Queries

In this section we will show how the trace query language can be exploited to enhance the understanding of model executions. Trace queries in the form presented here are only relevant in case of event or hypothetical type of questions. We divide model behavior and the need for understanding into the following categories:
9.6. Understanding Executions Through Trace Queries

- Understanding obtained results.
- Understanding applied dynamic laws.
- Understanding event sequences.
- Understanding state law evaluations.

Each of these will be described and illustrated by examples in the following sections. We have chosen 'generic' examples of PrM models for illustration in order to achieve uniformity and ease of understanding, even if models of other languages may be more illustrative in showing the potential of the query language in some cases. Furthermore, we assume for simplicity that the leaf processes shown have their interior computations described by a set of rules, for instance taken from a decision table, so that values of output flows are computed directly from values of input flows.

Understanding obtained results

As noted earlier, usually only the external behavior of a system or model can be observed when it executes. A typical question relates to how certain results were obtained, that is, a rationale for the outputs given is required. In general however, it is necessary to explain not only final results, but intermediate results as well. Sometimes this will help in explaining the final results themselves. At other times the intermediate results may be of interest of their own.

Also, it may be desirable to know why not other, alternative results were obtained, or why not completely other sets of output data were given.

![Diagram](image)

Figure 9.3: A process model to illustrate explanation of obtained results

Some examples will illustrate the points made above. Consider the process model depicted in Figure 9.3. The flows f1 and f2 comes from other parts of a larger model. One of these triggers P1, which in turn produces f3. This flow triggers P2, and depending on its value, either f4 or f5 is produced and returned to the external agent as P2 terminates.

**Understanding obtained result of f4:** Assume that the model is executed, and that f4 has got an instance which is returned to the external agent (for instance a user). Support of the result is found in the trace by the rule which produced f4. A possible query would be:
FIND LAST CHANGES FOR f4 WHERE insert,

or alternatively, if the rule which produce f4 can be specified directly:

FIND LAST APPLICATION OF <Rule name>.

In both cases, data about the rule application is returned, of which its precondition gives the rationale for why the rule was applied.

**Understanding obtained result of f3:** This flow can be considered an intermediate result. Data about its execution can be obtained exactly the same way as above. We will later see how chains of intermediate results and applied laws can be found in order to understand relationships from system input to output.

**Understanding not obtained result of f5:** It might be of interest to know why an instance of f5 was not produced, rather than one of f4. To understand this phenomenon, it is necessary to show that none of the rules which could have produced f5 could be applied. This is achieved by first finding the state components referenced in the precondition of rules generating instances of f5, and then by finding the values of all these components in the state where the rule which generated an instance of f4 was applied. A possible way to obtain this information is by the query:

FIND VALUE OF \(<\text{Comp}>\) IN
(FIND LAST STATE WHERE INITIATED
(FIND LAST CHANGES FOR f4 WHERE insert))

This query would be a bit simpler if the rule which produced f4 could be referenced directly.

**Understanding not obtained results of f3:** This flow always gets an instance when P1 is executed. However, it might be the case that there exist other rules which would have produced other results, depending on whether f1 or f2 triggered the process, or depending on the triggering flow’s value(s). Explaining why these rules did not apply can be done in the same way as above, i.e. by showing that their preconditions did not hold.

Note that it is also possible to focus on the complete history of a state component, and use navigation to obtain information about each update. For instance, the query

FIND FIRST CHANGES FOR f3, followed by a series of FIND NEXT CHANGES FOR f3
would retrieve information about each generation and deletion of instances of f3 within a prespecified interval. Also, assume that f3 has a boolean property value, and that only cases where a 'TRUE' value of this property are of interest. Then

\[ \text{FIND ALL CHANGES FOR } [X, \text{free}(X, f3, \text{true})] \text{ WHERE } X.1 = \text{TRUE} \]

retrieves the supportive information.

**Understanding applied dynamic laws**

The rationale for law executions include information about the values of the state components referenced in the preconditions, and the laws' contexts. Of course, the laws in questions may be positioned anywhere in the law hierarchy. It may also be necessary to show why a law was not applied in a state. Also this is supported by contextual information, as well as by any preconditions that did not hold.

There are also other situations where context is significant in understanding behavior. An example is the current status of computation. Understanding this require information about terminated laws, currently executing laws, and laws about to be executed. Temporal relationships among laws may be difficult to understand. In some cases, the precedence/succession relationships will constrain law applications, and enforce particular sequences of law applications. In other situations, application sequences are undetermined beforehand.

The following examples show how queries may be formulated to understand law executions. Also, we will show how input/output relationships can be understood by following refers-to relationships.

The examples will be based on the process model in Figure 9.4. The model may seem a bit awkward, but it is an example model which illustrates the features we wish to show here. The decomposed process P1 is triggered by the receipt of the flow f1, which at the same time trigger the subprocess P1.1. This process executes and produces either f2 or f3, so either P1.2 or P1.4 is triggered. Each of these will further produce flows to trigger either P1.3 or P1.5.

**Understanding application of P1.3:** The simplest explanation only show why the precondition held. Supportive information can be found through the query

\[ \text{FIND LAST APPLICATION OF P1.3} \]

However, including more information about why its precondition held, together with an account of why its superprocess is being executed, will improve our understanding. In the
example model, P1.3 was applied either because of P1.2 or because of P1.4. The appropriate information can be found from the trace by the query

**FIND IS-REFERRED-BY (FIND LAST APPLICATION OF P1.3)**

This query will retrieve information about the rule of P1.2 or P1.4 which produced the triggering flow of P1.3. If one instead is interested in which processes were responsible, the query

**FIND SUPER OF (FIND IS-REFERRED-BY ...)**

can be used. Note here that the computation paths can be found by further references backwards, until eventually external events are reached.

Contextual information can generally be obtained by inspecting the law structure and then make a query about the superlaw, or it can be obtained directly as indicated above by the query

**FIND SUPER OF (FIND LAST APPLICATION OF P1.3).**
9.6. Understanding Executions Through Trace Queries

Figure 9.5: A process model to illustrate concurrency

Also, finding all superlaw relationships is obtained through the query

\[
\text{FIND ALLSUPER OF (FIND LAST APPLICATION OF P1.3)}
\]

**Understanding why P1.5 did not apply:** In general, this amounts to showing that preconditions were not met, and/or that the superlaw was not executing. We recognize this situation from the previous discussion on understanding alternative results. Here it is necessary to show that neither f4 nor f6 was triggering the process, and further to show why they were not.

**Finding the current status of computation:** Consider the process model shown in Figure 9.5. After P1 has terminated, both P2 and P4 will apply, and hence be executing concurrently. Further, it may be that P2 finishes before P4, so that P3 will execute concurrently with P4, or P4 will terminate before P2, in which case P3 will execute 'alone'. In any case, we can find information about concurrency by the query

\[
\text{FIND CONCURRENT OF (FIND LAST APPLICATION OF P3)},
\]

which will return information about any concurrently executing laws with the same superlaw as P3.

If process networks are large, and have a high degree of concurrency, the query will have an advantage compared to query individual laws one at a time to reveal simultaneous execution.
Figure 9.6: A process model to illustrate explanation of event sequences

Understanding event sequences

Here, we will focus on the sequencing of external events. To illustrate, we will use the process model shown in Figure 9.6.

When the model is in a stable state, the external agent can enforce two different external events; either f1 or f2 can get instances. That is, both these events are permitted. If f1 gets an instance, P1 is executed, and the execution enters a state where generating an instance of f2 is no longer permitted. Instead, after P3 has been initiated, generating f5 is required to proceed execution. After P3 has finished, a stable state is reached, and both f1 and f2 can again be given values by the external agent.

Understanding required inputs: Consider a situation where it is necessary to show why an event is required, e.g. that an instance of the flow f5 must be generated. In general, this requires that the condition of the event is fulfilled. Here, the condition is that P3 has been triggered, i.e. a reference to the current state of execution. This can be understood by showing why P3 was applied, and possibly why its superlaw was applied. Queries to retrieve such information have been shown above.

Understanding rejected events: In case an external event is rejected, it must be shown that its condition is not fulfilled, and possibly that other events are required instead. It may be that event conditions are modeled as state laws. After all, an event involves an update of the system state. If this is the case, a rejected event is equivalent to a violated state law, see below.
Understanding state law evaluations

Evaluations of state laws yield two kinds of answers, either a law was violated, or it was not. Only information about violations are stored in the execution traces.

![Diagram of a process model to illustrate explanation of state law evaluations](image)

**Figure 9.7: A process model to illustrate explanation of state law evaluations**

**Understanding state law violations:** Understanding of state law violations is generally supported by providing the data which result in a violated law. Consider the example in Figure 9.7. First, the values of f1 may be restricted by a state law, so that the input from the environment must be checked. If the state law of f1 is violated, information about this event can be found by the query

**FIND LAST VIOLATION OF f1-state law.**

Furthermore, assume that D1 represents a database, upon which a state law has been stated. If this state law has been violated, the same kind of query can be formulated to find supportive data. In addition, it may be desirable to find out how the violating data was produced. A query like

**FIND LAST CHANGES FOR (FIND LAST VIOLATION OF D1-state law)**

can be used to find the dynamic laws which updated these state components.

Note that in general, a state law having a quantified formula will refer to many state components. It is still an open question how and if such references should be presented.

**Understanding state law obedience:** It may be necessary to show why a state law was not violated, for validation purposes. This can be supported in much the same way as when it was shown why a dynamic law was not applied, i.e. by showing that its precondition did not hold in the state when it was evaluated.
Understanding Hypothetical Situations

Hypothetical questions may refer both to the past and the future. They are usually of the format 'What if.....?', e.g. 'What laws would apply if variable var had value val?', or 'What output would have been given if input was x?'. Some of these questions require that an execution of the system is performed, without actually changing the system state\(^2\). A trace will be recorded and used in the same way as normal. Other questions can be answered by querying the already existing trace.

To support answering of hypothetical questions we provide the possibility to evaluate expressions in specified states, and allow state components to have hypothetical values in the expressions. For instance, assume that a trace shows that a particular law was not applied in a state, and one wishes to know whether it would have been applied if a state component referred to in its precondition had another value. A query like

\[
\text{FIND VALUE OF } \langle \text{Expression involving hyp(Comp=Val)} \rangle \text{ IN (FIND LAST STATE...)}
\]

could be used to evaluate the precondition with another value of the state component in question.

9.7 Finding Fragments of Explanations

This section is to a large extent based on [49], which shows how Gulla's explanation generator [47] and the tracing component may be integrated. The explanation generator is intended for use both in model construction and model validation. It is a general component which can be integrated with existing CASE architectures. The explanations generated are useful for a number of purposes, for instance for learning to use CML's, for explanation of verification checks, for model inspections, and for explanations of model executions. The latter is the most interesting to us, since we have been focusing on model executions as a validation technique. Explanations in this category are divided into history explanations which explain results produced during execution, and input justifications which explain to the user why certain inputs are needed. In generating explanations, the component follows the two-stage process presented in Chapter 5. First, a deep explanation is generated in response to a request. The deep explanation gives the structure and the content of the explanation, and includes information from the language metamodel, the conceptual model, and the execution trace, collectively denoted the source model. In doing this, it accesses a user model and a context specification in order to tailor the explanation to the receiver, and to the current context in which the explanation was requested. From the deep explanation, a surface generator produces a textual description, possibly with portions of the conceptual included.

\(^2\)The system state is left unchanged, and the execution is hidden
General Principles for Deep Explanation Generation

The deep explanation generator includes different knowledge sources and a number of explanation strategies. All of these are expressed in an **explanation modeling language, EML**. As in EES [84, 87], plan operators are used to encode the strategies, and the generation proceeds through a top-down planning process. We will neither go into details of the plan operator formalism nor of the planning algorithm here. It is sufficient with a simple example of a plan operator and an example explanation, related to the execution of a conceptual model. The plan operator shown in Figure 9.8 has two parts: A **characterization predicate** (cause(Inst)), and an **attribute value structure**.

This operator is used to explain how some result from an execution trace is computed from the conceptual model. **HEAD** is the name of the discourse goal attained by the operator. **NUCLEUS** and **SATELLITE** are subgoals of the head in the sense that **HEAD** makes explicit the relationship between them, and is attained by attaining the subgoals. Corresponding to each of these subgoals, other strategies are formulated, or they are attained by including elements directly from the source model. Subgoals may be single, they may be chosen from a number of subgoals, or they may be included in a list of subgoals, from which as many as possible are attained and taken into the explanation. In the cause operator, **NUCLEUS** is a source model element, i.e. the relationship between a generated flow instance and an instance of a dynamic concept which produced the flow instance. This information is taken from the execution trace. **SATELLITE** is a characterization of a substrategy instantiation(Model), which gives a rationale for the produced flow instance.

**PRECONDITION** is a list of predicates which refer to the source model, and each of them must be true in order for the operator to apply. They bind variables used in the rest of the operator. The predicates of the precondition here are used to find the instance of the concept which produced the flow instance, its type, and the last precondition which held and led to the computation path which gave as result that the flow instance was generated.

To generate a deep explanation, an appropriate discourse goal is picked, and the corresponding plan operator decomposed into its subgoals. Using a standard unification-like planning algorithm, the generator expands the subgoals until all the unexpanded subgoals refer to elements of the source model. The precondition part of the operators, and also the user model and the context specification, determine which operators to use.
<table>
<thead>
<tr>
<th>Relationship</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>aggregation(d1, d2)</td>
<td>$d_2$ is executed as part of $d_1$, e.g. in a process decomposition</td>
</tr>
<tr>
<td>succeed(d1, d2)</td>
<td>When $d_1$ terminates, $d_2$ is initiated, e.g. in a sequence in a PLD</td>
</tr>
<tr>
<td>receive(d, f)</td>
<td>$d$ receives flow $f$, e.g. input flow to a process</td>
</tr>
<tr>
<td>generate(d, f)</td>
<td>$d$ produces flow $f$, e.g. output flow from a process</td>
</tr>
<tr>
<td>trigger(d1, d2)</td>
<td>$d_1$ triggers the execution of $d_2$</td>
</tr>
<tr>
<td>terminate(d1, d2)</td>
<td>$d_1$ terminates the execution of $d_2$</td>
</tr>
</tbody>
</table>

Table 9.1: Relationships referenced in plan operators

For the parts of the explanation which refer to execution traces, the necessary information can be included by first sending one or more trace queries to the trace query handler, and then converting parts of the retrieved information to the required formats. Basically, this is done by defining functions in the explanation generation component that correspond to subsets of returned tuples from trace queries. When one of these functions are called, its parameters are translated to forms suitable for trace queries, a specific query is performed, and parts of the query result are translated to the EML formalism.

Plan operators refer to traces through a set of constructs and relationships from the conceptual models which are executed. During execution, these constructs and relationships are instantiated and made available in the traces. Some important relationships are given in Table 9.1. The task of the interface functions mentioned above is to retrieve the required instantiations from the tracing component. In the following we indicate how the relationships from Table 9.1 correspond with queries which can be given to the tracing component.

**Aggregations** To find the latest execution of a dynamic law $d_1$ and its superlaw $d_2$, the queries **FIND LAST APPLICATION OF** $d_1$ and **FIND LAST APPLICATION OF** $d_2$ can be issued.

**Sequences** Exactly the same queries can be made as above for the case that $d_2$ succeeds $d_1$.

**Receipts of data flows** Consider the case when the relationship holds between a process and a data flow. To find the last execution of the process is straightforward. In order to find the last receipt of the flow, the query **FIND LAST CHANGES FOR** flow WHERE delete. In an execution of a PPP model, the received data here would be a tuple corresponding to the execution of a receive construct in a PLD model.

**Generation of data flows** Information about data flow generation is done in the same way as for receipts, except that a reference to an insertion is made in the query.

**Triggering and termination** For the case that the trigger relationship holds between an event and a dynamic law $dl$, the occurrence of the event may be represented by setting a boolean variable to 'true'. Then this variable is referenced in the precondition of the dynamic law, and the required information is found through the query **FIND LAST APPLICATION OF** $dl$. 
9.7. Finding Fragments of Explanations

![Diagram showing functions used in EML, translation, and trace queries]

Figure 9.9: The interface between EML and the trace queries

In Figure 9.9, we have indicated how three of the functions in the explanation generator are related to queries from the tracing component. The function get_last returns an EML identifier for the last generated instance of a model element. Function get_beh returns the model statement instance responsible for the generation of a specified instance value, and pre_inst returns the instance last used as preconditions for the given model element.

An Explanation Example

Consider the execution of the bank model again, where the customer tries to withdraw an amount which is too large. Receiving an error message from the system, the following dialog may take place.

| system:          | Error - available amount is 80. |
| user:            | Why was my withdrawal rejected? |
| system:          | Withdrawal_rejection was generated because account_balance is less than withdrawal_transaction's amount. |
|                  | Account_balance was 80 and Amount 100. |

An indication of the deep explanation and the relationships to the text which is generated, is given in Figure 9.10. Compare the explanation with the PLD model presented in Chapter 2, and note that the condition of the alternative construct above the send construct which produced the flow Withdrawal_rejection is included. Also, the values of the two variables which are compared are included. In addition, it is possible to illustrate the explanation with the related portion of the PLD model. Hence, the explanation references the cause of the generated flow directly.
Figure 9.10: A deep explanation
Chapter 10

A Description of Component Prototypes

We first present some technical issues related to the implementation of a prototype of the proposed environment. Then we describe the core algorithms for the interpreters of the three languages developed. Most emphasis is made on the ECML interpreter, since this is the most complex of the three. The interpreters are documented in [137].

10.1 Some Technical Considerations

We have chosen to develop interpreters for the three languages TRL, ECML, and TRQL, mostly because the objective has been to illustrate and validate ideas at this stage. In Chapter 8, we also sketched a procedural implementation of TRL. For ECML, we have found direct interpretation to be most feasible, mostly because of an easier implementation of the different sublaw relationships.

The interpreters have been developed in PROLOG, for three reasons. First, the current PPP environment was developed in PROLOG, storing PPP models as PROLOG facts. Hence, a smooth integration with PPP was expected with this choice of language. Second, PROLOG is a language suitable for prototyping in general, as noted in Chapter 4. Finally, the metaprogramming capabilities of PROLOG makes it suitable for development of interpreters.

10.2 A TRL Interpreter

The interpreter for TRL has been implemented according to the algorithms presented in Chapter 8, with some minor limitations. As already mentioned in Chapter 8, we have not considered the treatment of situations where more than one translation rule is applicable.
Chapter 10. A Description of Component Prototypes

at a time. A very simple solution would be to just present the various alternatives to the user, and then let the user choose one of these. More advanced solutions would also give guidance to the user as to what rule should be chosen, and also store the choices made for later translations.

We have implemented most of the functions listed in Chapter 8, i.e. the functions which can be given as subgoals in translation rules. However, the parse function is a very simple parser for context free grammars without explicit operator precedence. Also, the unparses function has not been implemented. There are already a lot of available algorithms for parsing, and implementing them is only a matter of time spending. We can also make an interface to e.g. YACC, in order to solve partial translation problems.

With the limitations given above, the TRL interpreter has been implemented as a small PROLOG program. Backtracking is the means by which source model patterns of meta-entity rules are instantiated. Rules are represented as PROLOG facts. In the next chapter, we illustrate the usefulness of TRL in translating PPP models to ECML.

10.3 An ECML Interpreter

Fact Structures for State Components

The facts that a class instance is member of a class, and that a variable is of a particular type, are represented as follows:

\[
\text{instance(Class,Identifier)} \quad \text{variable(Type,Name)}
\]

Aggregates may represent complex properties of class instances, or part of a complex variable, e.g. a record like variable. Each aggregate has its own identifier, which is used to access its component values:

\[
\text{aggregate(Type,Identifier)}
\]

Finally, value-triplets are used to store the values of class properties and variables. A value-triplet has the form:

\[
\text{value(Identifier,Role,Value)}
\]

Here, Identifier may be a class instance, or the name of a variable. Role is the name of a property of a class, or a role in another aggregate structure. For a variable which is not an aggregate, it is 'nrole'. Value may be e.g. a simple constant, collection of constants (set,list etc.), a reference to an identifier (ids(Identifier)), or a collection of identifiers. As an example, if an account account@1 has a balance 1200, an interest rate of 7.5, and is owned by customer customer@1, this can be represented by the facts
instance(account@1)
value(account@1,balance,1200)
value(account@1,interest,7.5)
value(account@1,owner,ids(customer@1))

We chose the representation above in order to make it easier to manage manipulations of complex objects. An alternative would have been to collect all properties of an instance into one complex relation. However, some operations, especially updates, would then be more difficult to perform. Also, with this representation, it is very easy to access data by following relationships between class instances. As an example, to find the name of the customer which owns account@1, starting from the given account, two value-triplets need to be accessed, e.g.:

value(account@1,owner,ids(customer@1)), and
value(customer@1,name,\text{'arne'} )

Main Loop and Law Transitions

The main loop of the interpreter is given in Figure 10.1. It presents external events which can be reported by the user, and then reads the chosen event by name. Then, the operations associated with the event are performed, usually these are some input operations. Assuming now that the system is in an unstable state afterwards, an execution cycle is initiated to transform the system into a stable state. When a stable state has been reached, a new event can be input, and the cycle repeats.

The execution cycle in Figure 10.2 is the core of the interpreter. It is responsible for law transitions between the three sets \( \mathcal{E} \), \( \mathcal{C} \), and \( \mathcal{N} \), as these were specified in Chapter 7. However, it only keeps explicitly track of the members of \( \mathcal{C} \), since it is from this set that the search for applicable laws is initiated. Also, the other two sets \( \mathcal{E} \) and \( \mathcal{N} \) can easily be derived from knowledge of the simple laws which are applicable in a state, and knowledge about \( \mathcal{C} \):

\[
\begin{align*}
l \in \mathcal{C} & \Rightarrow \text{superlaw}(l) \in \mathcal{E} \\
l \in \mathcal{E} & \Rightarrow \text{superlaw}(l) \in \mathcal{E} \\
\mathcal{N} = \mathcal{C} \setminus (\mathcal{E} \cup \mathcal{C})
\end{align*}
\]

The basic idea is to perform the search for applicable simple laws top-down, starting with the laws in \( \mathcal{C} \). After execution of the applicable laws, termination of laws is performed bottom-up, starting from the set of executed laws. An alternative would be to examine all laws in the law structure for the transition conditions specified in Chapter 7. Then many unnecessary tests would be performed. The specification of the sublaw relationships suggest that application of laws is done top-down, and termination of laws bottom-up. We
Main_loop
BEGIN
    WRITE(External_events);
    READ(Event);
    REPEAT
        Perform_operations(Event);
        Execute_cycle;
        WRITE(External_events);
        READ(Event)
    UNTIL Event=nil;
END;

Figure 10.1: Main loop of interpreter

Execute_cycle
BEGIN
    \text{C}:=\{\text{System\_law}\};
    REPEAT
        Find_applicable_laws(\text{C}, \text{New\_to\_C}, \text{New\_to\_E}, \text{Simple\_to\_E});
        Execute(\text{Simple\_to\_E});
        Terminate_laws(\text{Simple\_to\_E}, \text{Term\_new\_to\_C}, \text{Term\_remove\_C}, \text{C}, \text{term});
        Evaluate_postconditions(\text{Simple\_to\_E}, \text{C}, \text{term});
        \text{C}:=((\text{C}\setminus \text{New\_to\_E})\cup \text{New\_to\_C}\cup \text{Term\_new\_to\_C})\setminus \text{Term\_remove\_C};
    UNTIL \text{C}=\emptyset;
END;

Figure 10.2: Execution cycle of interpreter

have specified interpreter procedures for some of the most common sublaw relationships, and we have also specified general procedures which work with all relationships according to their specification. We will later show how more efficient interpreter procedures can be generated automatically from the sublaw relationship specifications.

Assuming an initially unstable state, \text{C} is initialized to a set consisting of the root in the law hierarchy, i.e. the system law. It then calls the procedure Find_applicable_laws to search down the law hierarchy to find simple applicable laws (Simple\_to\_E), the total set of applicable laws (New\_to\_E), and new laws which are to enter \text{C} in the next cycle (New\_to\_C). Hence, it is responsible for the three transitions from \text{C} to \text{E}, from \text{N} to \text{E}, and from \text{N} to \text{C}. The procedure is described in more detail below.

After a set of applicable laws has been found, their operations are performed by the procedure Execute. Note that all operations are performed atomically from one state to the next.

The next step is to terminate laws according to their termination conditions, and imple-
ment the transitions from $\mathcal{E}$ to $\mathcal{C}$, from $\mathcal{E}$ to $\mathcal{N}$, and from $\mathcal{C}$ to $\mathcal{N}$. This process is performed bottom-up, starting from the set of executed simple laws. The terminate conditions of their superlaws are checked. If they hold, the laws terminate, and the process continues upwards. During this process, laws to add to $\mathcal{C}$ are found (Term_new_to_\mathcal{C}), laws to remove from $\mathcal{C}$ are found (Term_remove_\mathcal{C}), and complex laws which terminated are found (C_term). After the termination of dynamic laws, their postconditions are checked. Finally, the new $\mathcal{C}$ is computed from the old $\mathcal{C}$, New_to_\mathcal{E}, New_to_\mathcal{C}, Term_new_to_\mathcal{C}, and Term_remove_\mathcal{C}. The execution cycle terminates when $\mathcal{C}$ is empty. Note that both New_to_\mathcal{E}$ and C_term are stored in the execution trace, if the interpreter is interfaced to the tracing component. This is not shown in the figure.

The algorithm for Find_applicable_laws is given in Figure 10.3. It goes through all laws in $\mathcal{C}$, and initiate a top-down search from each law, if possible, through a call to the recursive procedure apply. It distinguishes between two cases: 1) The member of $\mathcal{C}$ is single, meaning that if its precondition holds, it is to be applied directly, and the top-down search continues from its sublaws if it is a complex law, 2) the member is grouped together with some sibling sublaws, to make it easier to perform a selection of laws which have preconditions holding. This depends on the semantics of the sublaw relationship between the sublaws and their superlaw. If all sublaws are selected, it is not necessary to have the grouping.

If the law examined in $\mathcal{C}$ is single and simple, it can be directly applied and is returned as part of S$\mathcal{E}$. If it is complex and it applies, the candidate and select functions defined for the sublaw relationship it has with its sublaws, determine the further search. The generic structure of the apply procedure is as shown in Figure 10.4. The presence and content of each part depends on the candidate and select functions, as well as on the terminate condition and add$\mathcal{C}$. We will later describe how such apply procedures can be automatically generated from the specification of sublaw relationships.

We have an analogous case with terminate procedures which are called from Terminate_laws in the execution cycle. Their generic structures are as shown in Figure 10.5. Also here, the presence and content of each part depends on the specification of sublaw relationships.

We can now illustrate the execution cycle by means of a small example. Figure 10.6 shows a law hierarchy, and the distribution of laws among the three sets $\mathcal{E}$, $\mathcal{C}$, and $\mathcal{N}$ as the model is executed. We have used the sublaw relationships SEQ, VCS, AGG, and EXC for illustration. The model is executed in four cycles, as depicted in parts a) to i). In a), the only law in $\mathcal{C}$ is the system law $l1$. Its precondition is true, so it is applied. Also $l2$ and $l5$ applies, so in b) $l1$, $l2$, and $l5$ are executing, and the variable b is updated to 3 by $l5$. When $l5$ terminates, $l6$ enters $\mathcal{C}$, since $l2$ is a SEQ. In the next cycle, $l3$ and its sublaw $l7$ are applied, but $l6$ can not be applied yet. Hence $l2$ and $l3$ are executing concurrently in d). As $l7$ terminates, $l3$ is still executing since $l8$ must be applied as well in the aggregated structure. Note that $l8$ entered the $\mathcal{C}$ as $l3$ and $l7$ were applied. Next, both $l6$, $l4$, and $l10$ are applied in f), and the operations of the simple laws are executed. When $l6$ terminates, $l2$ is terminated as well, but $l2$ enters the $\mathcal{C}$ since $l1$ is a VCS. The same is the case with $l4$ as $l10$ terminates. This is shown in g). In h), $l8$ is applied, and afterwards the execution is halted with no applicable laws in i).
Find_applicable_laws(C,Nc,Nc,Sε,Sε)
FOR ALL l ∈ C DO
BEGIN
IF Single(l) THEN
BEGIN
IF Holds_precondition(l) THEN
BEGIN
IF Complex(l,Struct,Sublaws)THEN
BEGIN
Apply(l,Struct,Sublaws,SNC,SNc,Sε,SSε);
NC:=NC∪SNc;
Nc:=Nc∪SNc∪{1};
Sε:=Sε∪SSε;
END
ELSE
BEGIN
Sε:=Sε∪{1}
Nc:=Nc∪{1}
END
END
END
ELSE
BEGIN
Preholds:=\{1' | l' ∈ l ∧ Holds_precondition(l')\}
Selectlaws(Preholds,Selected);
FOR ALL l' ∈ Selected DO
BEGIN
IF Complex(l',Struct,Sublaws) THEN
BEGIN
Apply(l',Struct,Sublaws,SNC,SNc,Sε,SSε);
NC:=NC∪SNc;
Nc:=Nc∪SNc∪{1'};
Sε:=Sε∪SSε;
END
ELSE
BEGIN
Sε:=Sε∪\{1'\}
Nc:=Nc∪\{1'\}
END
END;
END;
END;

Figure 10.3: Searching for applicable simple laws
10.3. An ECML Interpreter

Figure 10.4: The generic structure of apply

Head:-<Retract 'executes' for the law which terminated>
<IF the terminate condition of the superlaw holds THEN>
<Retract 'executed' for all sublaws>
<Try to terminate super-superlaw>
<Compute the laws to add to C>
<Compute the total set of terminated superlaws>
<Remove sublaws from C>

<ELSE>
<Compute the laws to add to C>
<Compute the total set of terminated superlaws>

Figure 10.5: The generic structure of terminate

Evaluation of Functional Expressions and Conditions

We will only illustrate the main principles for evaluation of functional expressions and conditions. Excerpts of the algorithm for evaluation of functional expressions is given in Figure 10.7. Constants are evaluated to themselves, variables to their values, and properties of class instances are evaluated by evaluating the value-field of their value triplets. A reference to a class instance is evaluated to itself. Functions are evaluated applicatively, i.e. by first evaluating their arguments, and then the function itself. In the figure, we have also indicated how externally specified functions may be called. So far, these functions must be specified in PROLOG as well.

Evaluation of conditions is presented in Figure 10.8. Basic predicates are evaluated in the same way as functions above, only they give a truth value as result. Quantified sentences are evaluated straightforwardly, since variables range over finite domains. In the PROLOG implementation, the next instance to test is found by a forced backtracking. Note that when a universally quantified sentence is evaluated, an escape is made as soon as it is clear that it will not hold. If the sentence is true, all instances need to be checked. A similar approach is taken in the evaluation of existentially quantified sentences. Although not shown in the algorithms, it is also possible for a variable to range over a domain other than a class, e.g. over a set-valued property, or a set-valued variable. In these cases, the set must first be computed, and afterwards the variable is instantiated to each of its members. Otherwise, the sentence is evaluated in the same manner.
Figure 10.6: A simple execution example
Evaluate(Expression)
BEGIN
  CASE
    Expression=Number: RETURN Number;
    Expression=Variable: RETURN Value(Variable);
    Expression=Function(Arg1,Arg2):
      BEGIN
        V1:=Evaluate(Arg1);
        V2:=Evaluate(Arg2);
        RETURN Evaluate(Function(V1,V2));
      END
    Expression=Externalfunction(Args):
      BEGIN
        Evaluatedargs:=Evaluate(Args);
        Externalfunction(Evaluatedargs,V);
        RETURN V;
      END;
  END
END

Figure 10.7: Evaluation of functional expressions

Interpretation of Operations

Queries

As explained in Chapter 7, there are three kinds of queries, two of which are evaluation of functional expressions and conditions. The more general query has the form query(Expressionlist,Freeformula). Expressionlist is a list of expressions which are to be evaluated for each distinct instantiation of the free variables in Freeformula. In the algorithm in Figure 10.9, it is shown how these instantiations can be found in a manner similar to evaluation of existentially quantified sentences. The main difference is that here, an escape is not made when an instantiation is found which makes the formula true, rather it is collected so that all instantiations are found. When an instantiation is found, the functional expressions can be evaluated in the manner described above. Note that in the PROLOG implementation, Expressionlist is passed on as argument of Holds, so that each variable referenced in the list is already instantiated when it is found that the formula is true.
Holds(Predicate(X,Y))
BEGIN
    V1:=Evaluate(X);
    V2:=Evaluate(Y);
    IF Holds(Predicate(V1,V2)) THEN RETURN TRUE
    ELSE RETURN FALSE;
END

Holds(\forall X:C P(X))
BEGIN
    X:=Get_next_instance(C);
    WHILE X\neq nil DO
    BEGIN
        IF NOT(Holds(P(X))) THEN RETURN FALSE
        ELSE X:=Get_next_instance(C);
    END;
    RETURN TRUE;
END;

Holds(\exists X:C P(X))
BEGIN
    X:=Get_next_instance(C);
    WHILE X\neq nil DO
    BEGIN
        IF Holds(P(X)) THEN RETURN TRUE
        ELSE X:=Get_next_instance(C);
    END;
    RETURN FALSE;
END;

Figure 10.8: Evaluation of sentences

Holds(X:C P(X), Instantiation)
BEGIN
    X:=Get_next_instance(C);
    WHILE X\neq nil DO
    BEGIN
        IF Holds(P(X), I) THEN Instantiation:=Instantiation\cup\{(X,I)\};
        X:=Get_next_instance(C);
    END;
END;

Figure 10.9: Evaluation of queries
10.3. An ECML Interpreter

Insert(Subcl, New, Freeformula, Assign, Super, Laws)
BEGIN
    IF Insert_holds(Subcl, New, Freeformula, Assign, Super) THEN
        Find_insert_statelaws(Subcl, Laws);
    END;

Performinserts(Class, New, Assignments, Super)
BEGIN
    Create_empty(Class, New, Super);
    Performupdates(Assignments);
END;

Figure 10.10: Insertion of a subclass instance

Insertions

As described in Chapter 7, there are five different variants of insertions. In two of these, a new class instance is read from the keyboard. The principle followed is to use the property types to guide the reading of values. The same approach is taken when the value of a variable is read. A special problem is when the new instance refers to existing instances. Then, the user can specify these instances through a formula with free variables which is instantiated in a way similar to what is the case with queries.

Here, we give the algorithms for insertion in a more complex case. The main algorithms are given in Figure 10.10. A subclass instance (New is to be inserted in class Subcl, and its property values can be derived from values of properties of other instances and variables. These assignments are found in Assign, and the values referred to are found through instantiations of Freeformula. Super identifies the new instance as a member of a superclass. Insert_holds is a procedure similar to Holds used previously. It searches for instantiations of free variables in Freeformula which makes the formula true, and for each instantiation found, a call is made to Performinserts. Insertions are made by first creating an empty structure corresponding to the type of the class, and then update this structure with the assignments given. Note that when a subclass instance is inserted, it must already exist as an instance of the superclass. Then the insertion makes this instance an instance of the subclass as well, and adds the extra properties of the subclass.

In a later section, we return to how relevant state laws are found and checked. We see that if the insertion is performed successfully, the relevant state laws are found and returned. Also, Performupdates will be described in connection with the update operation.
Deletions

Deletions are based on many of the same principles as insertions. The algorithms are given in Figure 10.11. For each instantiation of free variables in Freeformula which makes the formula true, the instances given by the variables in Variables are deleted. For a successful deletion, relevant state laws are found. Instances are deleted by deleting all property values (Find_all_roles and Delete_roles). In addition, all property values of other instances which refer to the deleted instance, must be nullified (Delete_refers_to). Finally, the instance is deleted from a specified class. If this class has subclasses, the instance must be deleted from the subclasses of which it is a member (Delete_subclass_instances). If it is also a member of some superclass(es), it is however not deleted from these, unless this is explicitly specified.

Updates

Updates also make use of instantiated free variables in exactly the same way as insertions and deletions. The algorithm is given in Figure 10.12. First, so called update paths are found to check that the updates are possible as specified. It is not possible to update instances of a class C on properties which belong to subclasses of C. It is however possible to update inherited properties. As an example of an update path, consider the following update:
Update(Freeformula,Assignments)
BEGIN
   Find_update_paths(Freeformula,Assignments,Paths);
   IF Updates_possible(Paths) THEN
      BEGIN
         IF Update_holds(Freeformula,Variables) THEN
            BEGIN
               Find_delete_statelaws(Freeformula,Variables,Laws);
               Check_delete_statelaws(Laws);
            END;
         IF Temporaries THEN Expand_temporaries;
      END;
   END;
END;

Performupdates(Updates)
BEGIN
   FOR ALL (Statecomponent,Expression) IN Updates DO
      BEGIN
         V:=Evaluate(Expression);
         Assign(Statecomponent,V);
      END;
END;

Assign(Statecomponent,Value)
BEGIN
   Get_old_value(Statecomponent,Oldvalue);
   Delete_structure(Oldvalue);
   Create_structure(Value,Structure);
   Store_new_value(Statecomponent,Structure);
END;

Figure 10.12: Updates of variables and class instances

update(x:account equal(x.account_id,123),[(x.balance,x.balance+deposit)])

Here, the update path is account.balance, since it is the balance property of the account
class which is updated.

Update_holds is similar to Delete_holds. It calls a procedure Performupdates for each
instantiation of the free variables in Freeformula which makes it true. An update is made by
first evaluating the update expression, and then storing the resulting value for the specified
state component. An assignment (Assign) is made by deleting the fact structure for the
old property value (Delete_structure), and create a new fact structure for the new value
(Create_structure). Finally, the link is made between the state component and the new fact
structure (Store_new_value).
For some updates, temporary data structures are created. If so, these will have to be removed, and the new values have to be stored for the updated state components.

Checking State Laws

As described in Chapter 7, a statelaw is specified through its name, a checking situation, a sentence, and an action to take when the sentence does not hold. The checking situation is used to determine which state laws that are to be evaluated. It consists of an operation (insert, update, delete), a class name for insertions and deletions, and an update path for updates. During execution, operations are analyzed and matched with specified state laws. If the checking situation of a state law matches with the operation, the corresponding law is checked upon completion of the dynamic law to which the operation belongs. By doing this, we avoid having to check every state law in every state, which is very inefficient for large models with many state laws. We also achieve that the operations of a law are performed as an atomic unit, allowing temporary violations of state laws within a simple dynamic law.

Generation of Interpreter Procedures from Sublaw Relationship Specifications

Interpreter procedures can be automatically generated from the specification of sublaw relationships. The decision tables in Figure 10.13 illustrate the main principles used. Each table corresponds to a part of the apply procedure outlined above. For each table, all but the last row correspond to different values of the functions specified for each sublaw relationship. The last row gives the value for the part of the apply procedure in question, for each relevant combination of function specifications. '-' indicates a situation where the specification makes no difference on the outcome of the decision.

For some sublaw relationship specifications, certain parts may be omitted, indicated by '---' in the table entries. To produce an apply procedure, parts are selected by finding the appropriate entries, one entry from each table. We have not displayed what the entries actually are, but we have indicated the generation of a procedure for the SEQ relationship in Figure 10.14. For this procedure, the candidate is identified in the head of the rule, so it is not necessary to have a separate procedure to identify candidates. Furthermore, the latter four parts of the generic apply procedure, can be omitted.

The generation of procedures can be specified and implemented using TRL, but we felt that the principles are best illustrated through the decision tables.

A very similar approach can be used for generation of terminate procedures, although we will not go into this here. The decision tables are considerably smaller in this case.
### 10.3. An ECML Interpreter

#### Rule head table:

<table>
<thead>
<tr>
<th>cand</th>
<th>first</th>
<th>all</th>
<th>else</th>
</tr>
</thead>
<tbody>
<tr>
<td>Head</td>
<td>h1</td>
<td>h2</td>
<td>h3</td>
</tr>
</tbody>
</table>

#### Find_candidates table:

<table>
<thead>
<tr>
<th>cand</th>
<th>first</th>
<th>all</th>
<th>onerandom</th>
<th>onerandom</th>
<th>user</th>
<th>user</th>
<th>condition</th>
<th>condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>notcand in addC</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Find candidates</td>
<td>---</td>
<td>---</td>
<td>fc1</td>
<td>fc2</td>
<td>fc3</td>
<td>fc4</td>
<td>fc5</td>
<td>fc6</td>
</tr>
</tbody>
</table>

#### Find_prec_holds table:

<table>
<thead>
<tr>
<th>cand</th>
<th>first, onerandom</th>
<th>first, onerandom</th>
<th>first, onerandom</th>
<th>first, onerandom</th>
<th>first, onerandom</th>
<th>first, onerandom</th>
<th>first, onerandom</th>
<th>first, onerandom</th>
<th>first, onerandom</th>
</tr>
</thead>
<tbody>
<tr>
<td>facts in termc</td>
<td>none</td>
<td>none</td>
<td>executes</td>
<td>executes</td>
<td>executed</td>
<td>executed</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>notprec in addC</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>select</td>
<td>fph1</td>
<td>fph2</td>
<td>fph3</td>
<td>fph4</td>
<td>fph5</td>
<td>fph6</td>
<td>fph7</td>
<td>fph8</td>
<td></td>
</tr>
<tr>
<td>Find_prec_holds</td>
<td>fph9</td>
<td>fph10</td>
<td>fph11</td>
<td>fph12</td>
<td>fph13</td>
<td>fph14</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Select_laws table:

<table>
<thead>
<tr>
<th>cand</th>
<th>first, onerandom</th>
<th>all</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
</tr>
</thead>
<tbody>
<tr>
<td>select</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Select_laws</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

#### Assert_facts table:

<table>
<thead>
<tr>
<th>cand</th>
<th>first, onerandom</th>
<th>all</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
</tr>
</thead>
<tbody>
<tr>
<td>select</td>
<td>-</td>
<td>-</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Assert_facts</td>
<td>---</td>
<td>---</td>
<td>af1</td>
<td>af1</td>
<td>af1</td>
<td>af1</td>
<td>af1</td>
<td>af1</td>
<td>af1</td>
<td>af1</td>
<td>af1</td>
<td>af1</td>
</tr>
</tbody>
</table>

#### AddC table:

<table>
<thead>
<tr>
<th>cand</th>
<th>first, onerandom</th>
<th>all</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
</tr>
</thead>
<tbody>
<tr>
<td>addC</td>
<td>-</td>
<td>none</td>
<td>nc</td>
<td>np</td>
<td>nc</td>
<td>np</td>
<td>nc</td>
<td>np</td>
<td>nc</td>
<td>np</td>
<td>nc</td>
<td>np</td>
</tr>
<tr>
<td>AddC</td>
<td>---</td>
<td>---</td>
<td>ac1</td>
<td>ac2</td>
<td>ac3</td>
<td>ac4</td>
<td>ac5</td>
<td>ac6</td>
<td>ac7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Apply_all table:

<table>
<thead>
<tr>
<th>cand</th>
<th>first, onerandom</th>
<th>all</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
<th>else</th>
</tr>
</thead>
<tbody>
<tr>
<td>select</td>
<td>-</td>
<td>-</td>
<td>first, onerandom</td>
<td>first, onerandom</td>
<td>ap1</td>
<td>ap1</td>
<td>ap1</td>
<td>ap1</td>
<td>ap1</td>
<td>ap1</td>
<td>ap1</td>
<td>ap1</td>
</tr>
<tr>
<td>Apply_all</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

Figure 10.13: Decision tables for each of the parts of the procedure apply
Execution Techniques

Here, we review some of the execution techniques described in Chapter 4, and show how they can easily be accommodated by modifying the core algorithms of the interpreter.

Step-by-step execution  By modifying the execution cycle in Figure 10.2, this feature can be offered. If the algorithm is extended with a call to the Interrupt procedure given in Figure 10.15 at the end of the loop, the user can interrupt the execution in every state. A set of actions are offered to the user, e.g. to give queries about the current state, to ask questions to the explanation generator, or to query the execution trace directly.

Breakpoints  Breakpoints can be supported by specifying a breakpoint set, i.e. a set of dynamic laws. When the interpreter executes one of the laws in the set, it should allow the user to interrupt with actions similar to those offered in step-by-step execution. In each execution cycle, the complex and simple laws applied are compared with the breakpoint set. If the intersection of the two is nonempty, a break is made. If the execution cycle is added with a call to the procedure Breakpoint given in Figure 10.16, the execution technique is supported.

Scenarios  Scenarios or programmed executions can be offered by going through a sequence of specified external events, one event per pass of the main loop in Figure 10.1. The presentation of available events and the reading of an event would be replaced by a single statement: Get_next_event( Event). The operations of the events in the sequence should be specified in such a way that no user interaction is necessary.
Interrupt
BEGIN
    WRITE('Go to next state Y/N?');
    READ(Go)
    WHILE Go≠Y DO
        BEGIN
            Offer_actions;
            READ(Action);
            Perform_action(Action);
            WRITE('Go to next state Y/N?');
            READ(Go)
        END;
    END;
END

Figure 10.15: Step-by-step execution in the ECML interpreter

Breakpoint(New_to_£)
BEGIN
    Breaks:= {x | x∈New_to_£∩Breakpoints}
    IF Breaks≠ Ø THEN
        BEGIN
            WRITE(Breaks);
            Interrupt;
        END;
    END;
END

Figure 10.16: Breakpoints in the ECML interpreter

Two Forms of Event Reporting  In the algorithm for the main loop, we have assumed that external events can only be reported in stable states. However, as we discussed in Chapter 7, there are situations where it is desirable to be able to report events even in unstable states, e.g. in executions of Petri-nets. This can be supported by having an action in the Interrupt procedure which allows external events to be reported, i.e. it is one of the actions presented to the user by Offer_actions.

Limitations

The major parts of the interpreter as specified through the algorithms in this chapter, have been implemented. However, some work still remains to be done. This relates to type checking, the user interface, support for complex types, rollback mechanisms, input and output parameters of dynamic laws, execution techniques, and generation of interpreter routines.
Type checking  ECML models are interpreted directly, with no static type checking done beforehand. For the simple types like strings and numbers, PROLOG performs checking at runtime, e.g. it is not possible to add a number to a string. We have not implemented type checking for the more complex types offered in ECML, so in the current implementation, it is possible to e.g. add a string to a set declared to be of type integer.

User interface  The user interface is line-based, and builds directly on the features offered by PROLOG. It would certainly improve the interpreter if a graphical interface is constructed, for reporting events and presenting results of executions.

Support for complex types  Currently, only sets, aggregates, and the simple types are supported by predefined predicates and functions. This means that e.g. properties may be complex, but their types may only involve the set and aggregate type constructors. It is however straightforward to extend the interpreter to support the other types as well.

Rollbacks  We have not considered rollbacks at all. In the current implementation, violated state laws are simply presented to the user. The development of efficient rollback mechanisms is a research topic on its own, and is considered to be further work.

Input and output parameters  We have not yet implemented the passing of parameter values between dynamic laws. For the integration of PPP, this was not required, so we have given less priority to this feature in the first version of the interpreter.

Execution techniques  The algorithms presented in the previous section show the feasibility of supporting various execution techniques. These have however not been included in the current implementation.

Generation of interpreter routines  The automatic generation of interpreter routines which was described above has not been implemented. Rather, we have built general routines which interpret the specifications during execution, and we have also 'hard coded' the sublaw relationships used in PPP models.

Interacting with the Interpreter

As mentioned above, the user interface of the interpreter is very simple. Therefore, rather than to describe a concrete execution session with a presentation of screen dumps, we describe in more general terms what can be done.

First, the model to execute must be loaded, together with the specifications of control structures used, and possibly with some class instances to initialize the model state. Now, the user may execute state changing operations directly, e.g. to insert or update class instances. Any violated state laws are displayed on the screen. At any time, queries may be given to access the model state, and hence the effects of the operations may be inspected.
In order to study the dynamic aspects of the model, the main loop described above is initiated. The external events which may be reported by the user are then presented. The user chooses an event, and all the associated operations are executed. After this, the model starts execution of the dynamic laws. During execution, the interpreter may ask for additional inputs, and it may present outputs to the user. Also, the user is made aware of state laws or postconditions that have been violated. When the execution terminates, i.e. the response to the external event is complete, the user may again inspect the model state through queries to study the effects of the external event and the subsequent execution.

When the interpreter is coupled to the tracing component and the explanation generator, the user may give queries to the query handler, and ask questions to the explanation generator, in order to further understand the execution.

## 10.4 A TRQL Interpreter

We will only describe some of the algorithms of the interpreter, since they are relatively straightforwardly developed. As will be seen, the algorithmic structures include logical specifications of the data which are to be returned for the different queries. Note that in the algorithms, we refer to tuples of the trace relations by $x \in \text{relation}$, and to the different attributes of the relation through the use of the dot-notation. The relations and attributes were defined as shown below:

- $\text{dynamic\_law}(\text{Id}, \text{From}, \text{To}, \text{Prec}, \text{Inputs}, \text{Outputs})$
- $\text{external\_event}(\text{Id}, \text{From}, \text{To})$
- $\text{state\_law}(\text{Id}, \text{From}, \text{To}, \text{Ref})$
- $\text{state\_change}(\text{Id}, \text{To}, \text{Ref}, \text{Statecomp}, \text{Value}, \text{Type})$

### Interpreting Queries in the Law Application View

Many queries may have subqueries, which have to be evaluated first. The same subquery may yield different results, depending on the context it is given in. The algorithm for interpretation of queries in the law application view is given in Figure 10.17. Evaluation of subqueries is presented in Figure 10.18. Generally, a subquery here returns a set of law identifiers.

If all applications of the laws are wanted, then for each law, applications ending within the specified period are retrieved. If the law is a simple law, then all corresponding state\_change tuples are retrieved as well.

If only the first application is desired, then the application following the beginning of the specified period is returned. When a navigation through a temporally ordered list of law applications is wanted, all applications within the specified period are first found and sorted, and then stored in a temporary relation $\text{dynamic\_law\_current}(\text{Id}, \text{Laws}, \text{Current})$. It
Query(dynami_iow.(Occurrence, Lawref), Beginning, End, Return)
BEGIN
Laws := Subquery_law_application(Lawref, Beginning, End)
IF Occurrence = all THEN
  Return := {(x, y) | x ∈ Laws ∧
              y = \{(z, u) | z ∈ dynamic_law ∧ z.id = x ∧ z.to > Beginning ∧ z.to ≤ End ∧
              u = \{v | v ∈ state_change ∧ v.id = z.id ∧ v.to = z.to\}\}\}
ELSE IF Occurrence = first THEN
  Return := {(x, y, z) | x ∈ Laws ∧ y ∈ dynamic_law ∧ y.id = x ∧ y.to > Beginning ∧ y.to ≤ End ∧
            ∀ u ∈ dynamic_law (u.id = x ∧ u.id = y.id ∧ u.to < y.to ⇒ u.to < Beginning),
            z = \{v | v ∈ state_change ∧ v.id = x ∧ v.to = y.to\}\}
ELSE IF Occurrence = last THEN % Similar to 'first'
ELSE IF Occurrence = next THEN
  IF ∃ x ∈ dynamic_law.current(x.id = y ∧ Laws = \{y\}) THEN
    BEGIN
      Return := {(u, v) | u ∈ dynamic_law.current ∧ x.id = y ∧ Laws = \{y\}
                  ∧ x.laws ∧ position(u) = x.current + 1 ∧
                  v = \{z | z ∈ state_change ∧ z.id = u.id ∧ z.to = u.to\}\}
      Update(x ∈ dynamic_law.current(x.id = y ∧ Laws = \{y\}), x.current := x.current + 1)
    END
  END ELSE
  BEGIN
    Query(dynamic_law(all, Lawref), Beginning, End, R2)
    Sorted_laws := {x | (u, v) ∈ R2 ∧ (x, y) ∈ v ∧
                    ∀ z ∈ Sorted_laws (z.to > x.to ⇒ position(z) > position(x))}
    Law := {x | (x, y) ∈ R2}
    dynamic_law.current := dynamic_law.current ∪ (Law, Sorted_laws, 2)
    Return := \{(x, u) | x ∈ Sorted_laws ∧ position(x) = 2 ∧
                u = \{y | y ∈ state_change ∧ y.id = x.id ∧ y.to = x.to\}\}\}
  END
ELSE IF Occurrence = previous % Similar to 'next'
END

Figure 10.17: Retrieving information about applied dynamic laws

is assumed that only one law is queried at a time. Current is a cursor that is used to navigate
backwards and forwards in the list of retrieved applications. The first time a 'next' is used
in a query, it is assumed that a previous query with 'first' has been issued. The temporary
relation is built up, and the second tuple in the sorted list is returned.

Interpreting Queries in the Refers View

Also here, a subquery is performed in order to find applications of dynamic laws or oc-
currences of external events. The algorithm is given in Figure 10.19. In the backwards
direction, it is searched for dynamic laws or external events which changed the state of
Subquery: lawapplication(Lawref, Beginning, End)
BEGIN
  IF Lawref=change(O,S,C) THEN
  BEGIN
    Query(change(O,S,C), Beginning, End, Result)
    RETURN \{ x | (u,v) \in Result \land (y,z) \in v \land x = y.id \}
  END
  ELSE IF Lawref=refer(L,D) THEN
  BEGIN
    Query(refer(L,D), Beginning, End, Result)
    RETURN \{ x | (u,v,w) \in Result \land (y,z) \in w \land x = y.id \}
  END
  ELSE IF Lawref=super(L,O) THEN
  BEGIN
    Query(super(L,O), Beginning, End, Result)
    RETURN \{ x | (u,v,w) \in Result \land y \in w \land x = y.id \}
  END
  ELSE RETURN \{ Lawref \}
END

Figure 10.18: A function which performs subqueries of law application queries

state components referenced either in the precondition or in the computation of new values. In the forwards direction, it is searched for dynamic laws which refer to the state components changed by a law application or an external event.

Interpreting Queries in the State Component View

Parts of the algorithm is given in Figure 10.20. A subquery first determines the state components of interest. The principles used are very much the same as those for the law application view. For the case that all changes within a period are to be retrieved, all tuples of state_change which represent a change to the specified component are retrieved, together with the corresponding tuples of dynamic_law or external_event. In order to be retrieved, the condition given with the query must hold, and the state component in the query must match with those of the state_change tuples. If x is the state component from the query, and y is the state component from a tuple of state_change, they can match in the following way:

\[
matches(x, y) \iff x = y \lor x = a \land y = a.b \lor x = a.b \land y = a \lor \text{instance}(y, x)
\]

In the latter disjunct, x is a class name, and y is an instance identifier.

The only problem with evaluating queries in the state component view, concerns the identification of state components being class instances. We rely on the capabilities of the
Query(refers(Law,Direction),Beginning,End,Return)
BEGIN
Applications:=Subquery_refers(Law,Beginning,End)
IF Direction=backwards THEN
Return:={ (l,c,d)|(l,c)\in Applications \land d=\{ x|((u,v)\in l.prec \land z\in c \land (u,v)\in z.ref) \land 
Query(change(last,u,true),l,l.from,x) \}}
ELSE
Return:={ (l,c,d)|(l,c)\in Applications \land 
d=\{ (x,y)|\exists c \land u=z.statecomp \land v=z.value \land 
x\in dynamic.law \land (u,v)\in x.prec \land \exists s(s\in state.change \land s.id=x.id \land s.to=x.to \land 
(u,v)\in s.ref) \land x.from>l_.to \land \neg \exists l'\exists w'(l'\in dynamic.law \land w'\in state.change \land 
w'.id=l'.id \land w'.to=l'.to \land w'.statecomp=u l'.to<x.from \land l'.to>l_.to) \land 
y=(w|w\in state.change \land w.id=x.id \land w.to=x.to) \}}
END

Figure 10.19: An algorithm for answering queries in the refers view

Query(change(Occurrence,Statecompref,Condition),Beginning,End,Return)
BEGIN
Statecomps:=Subquery_statecomponent(Statecompref,Beginning,End)
IF Occurrence=all THEN
Return:={ (s,c)|s\in Statecomps \land s=(x,y)|\exists (x) \in dynamic.law \land x\in external.event \land 
y\in state.change \land y.to=x.to \land \exists (s,y.statecomp) \land 
y.to\geq Beginning \land y.to\leq End \land holds.condition(s,Condition,x) \}}
ELSE IF Occurrence=first THEN % Similar as with the law application view
Rest of algorithm builds on already illustrated principles
END

Figure 10.20: An algorithm for answering queries in the state component view

ECML interpreter to do this. Otherwise, we would have to include more information about changed state components in the execution traces, i.e. information about the property values which were not changed. If a class instance has been deleted, so that it does not exist in the current state, the trace must be inspected anyhow. For the delete operation, all property values of the class instance deleted are stored together with 'delete' as the operation type. Hence, if we want to know about changes made to a component prior to its deletion, we first have to find its identifier from information stored about the delete operation. After that, information about prior changes is done in the same way as for other queries.
Query(super(Occurrence, Law), Beginning, End, Return)
BEGIN
    IF Law = system.law THEN Return := \{\} ELSE
    BEGIN
        Applications := Subquery.context(Law, Beginning, End)
        IF Occurrence = super THEN
            Return := \{(x, y, z) | (x, y) \in Applications \land super(x.id, Super) \land z \in dynamic.law \land z.id = Super \land z.from \leq x.from \land z.to \geq x.to\}
        ELSE
            Return := \{(x, y, z) | (x, y) \in Applications \land super(x.id, Super) \land u \in dynamic.law \land u.id = Super \land u.from \leq x.from \land u.to \geq x.to \land Query(super(all, u), Beginning, End, R1) \land z = u \lor R1\}
        END
    END
END

Figure 10.21: An algorithm for answering queries from the law context view

Interpreting Queries in the Law Context View

We have given the algorithm for finding the application of a superlaw from a specified law application in Figure 10.21. The subquery identifies law applications. The conceptual model or execution trace is inspected to find the identifier of the superlaw. Then the application of the superlaw is retrieved, i.e. the application which 'life-span' subsumes the one of the law application in question. Note that when all superlaw applications are to be retrieved, a recursive call is made to the procedure, until it reaches the system law.

Interpreting Queries in Other Views

Queries in the external event view and the state law view are interpreted very much the same way as in the law application view.

The state view query can be interpreted through the use of other queries. As an example, if information about the last state where a particular law was initiated is to be retrieved, a law application query is made first. Then the to state number is selected, and all relevant information about the state with this state number is returned. For the evaluation of expressions in historical states, the only problem is to find the value of the state components referenced in that state. This is done by finding the last change of the component prior to the state. If no such change is found, the initial value of the component is used.
Limitations

The limitations of the current prototype are threefold: The form of the query language, the interface with the model executor with storage of changes to class instances and input and output parameters of dynamic laws, and the user interface to the query handler.

The queries must be given in the form of predicates, as indicated in the parameters of the query procedures above. This means that queries as presented in Chapter 9 can not be interpreted in the current prototype. This limitation is not very serious, as it is relatively simple to construct an interpreter on top of the one developed so far, which turns queries in TRQL into the form needed.

The interpreter was developed together with an early version of the ECML interpreter, at a time when only named variables could be manipulated. The introduction of classes and class instances does not require substantial modifications. One problem is to report on state components referenced in the evaluation of quantified formulas, or in the computation of new property values of class instances. Another problem discussed briefly above, is to find identifiers of class instances through descriptive reference. We can rely on the ECML interpreter to do this, but the necessary integration of the two components to achieve this has not been implemented yet.

We have so far not considered how to create a good user interface to the query handler. In the prototype, the queries are given as predicates, and the retrieved relations are listed straightforwardly.
Chapter 11
Integration with PPP

In this chapter we discuss the integration of the translation and execution components with the PPP environment. We first discuss briefly the data and control aspects of such an integration. Then we focus on the translation from PPP models to ECML, for the three languages PhM, PrM, and PLD in turn. For each language, we present the main translation principles, and we illustrate them by examples of translation rules expressed in TRL.

11.1 Introduction

In order to illustrate the feasibility of our approach, we will present the integration of the translation and execution components with the PPP environment. The architecture presented in Chapter 6 shows how the components are meant to be integrated with a general CASE environment, and with PPP in particular. As described in the previous chapter, the prototypes of the two components have been developed with a smooth integration with PPP in mind, being based on the same technical platform.

Figure 11.1 shows the control and data integration aspects when the two components are integrated with the modeling editors in PPP. Both components may be called from the editors. The translator accesses both the PPP models, the specified translation rules, and the generated ECML models. The executor accesses the ECML models, and the data structures it builds up during the execution of a model. Note that even if not shown in the figure, these data can be stored persistently if desired after an execution session has finished.

In this chapter, we focus on translations from PPP models to ECML models. This is most central to the integration, and it will illustrate the advantages of using TRL and ECML to obtain executable conceptual models. In the examples, queries to the PPP models are based on the PPP repository schema, which for most constructs have a direct correspondence with the PPP metamodel presented in Chapter 2. The repository schema is included in the appendix. Expressions of PLD constructs follow the syntax given in Chapter 2.
As will be shown, for some constructs in the PPP language, there are more than one suitable alternative for representation in ECML. For other constructs, the choice of representation is rather obvious. In any case, the main objective is to find a translation which preserves the semantics of the original language.

An early version of the translation rules is found in [55]. A complete list of translation rules is found in the appendix.

### 11.2 Translation from PhM

We will describe the translation from PhM to ECML in three parts: 1) Translation of entity classes and attributes, 2) translation of relationships with cardinalities and coverage constraints, and 3) translation of subclasses. Figure 11.2 shows parts of the original PhM model for the bank system, and most of its representation in ECML.

#### Entity Classes and Attributes

The main principle here is to create a class in ECML for each entity class, and express attributes as properties (roles) of classes.

Identifier attributes (id) are translated to a role of the same type as the type of the attribute. In addition, a uniqueness constraint must be expressed by means of a state law in ECML. In Figure 11.2, this has been shown for the identifier custno of the class customer. The translation rule can be expressed as follows:
In the query (source model pattern), link corresponds to the connection between an entity class and an attribute. The From and To fields are identifiers of the involved objects, and the category 201 means that the attribute is an identifier. The name of the entity class is accessed through the variable Class, and the type and name of the attribute are accessed through the variables Type and Attr. The relations from the repository schema are described in the appendix.

Ordinary attributes (att) of entity classes are translated in exactly the same way, except that no state law is generated. In the example model, the attribute salary is translated to a role in the class person_customer. Set-valued attributes (rep) are translated similarly, except
that a new type is generated for the role of the class. This type is a set of the same type as the attribute which is translated. Finally, quality attributes are represented as roles of variables, one variable for each entity class which has such attributes. So, if for instance the class account has a quality attribute `average_balance` of type real, this is represented as follows:

\[
\text{type}(\text{account} \_ \text{quality}, \text{role}(\text{average} \_ \text{balance}, \text{real}))
\]
\[
\text{variable}(\text{account} \_ \text{quality}, \text{account})
\]

Hence, to access the value of the quality attribute, `account.average_balance` can be used. Note that this choice of representing qualities could have been made in other ways, e.g. by generating a special class for quality attributes.

### Relationships

Here, we only consider the translation of binary relationships. In short, information-bearing relationships are translated to classes, while non-information-bearing relationships are represented as properties of the involved entity classes. Coverages of relationships are represented as state laws. State laws can also be added for referential constraints.

The translation of the bank model illustrates how the relationship `takes_up` is represented as a property of the class `customer`. Since the cardinality is 1-N, the property is set-valued. We also see that the referential constraint from `customer` to `loan` is represented as a state law. The translation rule for this binary relationship is given below:

<table>
<thead>
<tr>
<th>name</th>
<th>xp_ny_rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal_achieved</td>
<td>binary_relationship</td>
</tr>
</tbody>
</table>
| source_model  | (link(L1,E1,R,C1,..),link(L2,E2,R,C2,..),not(E1=E2),
|               | entityclass(E1,....,Class1),entityclass(E2,....,Class2),
|               | relationship(R,....,Rel),not(link(.,R,....)),
|               | not((link(L3,E3,R,....),not(E1=E3),not(E2=E3))),
|               | or(C1=303,C1=304),or(C2=302,C2=304)) |
| precondition | true          |
| generate     | [class(Class1.role(Rel,Rel)),type(Rel,set(Class2)),
|               | statelaw(Sname,insert,Class1, 
|               | ∀X:Class1\[Y∈X,RelE2:Class2(Z=Y),report),
|               | statelaw(Sname,update,Class1.Rel, 
|               | ∀X:Class1\[Y∈X,RelE2:Class2(Z=Y),report]) |
| subgoals     | combine_texts(["Referential constraint from \',Class1,'to',Class2,\',Sname]

The source model pattern specifies that there are two links from two distinct entity classes to the same relationship class, that this relationship does not have an attribute, that there exist no third entity class involved in the relationship, that the cardinality is either 1-N or N-M, and the coverage is partial from the first class to the second.
If the relationship is information-bearing, it is translated to a class, with the attributes translated to the properties of this class. The connection between the relationship and the involved entity classes must be established, either by having references to the entity classes as properties of the class representing the relationship, or the other way around.

Subclasses

In the current PPP environment, a simple subclassing construct is offered, as described in Chapter 2. It is sometimes desirable to express additional constraints on the subclasses, e.g. to state that two subclasses are disjoint, or that they together compose the superclass. As an example, in the bank model one can assume that the two subclasses of customer partition the class, i.e. they are disjoint and composing. Such constraints can be expressed as state laws in ECML. The translation rule below specifies translation of disjoint subclasses to ECML.

```
[ name disjointsubclass_rule
goal_achieved disjointsubclass
source_model (link(_,From,To,501,_),
etntityclass(From,_____,Class1),
link(_,To2,Class2,402,_),
etntityclass(To2,_____,Class2),
subgroup(To,Group,_____,))
precondition true
generate [isa(Class1,Class2),
statelaw(Sname,insert,Class1,∀X:Class1(Disjoint),report)]
subgoals [forall(Z,Group,t_goal(disjoint_formula,Z,[Class1,X]_),Notexists),
distribute_function(and(Notexists),Disjoint),
combine_texts('Subclass disjointness for class',Class1,Sname)]
```

The facts referenced here correspond to a proposed extension of the PPP metamodel, to include disjoint and composing subclasses. In order to do so, subclasses are grouped (subgroup in the query), and the type of link between the individual class and the group determines the type of subclass constraints. A subgoal is issued to produce the disjointness constraint. It goes through all other subclasses in the same group, and formulates that a class instance can not be an instance of any of the other subclasses. Since the logical and is defined for two arguments only in ECML, the resulting formula must have this connective distributed (the call to distribute_function).

11.3 Translation from PrM

In the following, we go through each of the constructs in PrM, showing how they can be translated to ECML while preserving their semantics.
Processes

Processes are translated to complex dynamic laws in ECML, and the resulting law structure in ECML reflects the process decompositions in PrM models. The system law is created as having all laws corresponding to processes in the top level PrM model as sublaws. An undecomposed process is described through a PLD model, and its translation will be discussed in the next section.

In Chapter 7, we described the sublaw relationships in PrM as VCS - vague control structure. A process is activated spontaneously if it is not already flows. The triggering condition is translated to the precondition of the law corresponding to the process. When a process terminates, it may be executed again. When no processes in a decomposition executes, the superprocess terminates. The full specification of VCS was given in Chapter 7.

In the generation of prototypes from PPP models, so far only the lowest level processes which are described by PLD models have been used in translations. For instance, in the Ada generator, a task is created for each undecomposed process. Here, we propose to reflect the actual process hierarchy in the generated ECML model for mainly two reasons: 1) The hierarchy is preserved, and can be studied in its own right by executing the hierarchical model, and 2) It aids in a more efficient search for applicable laws. Rather than searching through all undecomposed processes in each execution cycle, it is enough to search through those which have superprocesses executing.

The rule for translating processes to dynamic laws is given below:

```
name
goal_achieved
source_model
precondition
generate
subgoals
```

```
processrule
process
(process(Pid,Decomposition,...),
in_port_table(Pid,Nodes,...),
first(Nodes,Node),rest(Nodes,[]),
prm_diagram(Decomposition,...,Pid,Subprocesses,...))
true
[dynamic_law(Pid,Prec,sublaws(vcs,Subprocesses),true,[],[],[])
[t_goal(process,precondition,[Processid,Node],[],Prec)]
```

The translation of triggering conditions to preconditions expressed in ECML is done by traversing the hierarchical port structure in a way similar to what was done in Chapter 8 for the generation of initial PLD models. The leaves of the structure correspond to input flows, and the generated condition at this level is the ECML representation of the condition 'there must exist unread data on this flow'. Figure 11.3 shows the translation of the process P1: Process transaction in the bank model.
Flows

As noted in Chapter 7, we have two alternative translations of data flows. They can be translated to both classes and variables in ECML. In the latter case, the variable is an aggregated structure which includes all data types which are connected to the flow, and an additional 'read-flag' which is updated by the reading and writing processes. Here, we assume that the former approach is taken, i.e. data flows are to be represented as classes.

Flows are translated differently depending on their sources and destinations. Triggering flows from agents, flows between processes, and flows to and from timers, are translated to classes in ECML. The types connected to these flows are types of generated properties of each flow class. The sequential order of the connected types is exploited to generate numbered roles.

Other flows do not need to be translated, because in this translation, these are managed by operations associated with receive and send constructs in PLD's. For instance, it is assumed that flows to and from stores correspond to direct manipulation of databases. Although at this stage these are not implemented in PPP, such operations are likely to be added to the receive and send constructs. For this direct manipulation, we do not need the data structures for the flows. Another example is output of flows to agents. In the translation we give here, this is implemented as display of data directly on the screen.

Agents

Triggering flows from agents to processes correspond to external events in ECML. During execution, they are presented to the user by name. When an event is chosen, its associated operations are performed. In the translation from PrM to ECML, there is a single insert operation, which allows a class instance to be input from the keyboard. The translation rule is given below.
The query specifies a triggering flow originating from an agent. Note that other inputs from agents than triggering flows are taken care of in the translation of receive constructs in PLD's.

Timers

The current implementation of PPP does not have precise constructs for dealing with timers, i.e. with clocks and delays. In [65], Kroghstie exploits the temporal capabilities of the TEMPORA rule manager to implement timers. Here, we sketch a translation of timers to ECML.

First of all, timers are based on the availability of a global clock. A timer acting as a clock will refer to the 'current time', and decides if it should send an output flow. A timer acting as a delay, will compute the next time to send an output flow from knowledge about the current time.

A very simple global clock can be implemented by using two variables current_time and period. The former holds the time of the global clock, and is updated in every system state. The latter specifies the interval for which a model is to be executed. A global clock is then implemented as a dynamic law which is placed as a sublaw of the system law, in its VCS structure. It has the following form:

```plaintext
dynamic_law(clock,current_time<period,
    operations([update(true,[(current_time,add(current_time,1)))]),true,[],[],[])
```

With this law, the current time will be updated in each state, as long as the execution period is not exceeded.

Timers acting as clocks can be implemented by having a variable next_signal which is set either through an incoming flow from a process, or by the clock itself when periodic clock signals are to be sent. The dynamic aspect of a clock is represented by dynamic laws which can set and reset the variable on arrival of input flows, and a dynamic law which continuously checks the current global time against the variable next_signal, and sends out a flow when they match.

Timers acting as delays have an input flow specifying the length of the delay, or the delay is fixed. From the delay, the next time to produce an output flow is determined. Also here, a next_signal variable can be used for the same purpose as with clocks.
In connection with the global clock, one can also hold the current time in different time units, and so the timers can refer to these units as desired. For instance, in the bank model, there is a timer which sends an output flow each first day of a month. This timer would most naturally refer to time in terms of days or months.

Resources

In PrM, a resource is a number of similar items which may be occupied or not. A flow from a resource to a process means that the process needs a specified number of items in order to execute. A flow from a process to a resource denotes the release of held resources. In the current PPP prototype, generation of code from resources is not supported. It is however easy to add the resource construct, and to translate it into constructs in ECML. The simplest way to support resources is to use a variable available to hold the current number of available items of the resource. This variable can be directly updated by processes when they occupy and release items.

Stores

Stores are connected to scenarios in the PPP metamodel presented in Chapter 2. The scenarios can be regarded as views of the complete PhM model. This means that individual stores do not need an explicit representation in ECML. Rather, the collection of stores is represented by the complete PhM model.

11.4 Translation from PLD

In Chapter 8, we presented an example translation from process ports to initial PLD models. The major problem there was to go from a hierarchical port structure to a flat graph structure in the generated PLD. When translating from PLD models to ECML, we face the opposite problem.

We start by explaining the translation of undecomposed processes and local variables. Then we go into the translation of the different constructs in PLD. Note that in the translation rules given below, all PLD constructs have the same representation:

\[
\text{pld}((\text{Id}, \text{Process}, \text{Type}, \text{From}, \text{To}, \text{Right}, \text{Xpos}, \text{Ypos}, \text{Height}, \text{Contents}))
\]

Here, Contents differ for the various constructs, as will be explained in connection with the translations below. The other parts should be self-explaining.

Figure 11.4 shows the main principles in the translations from PLD structures to ECML law structures. It will be referenced in the descriptions below.
Undecomposed Processes

Processes which are not decomposed and automated, are described through a PLD model. Such a process may have a duration over several states, while communicating with its environment and producing results. A PLD model is regarded as a hierarchical structure of statement blocks. At the topmost level, the model is a sequence of statements and blocks, and hence the complex law corresponding to the undecomposed process has a SEQ structure with its sublaws. The precondition of such a process is found in exactly the same way as for decomposed processes. The SEQ relationship was also described in Chapter 7. In Figure 11.4, the topmost level consists of the first receive construct, and the two selection blocks below it. Note how identifiers of constructs from the PLD model are used as identifiers of dynamic laws in the law structure.

Local Variables

An undecomposed process will have local variables. Here, we assume for simplicity that their types are the simple datatypes found in PhM. In order to assure locality of the variables, they become roles in an aggregated structure, one structure per process. Both a type and a variable corresponding to the aggregate is generated. As an example, if the local variables a of type ta and b of type tb belong to the process p, a type p..data is created as the aggregation of two components with roles a and b. In addition, a variable p is created of type p..data. Now, we can refer to the value of the local variable a by p.a.

Assignments

An assignment is translated to a law with an update operation in ECML. As the assignment expression is stored as a text string, it has to be parsed before translation rules working on the parse tree are used. The translation rule for assignments is given below.
11.4. Translation from PLD

name assignmentrule
goal_achieved assignment
source_model Id
precondition (pld(Id,Process,assign...,To,...,assign(Var,Expr)),
context []
return Sequenceids
generate [dynamic_law(Id, true,
operations(update(true, [(Pname, Var, NE)])), true, [], [], []]
subgoals [parse(Expr, Tree), l_goal(expression, Tree, Process, NE),
   l_goal(,...,To,[], Restsequence), combine_lists([(Id, Restsequence), Sequenceids])]

The identifier of an instance of a PLD construct is received as input. In the update operation, the process name is used together with the variable name to refer to the value of the local variable, as indicated above. A parsing function is used to create a parse tree from the expression, and a subgoal to translate this tree to an ECML expression is issued. The principles for this translation are very much the same as for the translation from SQL to ECML which was described in Chapter 8. A subgoal is also issued to translate the rest of the block below the assignment. The result from this translation is then combined with the identifier of the assignent and returned as the result from applying the rule.

Receipts

Receive constructs are translated differently depending on the source of the input flow:

- Receive constructs with triggering input flows from agents, or with inputs from other processes or timers, are translated to simple dynamic laws with precondition that a flow instance exists. Their operations are to update the local variables from the flow instance, and to delete the flow instance afterwards.

- Other inputs from agents are translated to direct updates of the local variables through reading of variable values from the keyboard.

- Input from a resource can be represented as a simple dynamic law which has as its precondition that a sufficient number of items are available, and an operation which decreases the number of available items accordingly.

- Flows from stores are taken care of by direct manipulation or readings from a database. As noted above, this has not so far been considered in PPP.

The translation rule for the first case is given below:
### Sendings

Send constructs are also translated to simple laws. Also here, there are four different cases to consider:

- A flow is sent to an agent. In this case, it is sufficient to display the value of the expressions in the send construct on the screen. Of course, these expressions must be translated to ECML.

- A flow is sent to another process or to a timer. The generated law will have as its precondition that there does not exist a flow instance already. This is a consequence of the current definition of PrM, where flows can not act as buffers. Its operation is an insert of a flow instance. The values of the properties of this instance are found by evaluating the expressions in the send construct.

- A flow sent to a resource simply updates the number of items available in the resource.

- A flow sent to a store corresponds to a change of the database. As mentioned above, this is not considered here.
Note that the result returned from the rule application is built up in the same way as with assignments and receipts.

**Selections**

A selection in PLD corresponds to an EXCSEQ structure as described in Chapter 7. It is translated to a complex law in ECML, with precondition 'true'. It has one sublaw for each of the alternative constructs and blocks to the right of it in the PLD. The precondition of these sublaws are generated in two ways:

- If the conditions in the alternative constructs are 'yes' and hence have receive constructs immediately below them, the precondition is the ECML representation of 'there exist an input flow instance', i.e. the same precondition as for the law generated from the receive construct.

- The condition is a comparison of some kind, which is evaluated to true or false. These conditions are translated to ECML conditions in a similar way as was the case with expressions. If the rightmost alternative has as its expression 'else', then the generated precondition for this sublaw is the negation of the disjunction of the other preconditions. A special case is when there is only one alternative to the right of a selection. Then a 'dummy' law with a precondition equal to the negation of the first alternative condition is created. Its list of operations is empty. In Figure 11.4, we see an example of this in the first selection block.

Each alternative in a selection is again a complex law which corresponds to the block below that alternative construct in the PLD. This block is translated in exactly the same way as the sequence of the overall PLD model. The sublaws of the EXCSEQ structure are combined into a list in the same way as with sequences. A subgoal is issued to translate the constructs to the right of the current alternative construct. When a rule has been applied to this problem, a sequence of law identifiers is returned. This sequence is concatenated with the identifier of the 'current' alternative construct, and returned as the rule for this construct terminates.

**Loops**

A loop structure in PLD is translated to a complex law in ECML. Since every law in a sequence relationship has to be executed, it is not possible to use the REPSEQ relationship described in Chapter 7 directly, with the precondition corresponding to the condition of the loop construct. Rather, two additional laws have to be added, as shown in Figure 11.5. The superlaw of the law corresponding to the loop construct, has precondition 'true', and hence it will always execute when it is encountered in a sequence. If the condition of the loop construct is true, the block may be repeatedly executed, as long as the condition holds. If the precondition does not hold, the law with the null operation is executed, and the loop body is not.
11.5 Object-Orientation, Real-Time Systems, and ECML

Although the main focus of this chapter has been on the integration of the modeling cycle components in PPP, we now briefly consider how ECML would fit in an object-oriented environment, and in an environment supporting the development of real-time, reactive systems.

The Object-Oriented Perspective and ECML

At the outset, ECML does not seem to fit very well with object-orientation. If an object-oriented analysis or design language includes sufficient detail to be executable, it seems to be more natural to translate it to some object-oriented programming language, perhaps to an object-oriented database programming language. Nevertheless, it is still interesting to see whether we can, in principle, support some of the features of object-orientation in ECML.

Basically, the object construct may be represented in ECML through a combination of state components and dynamic laws. Since semantic datamodels are very similar to the static part of object-oriented datamodels, it seems appropriate to represent objects as state components, and object classes as classes of state components. The same generalization relationships exist in both types of datamodels, so inheritance of object attributes is directly supported.

For modeling of the dynamic aspects of objects, methods are often used. A method can be represented as a dynamic law in ECML. It will usually compute some result on request, or update the state of an object. Doing so, it only references attributes of the object to which it belongs, but it may also communicate with other objects via messages. A message identifies a sender and a receiver object, the method to invoke, and any parameters passed to and from the called method. Messages may be represented as state components in ECML, if asynchronous message passing is desired. Then all methods in a system are grouped in a VCS structure. If the call another method is done directly, this corresponds to one dynamic law calling another, i.e. a sublaw relationship exists between the two.

Inheritance of methods can be taken care of when a model is translated to ECML. If an object calls a method that belongs to a superclass, it must be ensured that no other method
with the same name and parameters exist for other objects in the system. The link between an object class and the methods which belongs to this class must be established through appropriate naming, for instance by prefixing method names with class names in the translation to ECML.

From this description, we see that many of the most common features of object-oriented languages can be represented in ECML. However, we certainly see some limitations. It is not possible in ECML to dynamically change the law structure during execution of a model. This makes it difficult to create object instances of the type found in e.g. SIMULA, i.e. that multiple instances of the same object class execute concurrently.

Statecharts and ECML

The lack of temporal logic in ECML makes it difficult to model complex temporal aspects of real-time systems. However, the control aspects of such systems may still be represented in ECML. In order to illustrate this, we now indicate how Harel's Statecharts [52] can be translated to ECML. For simplicity, we consider only a subset of the language.

There are basically three parts of a Statechart which need translation: The states, the events, and the transitions. Ignoring features like default and historical states, we can represent both states and events as boolean variables with the same names in ECML. If the variable representing an event is true, this means that the event has just happened.

Transitions in a Statechart are represented as complex dynamic laws. Their preconditions are taken from the event and condition parts of the transitions. Their sublaws are aggregations of one sublaw for leaving the previous state, one sublaw for entering the new state, one sublaw for executing any associated actions, and one sublaw for 'consuming' the event which lead to the transition. All transitions in the system are placed in a VCS structure, which means that several transitions can take place simultaneously, and that a series of transitions may take place as new internal events are generated.

Corresponding to each state, both complex and simple, two dynamic laws are created in the translation. One is for entering the state, and one is for leaving the state. Interestingly, the sublaw relationships in the generated law structure correspond to the decompositions of states. For instance, leaving a state $u$ which is an XOR decomposition of $s$ and $t$, means leaving either $s$ or $t$, depending on which one of these is active. Hence, the dynamic law leave $u$ substates has two sublaws in an EXC structure. When leaving a state, the variable representing this state must be updated.

Returning to the representation of transitions, which sublaws should be included to leave and enter states? The answer is that one should first find the least common superstate of the state which is left and the state which is entered in the transition. Then include the dynamic law for leaving the substate of this common superstate, which is a superstate of the state at the 'tail' of the transition arrow. Also include the dynamic law for entering the a substate of the common superstate, which is a superstate of the state at the 'head' of the transition.
Let us turn to a concrete example to make these points clearer. Figure 11.6 shows a possible representation of a transition. Here, the precondition of the transition \texttt{u.f,v} (from \texttt{u} to \texttt{v} on event \texttt{f}) is that \texttt{u} and \texttt{f} are both true. It has three sublawa, one for leaving \texttt{u}, one for entering \texttt{v}, and one for consuming the event \texttt{f}. For leaving \texttt{u}, the state must be active. Since it has an XOR decomposition, its sublaw \texttt{leave_substates.u} has an EXC structure of sublawa, one for leaving \texttt{s}, and one for leaving \texttt{t}. Similarly, to enter the state \texttt{v}, which is a simple state, it is only necessary to update the variable \texttt{v} to true.

### 11.6 Concluding Remarks

This chapter has demonstrated the feasibility of using ECML as a target language for execution of conceptual models in PPP. It has further shown the feasibility of using TRL as a language for specifying and implementing these translations, and moreover that the translations are quite easily expressed in TRL. In the translations described, we have omitted certain constructs. First, we have not translated relationships of higher arity. The reason for this that the choice of representation depends on the way such relationships are manipulated and referenced. This brings us to the second omission: Database manipulation. As mentioned above, this may most naturally be connected to expressions in receive and send constructs. A proposal for a revised PhM with an algebra is presented by Mingwei Yang [139], however it is not integrated with the PLD language yet. Finally, we have not considered the use of complex types of data flows and of local variables of processes. With the rich type system in ECML, this should not cause problems.
Part IV

Evaluation and Conclusions

In Chapter 12, our results are compared with respect to the stated requirements, and with respect to existing work.

In Chapter 13, we summarize our contributions, and propose directions for further work.
Chapter 12

Evaluation

We evaluate the work both by comparing the results with the requirements identified in previous chapters, and by comparing it with existing work in the same area. Moreover, we present some experiences with the developed prototype, although they are limited at the present time. Finally, we discuss how our approach can be exploited by the PPP environment and CASE environments in general.

12.1 Evaluation with Respect to Stated Requirements

The requirements to the three languages TRL, ECML, and TRQL, and to the translator, the model executor, and the tracing component, have been presented in chapters 3, 4, 5, and 7. In the following we evaluate our work with respect to these requirements.

TRL and the Translator

TRL can be considered both as a concrete language for translation specification and implementation, and as a framework in which particular repository manipulation languages can be 'plugged' in. In this thesis, we have mostly used a language adapted to the PPP repository, since we then can support translations in PPP, and illustrate the usefulness of TRL on real examples.

We will now go through each of the requirements listed in Chapter 3, and evaluate TRL and its interpreter with respect to them.

Separate specification level A separate specification level is ensured by having the translation rules refer to the metamodels of the source and target languages. We have emphasized that such metamodels may be specified in integrated metalanguages, e.g. a semantic datamodel combined with a variant of BNF. This is particularly the case
when the language is executable, as it then most likely will have expressions being phrase structures.

**General requirements** We believe that the uniform representation of translation knowledge in the form of rules makes translations more easily expressed, and more easily maintained. For the translation tasks needed in the current PPP environment, the concrete TRL is complete.

**Independence from language and task** TRL considered as a concrete language is only adapted to the PPP repository, and not to the particular translation tasks, or to the source and target languages.

**Mixed model representations** In TRL, basic-construct rules always refer to the repository schema, while construct-property rules may refer to parse trees. Hence the two representations are integrated. We have assumed a parsing function of sufficient power to be available. The produced parse tree is then traversed and referenced by the translation rules. Alternatively, we could in many cases have performed the translation during the parsing. One way to achieve this, is to integrate a parser generator like YACC with TRL, but this will only work if no contextual information is needed in the translations. YACC or similar tools could also be used simply to produce the parse tree.

**Predefined semantic actions** These correspond with the predefined functions and predicates offered by TRL. The set of functions and predicates may easily be extended or modified if needed for particular translation tasks, or for other data structures in a repository.

**User interaction** So far, we have not considered this issue in any depth, partly because the translations we have focused on for prototype generation should be fully automated. Involving the user means introducing issues like support for the choice of translation rules, and recording the choices made for later translations. These issues have been outside the scope of our work.

**Automatic implementation** We have developed a simple interpreter for TRL, which has demonstrated the feasibility of TRL, and allowed us to experiment with some real translation problems. As an interpreter, it is necessarily less efficient than a compiled solution. In Chapter 8, we have discussed how a more efficient, procedural implementation can be automatically generated from rules expressed in TRL. As pointed out before, the automatic implementation may be regarded as a translation task in its own right, and hence TRL could be used for this task as well.

**ECML and the Executor**

Also here, we use previously identified requirements to guide the evaluation. These requirements were identified in the beginning of Chapter 7.

**Expressiveness** ECML was derived from a unified metamodel, which again was a result of analysis of existing unified metamodels. We claim that ECML is highly expres-
sive, and being so at a high level of abstraction as required. At the same time, executability most often requires that details must be specified as well.

For the modeling of static system aspects, ECML offers both classes and variables. The properties of classes, as well as variables, can be of arbitrary complex types. Although we have selected some commonly used types to be supported, other types with their associated functions and predicates may easily be added, if needed. State laws are used for modeling of constraints on state components. We have proposed a typed first order logic to specify state laws, and we allow the checking situation to be specified in order to make evaluation more efficient.

Modeling of system dynamics is characterized with a richness of control structures available, and an extensive use of hierarchies. We have found a principle for specifying sublaw relationships, and for automatically generating interpreter routines from these specifications. This means that we can offer a wide range of control structures found in existing languages, as was explained in Chapter 7. In particular, concurrency and non-determinism is supported. Manipulation of state components is achieved by means of operations, which are grouped together and associated with the leaves of the law hierarchies. The operations build on logical specifications of the components to be manipulated, and of the changes to be made. Finally, external events and input operations constitute the input interface with the user. The events have operations associated with them, which are executed when the user selects them during execution.

Having said this, there certainly exist constructs not directly supported by ECML. First, we have not considered temporal logic in connection with ECML. Rather, we treat time as any other property of state components. Second, dynamic constraints are not supported directly, although there exist ways to implement simple dynamic constraints. As an example, for constraints which relate property values in two subsequent states, the property can be duplicated in a new and an old version. For instance, the constraint 'the salary of an employee can never decrease' can be checked by comparing the property oldsalary with newsalary. Note however, that more complex dynamic constraints referring to multiple states require a temporal logic.

Another construct not supported directly, is internal events. In cases where internal events are used to trigger dynamic laws, they can be represented as boolean variables. The updates of these variables are taken care of by dynamic laws which detect the occurrence of the events. Events can also be detected and recorded directly by the laws which actually bring them about.

**Executability** ECML is executable, although the prototype interpreter may not have the performance of a commercial tool. It is possible to separate out different aspects of a model, and still have it executed. For instance, it would be possible to prototype only the parts of models describing static system aspects. It is also possible to perform symbolic executions while abstracting away details of data manipulation, and rather focus on the dynamic aspects. Also, sublaw relationships can be defined which let incompleteness and non-determinism be resolved by random choices and user interaction.
**Styles of expression** Both declarative and procedural styles of expression are offered. Specification of state components, state laws and operations have a declarative style. Dynamics can be specified both declaratively, e.g. by sublaw relationships representing flat rule structures, and procedurally, e.g. through the traditional control structures found in programming languages.

**Explanations** Here, the major requirement was that all dynamic laws can be referenced and traced, so that the causes of state changes can be uniquely identified. Dynamic laws at all levels in the law hierarchy have unique identifiers, so this requirement is satisfied. Also, state laws are identified by name and checking situation.

**Technical considerations** We are concerned with two issues, one of which is efficiency. One way to enhance efficiency both in developing an executable conceptual model and in its execution, is to allow functions to be called which are written in other languages. This would permit and suggest reuse of commonly occurring functions. In ECML, we support this feature, however in the prototype interpreter, functions must be represented as clauses in PROLOG.

Another way to enhance efficiency is to limit the search for applicable dynamic laws through the use of hierarchies. Hierarchies have an important role in ECML, both for the decomposition of complex problems, and for efficiency reasons. A final point to be made about efficiency, is that it should be possible to execute parts of models. In order to achieve this, we would specify translation rules so that the environments of the selected parts of a model are replaced by user interactions, i.e. so that the user would simulate the rest of the system. Inputs originating from other model parts must be given by the user, and outputs produced to these parts are presented to the user.

The second technical consideration is integratability with CASE environments. We have demonstrated this ability through the proposed integration with the PPP environment.

**Other requirements** ECML models are precise, but they can embody non-determinism. The operational semantics of law executions has been defined in Chapter 7, but we have considered the semantics of operations, functions, and predicates to be well-known, and hence we did not consider these.

**TRQL and the Tracing Component**

The requirements to tracing components were identified in Chapter 5. The major goal of our work has been to satisfy the functional requirements.

**Language independence** Execution traces of models from a wide range of languages fit with the trace schema specified. It can certainly be used for tracing of ECML models, and hence it also has the generality of ECML. The tracing component as described in Chapter 9 is however not independent of the conceptual models which are executed, since for some queries, the conceptual model and its execution instances need to be accessed.
Expressiveness The trace schema includes relations for both state changes and for the events which bring about state changes, and their relationships. As noted above, sometimes it is required that the conceptual models are accessed. If this is not desirable or possible, the needed information must be added to the trace, e.g. sublaw relationships. Also, the states of the model executor are not recorded. This means that only the laws actually executed are traced, and not those only evaluated for execution, i.e. for ECML models, the details of candidate selection, evaluations of preconditions etc. Storing this information would increase the size of the trace considerably in many cases. In order to explain why dynamic laws did not apply in a given state, we rather rely on the ability to recompute values of state components in the state. Finally, as discussed previously, we have chosen a level of granularity on the state components stored in the trace. A finer granularity could be supported, but at a higher cost, and with the problems of specifying and retrieving changes to the right state components.

Event reporting Execution tracing does not change the behavior of conceptual models, other than that it makes the execution slower. It is therefore possible to turn off tracing if it is not desired. The temporal ordering of events is maintained by the reported events handler through the use of a global clock which repeatedly updates a state number. Collecting the information to report is the responsibility of the model executor. The current prototype has some limitations in this respect, as was described in Chapter 10.

Information retrieval We have defined a trace query language, which gives the possibility to retrieve information from execution traces through declarative specification. The queries offered are tailored to understand the phenomena occurring during execution of a conceptual model. They can be used as an interface to an explanation generator, to offer comprehensive validation support. They can also be used directly by system developers to provide quick explanations when only trace information is needed. It is possible to tailor queries to the language of the traced model. The very compact trace schema means that some constructs found in modeling languages must be mapped to the same construct in the schema. In order to distinguish these at the time a query is made, knowledge of the mapping is needed. The queries expressed in the CML-specific query language must then be translated to the form accepted by the query handler.

12.2 Comparison with Existing Work

TRL and the Translator

In developing TRL, we were very much inspired by existing work and ideas proposed by others. This is particularly true for the idea of having the specification closely connected to language metamodels, and for using rules to represent translation knowledge. However, we had to modify and combine previous approaches in order to cope with translation problems in CASE environments which support conceptual modeling. The following comparison with the four systems presented in Chapter 3 illustrates this. Note that, as we have
previously compared both these systems and TRL against the requirements identified in Chapter 3, they have already implicitly been compared. Here, we will therefore only focus on some major points.

The well-known parser generator YACC [3] works with grammars specified in a BNF variant, and is not well suited for CASE repositories built on databases. Furthermore, it offers no predefined semantic actions, and contextual information can not be passed on as translations proceed. We have seen many examples that contextual information is useful in the PPP translations.

POPART [135] has for our purposes the same disadvantages as YACC, i.e. no contextual information can be passed between rules, and the translations deal strictly with syntax trees. In POPART, backtracking is used as a means to try out different rules when some rules have failed. In TRL, we have rather used preconditions and contextual information to be able to make a commit to one translation rule. In other words, when a rule is to be applied, enough information is made available to be certain that the rule will succeed in accomplishing its task.

The transformation approach in IPSEN[74] is similar to ours in that both use rules to formulate translations. The rule language in IPSEN is still under development, but rules are said to be expressed as Pattern → Action pairs. The interface of the current system to the source and target documents is however procedural, and the documents are stored in a database. Translations involving mixed representations seem so far not to have been considered. IPSEN makes an emphasis on user involvement during translations, something which we have not considered.

Many of the same comments that were made for IPSEN can be repeated for DAIDA[61]. The representations dealt with are those of DAIDA's knowledge base. Translation of phrase structures is not considered, and is not necessary for the class of source and target languages which can be defined in Telos. As with IPSEN, user interaction plays an important role in translations.

**ECML and the Executor**

We have previously compared the use of executable CML's with high level and very high level programming languages. This comparison applies to ECML as well, so we do not repeat it here. Rather than comparing ECML against a number of existing CML's, we will give a brief comparison with two languages which were developed for similar purposes as ECML. The two languages are the internal representation language in ARIES[62], and Lubars’ GDR[80].

In ARIES, the constructs offered for modeling of static system aspects are quite similar to those of ECML. One difference is the support for multiple inheritance in ARIES. Unless the same properties are inherited from two different superclasses, multiple inheritance could be added to ECML quite easily. We have not considered multiple inheritance in this first version of ECML due to the very limited support for this feature in existing languages.
Another difference is that in ECML, arbitrary complex types of properties and variables can be defined. As seen from a number of semantic datamodels, complex attributes (aggregates, set-valued etc.) can be expressed. Finally, invariants in ARIES corresponding to state laws in ECML, are not specified with a checking situation, which means that every state law must be examined in every system state. This clearly reduces the efficiency of model executions.

For modeling of dynamics, ARIES has a limited set of control structures. Events can be activated spontaneously as their preconditions hold, and events may call other events. Methods have the control structures of traditional programming languages. From this we see that there is little emphasis on hierarchical relationships between events, which is important in ECML. The lack of hierarchies makes it impossible to represent e.g. the process hierarchy of a DFD. On the other hand, preconditions and postconditions of events are formulated similarly as conditions of dynamic laws in ECML. Summing up, ECML provides more flexible and expressive constructs for modeling of system dynamics, with an emphasis on hierarchical structures. It is also primarily in this respect that ECML has its advantages compared to the other executable CML's presented in Chapter 4. In ECML, all the control structures presented for these languages can be offered.

GDR has been developed from languages for real-time systems modeling, and provides only symbolic executions similar to executions of Petri-nets. Although tokens can have their data types defined, it is not possible to specify how outputs from processes are computed from inputs. Compared with ECML then, the language does not offer expressive constructs for modeling of static system aspects. It is powerful in modeling system behavior, but all its constructs can be represented in ECML. A simple reason for this is that the language has its semantics defined by translations to Petri-nets, which can be represented in ECML.

**TRQL and the Tracing Component**

It is most relevant to compare our tracing component with existing database approaches to execution tracing. None of the reviewed approaches are directed towards tracing of conceptual models, which is the major source of differences with our approach.

We find that Snodgrass' program monitoring system [109] is most language independent, since schemas can be defined as wanted. This has also the disadvantage that no guidelines are given as to what information to store about executions in order to understand them. Basu's [10] and Ledoux' [72] systems are directed to specific languages. Our approach comes in between, since we have defined a general schema suitable for different conceptual modeling languages.

Snodgrass' monitoring system only focuses on tracing dynamic program information like procedure calls etc. Basu's system traces Horn-clause resolutions, and does not consider updates of state components at all. In YODA, both tasking and updates of variables are traced. However, these variables are only of simple types. Our tracing component can store changes made to more complex data, including class instances.
Event reporting is done basically the same way in the different systems. Snodgrass relies on sensors to be placed in the program code or in basic system software to report the different events defined in the trace schema. In Basu's system, event reporting is not discussed at all. In YODA, a program annotator adds probes to the Ada code automatically. These are calls to report on events. In our system, the model executor is responsible for reporting events to a reported events handler. It is perfectly possible to use the tracing component integrated with other model executors, as long as they can use the reporting routine of the reported event handler.

Finally, trace querying is done somewhat differently in the different systems. Both TQuel and ESQL are general purpose query languages which provide powerful retrieval mechanisms. Their weakness is that they do not suggest how model executions should be understood. This can be overcome by developing more specialized languages on top of them. In YODA, the query language is PROLOG extended with some temporal operators. This provides more powerful retrieval mechanisms, e.g. recursive queries, but requires more skills from the person using it. Compared with these query languages, we claim that TRQL is more adapted to the problem of understanding executions of conceptual models. TQuel and YODA's query language have both a temporal expressiveness which is not offered by the current TRQL. Note however that some temporal expressiveness can be offered by TRQL simply by combining the queries in new ways. For instance, to answer a query like e.g. *What were the applications of rule r1 during the execution of process p1?*, first specify the execution period of interest to be the period that p1 was executing. Then ask for all executions of r1 during this period.

**The Modeling Cycle Framework**

We have now compared each of the parts of our work with existing research and existing systems. Each of these parts introduces some novelties on their own, however we also believe that the total modeling cycle framework as presented in this thesis has brought about some new ideas. There certainly exist environments which offer modeling, execution, and tracing of conceptual models, e.g. STATEMATE. However, in the framework introduced in this thesis, we have been striving for more general solutions. In the following, we therefore compare it with two groups of environments: Environments which support a set of alternative modeling languages, and meta CASE environments.

Multilanguage environments are based on the idea of having an expressive internal language, and on implementing translations back and forth between the external languages offered. In these environments, the external languages are fixed, and no support is given for metamodelling, i.e. new languages can not easily be introduced. With fixed languages, translations tend to be implemented without the support of a powerful translation facility. Neither in the AMADEUS project [91], nor with GDR such support is mentioned. A notable exception is ARIES, which uses POPART to specify translations.

Both the internal language in ARIES and GDR were compared with ECML above. Of these, ARIES with its translation facility has the greatest similarity with our approach to translation and execution. The simulation component in ARIES (Benner [11]) also makes
use of execution traces to examine behaviors, as was explained in Chapter 4. This work has been carried out simultaneously with our work on execution traces and explanation generation [49]. The validation questions used there complement the use of trace queries to understand model behavior, since they recognize patterns in the execution trace, while trace queries are aimed at explaining why certain patterns occur. Also, the interface to the explanation generator makes our approach different. A final comment is that ARIES does not aim at providing any metamodeling support, and is based on textual input languages only. Our concept suits meta CASE environments very well, as will be explained below.

Meta CASE environments like MetaEdit [108] and IPSYS [7] were presented in Chapter 2. Most such environments only give support for defining the syntax and presentation of modeling languages, ignoring issues like executability and validation. Also, very few give specific support for translations. As noted in Chapter 2, IPSYS provides a programming language for development of code generators, and hence for translations in general. This language is however similar to C and Ada, and hence it is not particularly tailored to translation tasks, as is TRL. From this we see that our modeling cycle framework complements existing meta CASE environments.

12.3 Experiences with the Prototypes

Due to time constraints, we have limited experience with the developed prototypes. The TRL interpreter has been tested with many of the PPP to ECML translation rules, in particular for translation of the bank model. It has further been tested on a Petri-net to ECML translation. Our experience is that both the specified rules and the interpreter work as desired.

ECML with its interpreter have been tested with parts of the bank model introduced in Chapter 2. We have modeled the transaction processing and its decomposition in ECML. We have included process logic for all of the five processes in the decomposition, with database manipulation included as operations of send and receive constructs in PLD’s. We have only executed the model with a limited number of class instances, which has given an acceptable performance. Models representing simple Petri-nets and finite state machines have also been executed with satisfactory behavior and performance.

The tracing component and its integration with the model executor have only been tested with Petri-net and FSM executions. For these executions, both the reported events handler and the query handler worked satisfactorily. When all the details of PPP executions are traced, we must expect the performance of the executor to decrease. It is however possible to turn off the tracing, if it is not needed. At the present time, the integration with the explanation generator has only been achieved on the conceptual level. However, the explanation generator has been developed and tested on some small example models. As described in Chapter 9, the interface to the trace query handler can be implemented with relatively small costs.
12.4 Integration with PPP

The integration of the developed components with the PPP environment was described in the previous chapter. It gives support for the modeling cycle with translations, executions, and tracing in PPP. Compared with the Ada and C generators, complete PPP models may now be executed, which include database manipulation and the use of timers and resources. With the expressiveness of ECML, we can also easily add new languages for specification of process logic. One alternative would be to use rules similar to the action rules in TEMPORA[78], another would be to use decision tables or decision trees. Mingwei Yang [139] proposes an extended PhM language, with complex attributes and methods, and data manipulation. So far, this language has not been supported by tools in PPP, and data manipulation has not been conceptually integrated with PLD. We believe that the extended PhM also can be successfully translated to ECML. Methods would have to implemented as functions or dynamic laws (without side effects) and dynamic laws (with side effects).

The translator facilities offered can be exploited to experiment with different implementations from PPP models, and they may also be used to support phase transitions in PPP. In Chapter 8, we indicated how initial PLD models can be generated from ports of undecomposed processes. Also if additional design languages are introduced in PPP, translations can be used to automate transitions to development phases using these languages.

Finally, with execution tracing and explanation generation, the support for model validation is enhanced in PPP.

12.5 Integration with Meta CASE Environments

As mentioned above, our modeling cycle framework complements the features of current meta CASE environments very well. Combining the two, user-orientation and model validation would be supported in the following ways:

- The languages used for modeling can be adapted to user or developer preferences, and to the characteristics of the problem domain.

- Model translations to executable models represented in ECML can be expressed in TRL. Models can subsequently be executed, and hence the advantages of prototyping can be exploited for model validation.

- Model executions are better understood by execution tracing and explanation generation. The trace query language can even be tailored to the modeling language by translating queries expressed in an 'external' query language into the internal form of TRQL queries.

Another advantage, not directly related to model validation, is that the translation facility can be exploited to support automatic phase transitions, as was suggested for PPP.
Chapter 13

Conclusions

In this final chapter, we summarize our achievements, and suggest directions for further research.

13.1 Main Results

In this thesis, we have motivated for the use of executable conceptual models in order to understand and validate requirements to information systems. We have developed an architecture which is aimed at supporting executable CML's through a modeling cycle framework. The major components of this architecture support in the translation, execution, and validation of conceptual models. The state of the art for each of these components has been assessed, and requirements to them have been identified. Generality leads to the ability to transfer the components to different CASE environments. Both theoretical and practical aspects of the components have been investigated. More specifically, the main results are the following:

- We have developed a rule-based language/framework for translation specification and implementation, TRL. The rules let translation knowledge be represented with source and target patterns at the metamodel level. Metamodels for executable CML's are assumed to be datamodels, possibly integrated with BNF or a similar formalism to express parts of the languages' syntaxes. Although the primary goal of this language is that it should be used for specification of prototype generation, we have discussed how it can be used for other translation tasks as well. We have developed a simple interpreter for the language, and we have exploited it for the specification of a prototype generator from conceptual models expressed in the CML's of the PPP CASE environment. The generality of the component makes it feasible to connect it to meta environments, and let executable CML's be designed according to the needs of system developers and users.

- To ease the specification of prototype generation, we have developed ECML, a highly expressive executable CML. This language is still at a high level of abstraction, and with implementation independent constructs. These were derived from an
analysis of existing generalizations of CML's. The language is particularly strong in the constructs it offers for modeling of system dynamics, since new control structures or hierarchical relationships can be specified declaratively. Also for this language, an interpreter has been implemented, and we have shown that it is a suitable target language for prototyping from conceptual models in PPP.

- We have developed a tracing component which is used as an aid in understanding model executions. It is founded on a general trace schema, which was derived from a view of a system history as a directed graph with state nodes and event edges. We further suggested a declarative trace query language TRQL for retrieval of interesting trace information. We have both shown how the queries can be used for quick explanations which only need access to trace information, and how they can be used as an interface to an explanation generator. An interpreter of an early version of the trace query language has been implemented. Together, conceptual modeling in a user-oriented language, model executions with tracing, and explanation generation form a comprehensive approach to validation of conceptual models.

We have evaluated each of the components and the modeling cycle framework with respect to the state of the art, and each of the components have been compared with their identified requirements. In this evaluation, we found some novelties both in the total concept, and in the individual components. So far, we have only limited experience with the prototypes, but they have been developed with a quick exploitation within the PPP environment in mind.

13.2 Further Work

There are many issues we would like to take up for further research. Some of the major one's are described in the following.

- We would like to complete and test the prototypes developed. The focus should be on robustness and efficiency. The limitations of the prototypes were listed in Chapter 10.

- It would be interesting to look more closely at the possibility to integrate the components with a state of the art meta CASE environment.

- The interface to the explanation generator has so far only been developed at the conceptual level. In order to demonstrate the usefulness of the comprehensive approach to validation of executable models, we must implement the necessary interface routines which turns results of trace queries into representations which can be understood by the explanation generator.

- Perhaps most of all, we need experience with using the modeling cycle components. Only in this way can we provide the methodological guidelines for how they should be exploited by developers and users in information systems development.
13.2. Further Work

- We would like to evaluate ECML through translations from more CML's. Although the language is claimed to be general, it may that some constructs should be supported explicitly, rather than through combinations of other constructs. We tried to find a balance between the number of constructs, their orthogonality in meaning, their abstraction level, and the ease of specifying translations. For the PPP language, the chosen balance seems appropriate. We need experimentation with other CML's as source languages in order to find if this holds in the general case.

Despite the remaining work, we hope that the reader is convinced that the modeling cycle framework as presented in this thesis is a feasible and good approach to support the development and validation of conceptual models.
Appendix A

Type Definitions in ECML

We present a list of types, type constructors, and functions and predicates defined upon these types, which are currently found interesting to include in ECML. The presentation is rather informal, based on the assumption that the semantics of functions and predicates are well-known. For a definition, see for instance [113].

Basic Types

Integer Numbers

\textbf{Functions} \ +, -, *, \textit{mod}, \textit{div} \\
\textbf{Predicates} \ =, \neq, >, \geq, <, \leq

Real Numbers

\textbf{Functions} \ +, -, *, \textbackslash \\
\textbf{Predicates} \ As for integers.

Strings

\textbf{Functions} \ S1 \oplus S2 (concatenation) \\
\textbf{Predicates} \ =, \neq, > (\textgreater \text{ is defined by comparison of ascii-codes})

Booleans

\textbf{Functions} \ The usual logical connectives. \\
\textbf{Predicates} \ =
Appendix A. Type Definitions in ECML

Constructed Types

Sets

- **Definition**: $T, T'$: type names, $T' = \text{set}(T)(\{T\})$ is a type definition.
- **Functions**: $\cup, \cap, \setminus$
- **Predicates**: $=, \subseteq, \emptyset(S)(S = \emptyset), s \in S$
- **Access mechanism**: $x \in S$

Sequences, Lists

- **Definition**: $T, T'$: type names, $T' = \text{list}(T)([T])$ is a type definition.
- **Functions**: $\oplus$ (concatenation) and $\ominus$ (subtraction), $\text{reverse}, \text{head}, \text{tail}, \text{last}, \text{front}$
- **Predicates**: $=, l \in L$
- **Access mechanism**: $x \in L$

Bags

- **Definition**: $T, T'$: type names, $T' = \text{bag}(T)(<T>)$ is a type definition.
- **Functions**: $\cup$ (bag union)
- **Predicates**: Same as for sets.
- **Access mechanism**: $x \in B$

Aggregations, Relations

- **Definition**: $T_1, \ldots, T_n, T'$: type names, $r_1, \ldots, r_n$ : role names
  $T' = (r_1 : T_1, \ldots, r_n : T_n)$ is a type definition.
- **Functions**: No special functions
- **Predicates**: $=$
- **Access mechanism**: $x.r_i$

Union Types

- **Definition**: $T_1, \ldots, T_n, T'$: type names, $T' = \bigcup T_i$ is a type definition
- **Access mechanism**: Must be compatible with at least one of the $T_i$. 
Enumerated Types

Definition If \( c_1, \ldots, c_n \) are constants, then \( T' = (c_1, \ldots, c_n) \) is a type definition.

Types of Classes

Definition \( T_1, \ldots, T_n : \) type names, \( r_1, \ldots, r_n : \) role names, \( C : \) class name
\( C = (r_1 : T_1, \ldots r_n : T_n) \) is a class definition.
The \( r_i \)'s are names of properties of the class \( C \).
If \( C_1 \) and \( C_2 \) are class names, then \( isa(C_1, C_2) \)
is a generalization definition with inheritance of properties:
\( x : C_1 \land y : C_2 \land isa(x, y) \land y.r_i = k \Rightarrow x.r_i = k \)
\( isa(x, y) \land isa(y, z) \Rightarrow isa(x, z) \)
Only single inheritance is supported so far.

Access mechanism \( x : C \) (\( x \) is an instance of class \( C \)), \( x.r_i \)
Appendix B

ECML Grammar

The grammar of ECML is presented. Note that function names or predicate names are either predefined (see section on type definitions), or self-defined. In the prototype, the latter must be coded in PROLOG. Refer back to Chapter 2 if the notation used is unclear.

Classdef ::= class(<Classname>,role(<Rolename>,Type))
Type ::= <Typename>|integer|real|string|boolean
Isa ::= isa(<Classname>,<Classname>)
Typedef ::= typedef(<Typename>,role(<Rolename>,Type))|typedef(<Typename>,set(Type))|typedef(<Typename>,list(Type))|typedef(<Typename>,bag(Type))|typedef(<Typename>,union([{Type}]*))|typedef(<Typename>,[{Constant}]*)
Variabledef ::= variable(Type,<Variablename>)
Initvariable ::= init(<Variablename>,Constant)
Constant ::= <Atom>||{Constant}||<Typename>||{(<Rolename>,Constant)}||
Statelaw ::= statelaw(<Statelawname>,Operationname,Statecomponentpath,Closedformula,Violateaction)
Operationname ::= insert|delete|update
Statecomponentpath ::= <Variablename>|<Classname>|<Variablename>.Updatepath|<Classname>.Updatepath
Updatepath ::= <Rolename>|<Rolename>.Updatepath|elements|elements.Updatepath
Violateaction ::= report|rollback|report_and_rollback|Lawreference
Dynamiclaw ::= dynamic_law(<Lawidentifier>,Precondition,Body,Postcondition,Inputs,Outputs,Lawproperties)
Precondition ::= Closedformula
Postcondition ::= Closedformula
Inputs ::= [{Variabledef}]*
Outputs ::= [{Variabledef}]*
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lawproperties</td>
<td>${\text{Lawproperty}}^*$</td>
</tr>
<tr>
<td>Lawproperty</td>
<td>$&lt;\text{Propertyname}&gt;({\text{Atom}})$</td>
</tr>
<tr>
<td>Body</td>
<td>sublaws(${\text{Lawstructure}}$, ${\text{Lawreference}}^*$)</td>
</tr>
<tr>
<td>Lawreference</td>
<td>$&lt;\text{Lawidentifier}&gt;({\text{Actualinputs}}, {\text{Actualoutputs}})$</td>
</tr>
<tr>
<td>Actualinputs</td>
<td>${\text{Actualinput}}^*$</td>
</tr>
<tr>
<td>Actualinput</td>
<td>$&lt;\text{Variablename}&gt;</td>
</tr>
<tr>
<td>Actualoutputs</td>
<td>Actualinputs</td>
</tr>
<tr>
<td>Tempvariable</td>
<td>A string starting with an uppercase letter</td>
</tr>
<tr>
<td>Operation</td>
<td>Insertoperation</td>
</tr>
<tr>
<td>Insertoperation</td>
<td>$\text{insert}(&lt;\text{Classname}&gt;)$ for the insertion of a class</td>
</tr>
<tr>
<td></td>
<td>$\text{insert}(\text{subclass}(&lt;\text{Classname}&gt;),\text{Tempvariable},\text{Superinstancespec})$</td>
</tr>
<tr>
<td></td>
<td>$\text{insert}(&lt;\text{Classname}&gt;,\text{Tempvariable},\text{Superinstancespec},\text{Assignments},\text{Tempvariable})$</td>
</tr>
<tr>
<td></td>
<td>$\text{insert}(&lt;\text{Classname}&gt;,\text{Tempvariable},\text{Freeformula},\text{Assignments})$</td>
</tr>
<tr>
<td></td>
<td>$\text{insert}(&lt;\text{Classname}&gt;,\text{Tempvariable},\text{Assignments})$</td>
</tr>
<tr>
<td>Superinstancespec</td>
<td>$\text{Freeformula}$</td>
</tr>
<tr>
<td>Deleteoperation</td>
<td>$\text{delete}(\text{Freeformula},{\text{Tempvariable}}^*)$</td>
</tr>
<tr>
<td>Updateoperation</td>
<td>$\text{update}(\text{Assignments})\text{update}(\text{Freeformula},\text{Assignments})$</td>
</tr>
<tr>
<td>Query</td>
<td>$\text{query}(\text{e(Term)})</td>
</tr>
<tr>
<td>Outputoperation</td>
<td>$\text{outputstring}(&lt;\text{String}&gt;)$</td>
</tr>
<tr>
<td>Readvariable</td>
<td>$\text{readvariable}(&lt;\text{Variablename}&gt;)$</td>
</tr>
<tr>
<td>Readpath</td>
<td>$&lt;\text{Rolename}&gt;</td>
</tr>
<tr>
<td>Externalcall</td>
<td>$\text{call}(\text{Lawreference})$</td>
</tr>
<tr>
<td>Applicationrule</td>
<td>$\text{applicationrule}(&lt;\text{Lawstructure}&gt;,\text{Candidates},\text{Select},\text{Addccl},\text{Addfacts})$</td>
</tr>
<tr>
<td>Candidates</td>
<td>$\text{first}[\text{all}]\text{user}[\text{onerandom}]\text{Candidatecondition}$</td>
</tr>
<tr>
<td>Select</td>
<td>$\text{first}[\text{all}]\text{user}[\text{onerandom}]\text{Selectcondition}$</td>
</tr>
<tr>
<td>Candidatecondition</td>
<td>$&lt;\text{Predicatename}&gt;({\text{Propertyname}},\text{Comp})$</td>
</tr>
<tr>
<td></td>
<td>$\text{not}(\text{Candidatecondition})</td>
</tr>
<tr>
<td></td>
<td>$\text{or}(\text{Candidatecondition},\text{Candidatecondition})$</td>
</tr>
<tr>
<td>Comp</td>
<td>$&lt;\text{Propertyname}&gt;({\text{Atom}})</td>
</tr>
<tr>
<td></td>
<td>$\text{min}(&lt;\text{Propertyname}&gt;)</td>
</tr>
<tr>
<td>Selectcondition</td>
<td>$\text{Candidatecondition}$</td>
</tr>
<tr>
<td>Addccl</td>
<td>$[\text{notcandidates}, [\text{notchosen}, [\text{notprecholds}]]]\text{none}$</td>
</tr>
<tr>
<td>Addfacts</td>
<td>$\text{none}[\text{executes}][\text{executed}][\text{executes},\text{executed}]$</td>
</tr>
<tr>
<td>Terminationrule</td>
<td>$\text{terminationrule}(&lt;\text{Lawstructure}&gt;,\text{Subfacts},\text{Terminatecondition},\text{Addtermccl},\text{Removeccl},\text{Subsublawfacts})$</td>
</tr>
<tr>
<td>Subfacts</td>
<td>$\text{none}[\text{executes}]$</td>
</tr>
<tr>
<td>Terminatecondition</td>
<td>$\text{true}[\text{lastsublaw}][\text{no_sublaw_executes}][\text{all_executed}][\text{noPrec_holds}][\text{prec_not_hold}][\text{not(Terminatecondition)}][\text{and}(\text{Terminatecondition},\text{Terminatecondition})][\text{or}(\text{Terminatecondition},\text{Terminatecondition})$</td>
</tr>
</tbody>
</table>
Addtermccl ::= next|none|sublaw
Removeccl ::= all|sublaw|none
Subsublawfacts ::= none|executed
Term ::= Variable|Constant|Function
Variable ::= <VariableName>|Tempvariable|<VariableName>.Rolenames
            Tempvariable.Rolenames
Rolenames ::= <Rolename>|<Rolename>.Rolenames
Function ::= <Functionname>((Term,Term)
            Setconstructor|Listconstructor|
            Bagconstructor|Aggregateconstructor
Setconstructor ::= set(Expressions,Freeformula)
Listconstructor ::= list(Expressions,Freeformula)
Bagconstructor ::= bag(Expressions,Freeformula)
Aggregateconstructor ::= <Typename>(<Rolename>,Term)*
Formula ::= Atomicpredicate|Quantifier(Tempvariable,Range,Formula)|
            not(Formula)|and(Formula,Formula)|or(Formula,Formula)|
            xor(Formula,Formula)|implies(Formula,Formula)
Closedformula ::= Formula with no free Tempvariable
Freeformula ::= free(Tempvariable,Range,Formula)
Atomicpredicate ::= <PredicateName>(<VariableName>(Term,Term)
Range ::= <Classname>|in(Variable)
Quantifier ::= forall|exists
Expressions ::= [{Term}]
Assignments ::= [{Variable,Term}]
Externalevent ::= externalevent(<Eventname>.operations({Operation}*))
Appendix C
TRL Grammar

The TRL grammar is presented. First, the grammar used for translations in PPP is given, adapted to the PPP repository. We also give the productions for the non-terminals which are different for the TRL based on the metalanguage introduced in Chapter 2.

T_program ::= t_program([<Goal>]*)
T_rule ::= Basic_construct_rule | Construct_property_rule
Basic_construct_rule ::= t_rule(<Name>,<Goal_achieved>,Source_model_pattern,
                       Precondition,Generate,Subgoals)
Construct_property_rule ::= t_rule(<Name>,<Goal_achieved>,Source_model_pattern,
                       Precondition,Context,Return,Generate,Subgoals)
Alternative_group ::= alternatives([<Name>]*)
Source_model_pattern ::= Metamodelquery|Othermodellement
Metamodelquery ::= Metamodelrelation|Metamodelrelation,|Condition|)*
Othermodellement ::= Modelement|[|Modelement|]*
Modelement ::= Term|Parsetree
Parsetree ::= <Nonterminal>|(Producing)*]|<Nonterminal>(<Constant>)
Production ::= Terminal|<Variable>|Parsetree
Terminal ::= '<Constant>'
Precondition ::= Condition|Condition|)*
Condition ::= true|Predicate|not(Predicate)|or(Condition,Condition)|
exist(Metamodelquery)
Predicate ::= Metamodelrelation|Comparison
Metamodelrelation ::= <Relation>|(Term)*
Term ::= <Variable>|<Constant>
Comparison ::= Term Relop Term
Relop ::= |=|<|>⋯
Context ::= Term|[|Term|]*
Subgoals ::= [|Subgoal]*
Subgoal ::= T_goal|Forall|Parse|Unparse|Combine|Text|
Other|Alternatives

259
Appendix C. TRL Grammar

TRL based on the metalanguage introduced in Chapter 2 is given below. Productions here replace those given above with the same non-terminals. The two variants are otherwise identical.

Metamodelquery ::= Metamodelrelation|{{Metamodelrelation}*},{Condition}*)
Metamodelrelation ::= <Variable>:=<Construct>
Condition ::= true|Predicate|not(Predicate)|or(Predicate,Predicate)
Relation ::= Relation(Term,Term)
Relation ::= Corresponds to Relop above
Term ::= <Variable>,<Propertyname>|<Constant>|<Variable>
Generate ::= <Construct>(({{<Propertyname>,Term})*)}
Appendix D

Translation Rules for PPP to ECML Translation

The translation rules from PPP to ECML are given, after a listing of the repository schema for PPP.

PPP Repository Schema

The schema here is taken from [5]. We only list those relations which will be used in the translation rules below.

PhM Relations

The relations are as follows:

entityclass( Id, Fromlist, Tolist, Type, Name)
subclass( Id, Fromlist, Tolist, Type, Name)
subgroup( Id, Fromlist, Tolist, Type, Name)
relationship( Id, Fromlist, Tolist, Type, Name)
attribute( Id, Fromlist, Tolist, Type, Name)
link( Id, From, To, Class, Name)
The domains are as follows:

<table>
<thead>
<tr>
<th>Domain</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>The identifier of the object</td>
</tr>
<tr>
<td>Fromlist</td>
<td>List of identifiers of links pointing at object.</td>
</tr>
<tr>
<td>Tolist</td>
<td>List of identifiers of links pointing from object.</td>
</tr>
<tr>
<td>Type</td>
<td>For attributes: Their value domain.</td>
</tr>
<tr>
<td>Name</td>
<td>Name of object.</td>
</tr>
<tr>
<td>From</td>
<td>The identifier of object at 'tail' of a link.</td>
</tr>
<tr>
<td>To</td>
<td>The identifier of object at 'head' of link.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>Type of link. Must be one of the following:</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>subclass link</td>
</tr>
<tr>
<td>201</td>
<td>id-attribute link</td>
</tr>
<tr>
<td>202</td>
<td>attribute link</td>
</tr>
<tr>
<td>203</td>
<td>repeating attribute link</td>
</tr>
<tr>
<td>204</td>
<td>quality attribute</td>
</tr>
<tr>
<td>301</td>
<td>in a relationship with full-1 type</td>
</tr>
<tr>
<td>302</td>
<td>in a relationship with full-n type</td>
</tr>
<tr>
<td>303</td>
<td>in a relationship with partial-1 type</td>
</tr>
<tr>
<td>304</td>
<td>in a relationship with partial-n type</td>
</tr>
<tr>
<td>401</td>
<td>has a subclass group</td>
</tr>
<tr>
<td>402</td>
<td>has a disjoint subclass group</td>
</tr>
<tr>
<td>403</td>
<td>has a composing subclass group</td>
</tr>
<tr>
<td>404</td>
<td>has a disjoint and composing subclass group</td>
</tr>
<tr>
<td>501</td>
<td>is a member of a subclass group</td>
</tr>
</tbody>
</table>

**PrM Relations**

The relations are as follows:

```plaintext
prm_diagram(Id,Hid,Procid,Plist,Slist,Alist,Tlist,ElFlist,EOFlist,OFlist,Flist)
```
The domains of the diagram relation are as follows:

<table>
<thead>
<tr>
<th>Domain</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>The diagram identifier.</td>
</tr>
<tr>
<td>Hid</td>
<td>The identifier of the higher level diagram.</td>
</tr>
<tr>
<td>Procid</td>
<td>The process which is decomposed into this diagram.</td>
</tr>
<tr>
<td>Plist</td>
<td>A list of processes (id’s) which are in the diagram.</td>
</tr>
<tr>
<td>Slist</td>
<td>A list of stores in the diagram.</td>
</tr>
<tr>
<td>Alist</td>
<td>A list of agents in the diagram.</td>
</tr>
<tr>
<td>Tlist</td>
<td>A list of timers in the diagram.</td>
</tr>
<tr>
<td>EIFlist</td>
<td>A list of flows entering the diagram from outside.</td>
</tr>
<tr>
<td>EOFlist</td>
<td>A list of flows leaving the diagram to outside.</td>
</tr>
<tr>
<td>OFList</td>
<td>A list of figures not among PrM constructs.</td>
</tr>
<tr>
<td>Flist</td>
<td>A list of the flows in the diagram.</td>
</tr>
</tbody>
</table>

process(Id,Decomp,Fromlist,Tolist,Pld,Name)
store(Id,Fromlist,Tolist,Type,Name)
agent(Id,Fromlist,Tolist,Type,Name)
timer(Id,Fromlist,Tolist,Type,Name)
flow(Id,From,To,Type,Name)
flowtypes(Id,Flowtypes)

The domains of these relations are as follows:

<table>
<thead>
<tr>
<th>Domain</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>Identifier of object.</td>
</tr>
<tr>
<td>Decomp</td>
<td>Identifier of decomposition diagram</td>
</tr>
<tr>
<td>Fromlist</td>
<td>List of flows entering the object.</td>
</tr>
<tr>
<td>Tolist</td>
<td>List of flows leaving the object.</td>
</tr>
<tr>
<td>Pld</td>
<td>If 0, the process has a PLD. If 1, the process is decomposed.</td>
</tr>
<tr>
<td>Type</td>
<td>For a flow, this may be the following: 0: non-terminate, non-trigger. 1: non-terminate, trigger. 2: terminate, non-trigger. 3: terminate, trigger.</td>
</tr>
<tr>
<td>Name</td>
<td>Name of object.</td>
</tr>
<tr>
<td>From</td>
<td>Identifier of object which is source of flow.</td>
</tr>
<tr>
<td>To</td>
<td>Identifier of object which is sink of flow.</td>
</tr>
<tr>
<td>Flowtypes</td>
<td>The names of types (simple) connected to a flow.</td>
</tr>
</tbody>
</table>

For port structures, the following relations are used:

in_port_node(Procid,Id,Level,T1,T2,Parent,Delist)
out_port_node(Procid,Id,Level,T1,T2,Parent,Delist)
in_port_table(Procid, Nodelist, Name)
out_port_table(Procid, Nodelist, Name)

The domains are as follows:

<table>
<thead>
<tr>
<th>Domain</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procid</td>
<td>The process to which the port belongs.</td>
</tr>
<tr>
<td>Id</td>
<td>Identifier of the port.</td>
</tr>
<tr>
<td>Level</td>
<td>Gives level in a composite structure. For a flow, it is 1.</td>
</tr>
<tr>
<td>T1</td>
<td>and, xor, or.</td>
</tr>
<tr>
<td>T2</td>
<td>single, repeat, cond, repcond, or condrep.</td>
</tr>
<tr>
<td>Parent</td>
<td>Identifier of higher level port.</td>
</tr>
<tr>
<td>Delist</td>
<td>List of port descendants.</td>
</tr>
<tr>
<td>Nodelist</td>
<td>For a complete port, the id of this port.</td>
</tr>
</tbody>
</table>

PLD Relations

The relations are as follows:

pld(Id, Procid, Type, From, To, Right, X, Y, H, Contents)
variable(Process, Variable, Type)

The domains are as follows:

<table>
<thead>
<tr>
<th>Domain</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>Identifier of pld construct</td>
</tr>
<tr>
<td>Procid</td>
<td>Identifier of process to which the pld belongs.</td>
</tr>
<tr>
<td>Type</td>
<td>start, assignment, choice, alternative, loop, receive, or send.</td>
</tr>
<tr>
<td>From, To, Right</td>
<td>Identifier of construct before, after and to the right.</td>
</tr>
<tr>
<td>X, Y, H</td>
<td>Layout information.</td>
</tr>
<tr>
<td>Contents</td>
<td>Differs for different constructs: start: empty assignment: assign(variable name, expression) choice: empty alternative: alternative('yes', 'else', or condition). loop: loop(condition) receive: receive(source, flow, list of variables and types) send: send(sink, flow, list of expressions)</td>
</tr>
<tr>
<td>Process</td>
<td>Name of process.</td>
</tr>
<tr>
<td>Variable</td>
<td>Name of variable.</td>
</tr>
<tr>
<td>Type</td>
<td>Type of variable.</td>
</tr>
</tbody>
</table>
Translation Rules from PhM to ECML

<table>
<thead>
<tr>
<th>name</th>
<th>id_attributerule</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal_achieved</td>
<td>id_attribute</td>
</tr>
<tr>
<td>source_model</td>
<td>(link(From,To,201),</td>
</tr>
<tr>
<td></td>
<td>entityclass(From,Class),</td>
</tr>
<tr>
<td></td>
<td>attribute(To,Type,Attr))</td>
</tr>
<tr>
<td>precondition</td>
<td>true</td>
</tr>
<tr>
<td>generate</td>
<td>[class(Class,role(Attr,Type)),</td>
</tr>
<tr>
<td></td>
<td>statelaw(Sname,insert,Class,</td>
</tr>
<tr>
<td></td>
<td>∀X:Class∀Y:Class(X≠Y ⇒ X.ATTR≠Y.ATTR),report),</td>
</tr>
<tr>
<td></td>
<td>statelaw(Sname,update,Class.ATTR,</td>
</tr>
<tr>
<td></td>
<td>∀X:Class∀Y:Class(X≠Y ⇒ X.ATTR≠Y.ATTR),report])</td>
</tr>
<tr>
<td>subgoals</td>
<td>[combine_texts([&quot;Id.attribute.of.&quot;,Class],Sname)]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>att_attributerule</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal_achieved</td>
<td>att_attribute</td>
</tr>
<tr>
<td>source_model</td>
<td>(link(From,To,202),</td>
</tr>
<tr>
<td></td>
<td>entityclass(From,Class),</td>
</tr>
<tr>
<td></td>
<td>attribute(To,Type,Attr))</td>
</tr>
<tr>
<td>precondition</td>
<td>true</td>
</tr>
<tr>
<td>generate</td>
<td>[class(Class,role(Attr,Type))]</td>
</tr>
<tr>
<td>subgoals</td>
<td>[]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>rep_attributerule</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal_achieved</td>
<td>rep_attribute</td>
</tr>
<tr>
<td>source_model</td>
<td>(link(From,To,203),</td>
</tr>
<tr>
<td></td>
<td>entityclass(From,Class),</td>
</tr>
<tr>
<td></td>
<td>attribute(To,Type,Attr))</td>
</tr>
<tr>
<td>precondition</td>
<td>true</td>
</tr>
<tr>
<td>generate</td>
<td>[class(Class,role(Attr,Attr)),</td>
</tr>
<tr>
<td></td>
<td>type(Attr.set(Type))]</td>
</tr>
<tr>
<td>subgoals</td>
<td>[]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>quality_attributerule</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal_achieved</td>
<td>quality_attribute</td>
</tr>
<tr>
<td>source_model</td>
<td>(link(From,To,204),</td>
</tr>
<tr>
<td></td>
<td>entityclass(From,Class),</td>
</tr>
<tr>
<td></td>
<td>attribute(To,Type,Attr))</td>
</tr>
<tr>
<td>precondition</td>
<td>true</td>
</tr>
<tr>
<td>generate</td>
<td>[type(Quality,role(Attr,Type))]</td>
</tr>
<tr>
<td>subgoals</td>
<td>[combine_texts([Class,'quality'],Quality)]</td>
</tr>
</tbody>
</table>
Appendix D. Translation Rules for PPP to ECML Translation

name: quality_variable_rule

goal_achieved: quality_variable

source_model:
  (entityclass(From,...,Class),
   exist(link(.,From,...,204,..)))

precondition: true

generate: [variable(Quality,Class)]

subgoals: [combine_texts([Class,'quality'],Quality)]

name: xp_1y_rule

goal_achieved: binary_relationship

source_model:
  (link(L1,E1,R,C1,...),link(L2,E2,R,C2,...),not(E1=E2),
   entityclass(E1,...,Class1),entityclass(E2,...,Class2),
   not(link(...,R,...)),not(link(L3,E3,R,...),not(E1=E3),not(E2=E3)),
   or(C1=303,C1=304),or(C2=301,C2=303),
   relationship(R,...,Rel))

precondition: true

generate: [class(Class1.role(Rel,Class2)),
   statelaw(Sname,insert,Class1,
   ∀X:Class1(X,Rel≠null ⇒ ∃Y:Class2(Y=X,Rel)),report),
   statelaw(Sname,update,Class1,Rel,∀X:Class1(X,Rel≠null ⇒ ∃Y:Class2(Y=X,Rel)),report)]

subgoals: [combine_texts(['Ref. constr. from ',Class1,'over ',Rel,'to ',Class2,Sname])]

name: xf_1y_rule

goal_achieved: binary_relationship

source_model:
  (link(L1,E1,R,C1,...),link(L2,E2,R,C2,...),not(E1=E2),
   entityclass(E1,...,Class1),entityclass(E2,...,Class2),
   not(link(...,R,...)),not(link(L3,E3,R,...),not(E1=E3),not(E2=E3)),
   or(C1=301,C1=302),or(C2=301,C2=303),
   relationship(R,...,Rel))

precondition: true

generate: [class(Class1.role(Rel,Class2)),
   statelaw(Sname,insert,Class1,∀X:Class1 ∃Y:Class2(Y=X,Rel)),report),
   statelaw(Sname,update,Class1,Rel,∀X:Class1 ∃Y:Class2(Y=X,Rel)),report)]

subgoals: [combine_texts(['Ref. and coverage constr. from ',
   Class1,'over ',Rel,'to ',Class2,Sname])]
name xp.ny_rule

goal_achieved binary_relationship

source_model (link(L1,E1,R,C1,...), link(L2,E2,R,C2,...), not(E1=E2),
entityclass(E1, ..., Class1), entityclass(E2, ..., Class2),
not(link(..., R, ...)), not(link(L3,E3,R,...), not(E1=E3), not(E2=E3)),
or(C1=303, C1=304), or(C2=302, C2=304),
relationship(R, ..., Rel))

precondition true

generate [class(Class1, role(Rel, Rel)), type(Rel, set(Class2)),
state law(Sname, insert, Class1, 
\[\forall X : Class1 \forall Y \in X, Rel \exists Z : Class2(Z = Y), \text{report},
state law(Sname, update, Class1, Rel, 
\[\forall X : Class1 \forall Y \in X, Rel \exists Z : Class2(Z = Y), \text{report}])

subgoals [combine_texts(["Ref. constr. from 'Class1, 'over ',Rel,'to',Class2,Sname])]


name xf.ny_rule

goal_achieved binary_relationship

source_model (link(L1,E1,R,C1,...), link(L2,E2,R,C2,...), not(E1=E2),
entityclass(E1, ..., Class1), entityclass(E2, ..., Class2),
not(link(..., R, ...)), not(link(L3,E3,R,...), not(E1=E3), not(E2=E3)),
or(C1=301, C1=302), or(C2=302, C2=304),
relationship(R, ..., Rel))

precondition true

generate [class(Class1, role(Rel, Rel)), type(Rel, set(Class2)),
state law(Sname1, insert, Class1, \[\forall X : Class1(X \neq \emptyset), \text{report},
state law(Sname1, update, Class1, Rel, \[\forall X : Class1(X \neq \emptyset), \text{report},
state law(Sname2, insert, Class1, \[\forall X : Class1 \forall Y \in X, Rel \exists Z : Class2(Z = Y), \text{report},
state law(Sname2, update, Class1, Rel, 
\[\forall X : Class1 \forall Y \in X, Rel \exists Z : Class2(Z = Y), \text{report}])

subgoals [combine_texts(["Ref. constr. from 'Class1, 'over ',Rel,'to',Class2,Sname2),
combine_texts(["Cov. constr. for Class1, 'over ',Rel,'to',Class2,Sname1])]

name relationship_att_attribute

goal_achieved relationship_att_attribute

source_model (link(... From, To, 202,...),
relationship(From, ..., Rel),
attribute(To, ..., Type, Attr))

precondition true

generate [class(Rel, role(Attr, Type))]

subgoals []
name: relationship_entityrule

name: relationship_entity

source_model: (relationship(From,....,Rel),
exist(link(.,From,To,202,..),
link(.,.,Entity,From,..),
entityclass(Entity,....,Ename))
precondition: true
generate: [class(Rel,role(Ename,Ename))]
subgoals: []

name: relationship_rep_attributeerule

goal_achieved: relationship_rep_attribute

source_model: (link(.,From,To,203,..),
relationship(From,....,Rel),
attribute(To,....,Type,Attr))
precondition: true
generate: [class(Rel,role(Attr,Attr)),
type(Attr,set(Type))]
subgoals: []

name: relationship_quality_attributeerule

goal_achieved: relationship_quality_attribute

source_model: (link(.,From,To,204,..),
relationship(From,....,Rel),
attribute(To,....,Type,Attr))
precondition: true
generate: [type(Quality,role(Attr,Type))]
subgoals: [combine_texts([Rel, 'quality', Quality])]

name: relationship_quality_variablererule

goal_achieved: relationship_quality_variable

source_model: (relationship(From,....,Rel),
exist(link(.,.,From,.,204,..)))
precondition: true
generate: [variable(Quality,Rel)]
subgoals: [combine_texts([Rel, 'quality', Quality])]

name: subclass_rule

goal_achieved: subclass

source_model: (link(.,From,To,101,..),
entityclass(From,.,.,Class1),
entityclass(To,.,.,Class2))
precondition: true
generate: [isa(Class1,Class2)]
subgoals: []
Appendix D. Translation Rules for PPP to ECML Translation

name: composingsubclass_rule
goal_achieved: composingsubclass
source_model:
(link(_From,To,501,...),
 entityclass(From,......,Class1),
 link(_,To,2,403,...),
 entityclass(To2,......,Class2),
 subgroup(To,Group,......))
precondition: true
generate:
[isa(Class1,Class2),
 statelaw(Sname,delete,Class2,\forall X:Class2(Composing),report),
 statelaw(Sname,insert,Class2,\forall X:Class2(Composing),report)]
subgoals:
[forall(Z,Group,t_goal(composing_formula,Z,X...),Exists),
 distribute_function(or(Exists),Composing),
 combine_texts(['Composing subclasses of',Class2,Sname])]

name: composing_formula_rule
goal_achieved: composing_formula
source_model: Subclassid
precondition: true
context: Variable
return: \exists X:Class(X=Variable)
generate: []
subgoals:
[entityclass(Subclassid,......,Class1)]

name: disjointcomposingsubclass_rule
goal_achieved: disjointcomposingsubclass
source_model:
(link(_From,To,501,...),
 entityclass(From,......,Class1),
 link(_,To,2,404,...),
 entityclass(To2,......,Class2),
 subgroup(To,Group,......))
precondition: true
generate:
[isa(Class1,Class2),
 statelaw(Sname1,insert,Class2,\forall X:Class2(Composing),report),
 statelaw(Sname2,insert,Class1,\forall Y:Class1(Disjoint),report)]
subgoals:
[forall(Z,Group,t_goal(composing_formula,Z,X...),Exists),
 distribute_function(or(Exists),Composing),
 combine_texts(['Composing subclasses of',Class2,Sname1),
forall(U,Group,t_goal(disjoint_formula,U,[Class1,Y]...),Notexists),
 distribute_function(and(Notexists),Disjoint),
 combine_texts(['Subclass disjointness for class',Class1,Sname2])]
### Translation Rules from PrM to ECML

<table>
<thead>
<tr>
<th>name</th>
<th>system.law.rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal_achieved</td>
<td>system.law</td>
</tr>
<tr>
<td>source_model</td>
<td><code>prm_diagram(D, ...</code>.Processes, ...)</td>
</tr>
<tr>
<td>precondition</td>
<td>true</td>
</tr>
<tr>
<td>generate</td>
<td><code>[dynamic.law(D,true,sublaws(vcs,Processes),true,[],[],[]])</code></td>
</tr>
<tr>
<td>subgoals</td>
<td>[]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>processrule</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal_achieved</td>
<td>process</td>
</tr>
</tbody>
</table>
| source_model          | `(process(Pid,Decomposition, ...),`
|                       | `in_port_table(Pid,Nodes, ...),`
|                       | `first(Nodes,Node),rest(Nodes,[],),`
|                       | `prm_diagram(Decomposition,Pid,Subprocesses, ...))` |
| precondition          | true                          |
| generate              | `[dynamic.law(Pid,Proc,sublaws(vcs,Subprocesses),true,[],[],[]))` |
| subgoals              | `[l.goal(process_precondition,[Pid,Node],[],[],[]))` |

<table>
<thead>
<tr>
<th>name</th>
<th>process_preconditionrule_trig_flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal_achieved</td>
<td>process_precondition</td>
</tr>
<tr>
<td>source_model</td>
<td><code>[Process,Portnode]</code></td>
</tr>
</tbody>
</table>
| precondition          | `(in_port_node(Process,Portnode,1, ...),`
|                       | `flow(Portnode,...,Process,Type,Flowname),` |
|                       | `or(Type=1,Type=3))`              |
| context               | []                                |
| return                | `∃X:Flowname(true)`               |
| generate              | []                                |
| subgoals              | []                                |

<table>
<thead>
<tr>
<th>name</th>
<th>process_preconditionrule_non_trig_flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal_achieved</td>
<td>process_precondition</td>
</tr>
<tr>
<td>source_model</td>
<td><code>[Process,Portnode]</code></td>
</tr>
</tbody>
</table>
| precondition          | `(in_port_node(Process,Portnode,1, ...),`
|                       | `flow(Portnode,...,Process,Type,Flowname),` |
|                       | `or(Type=0,Type=2))`                   |
| context               | []                                     |
| return                | []                                     |
| generate              | []                                     |
| subgoals              | []                                     |
Appendix D. Translation Rules for PPP to ECML Translation

```plaintext
name: process.preconditionrule.cond_rep

goal.achieved: process.precondition

source_model: [Process, Portnode]

precondition: (in_port_node(Process, Portnode, _, T2, _),
or(T2 = cond, or(T2 = condrep, T2 = repcond))
context: []
return: []
generate: []
subgoals: []

name: process.preconditionrule.and_xor

goal.achieved: process.precondition

source_model: [Process, Portnode]

precondition: (in_port_node(Process, Portnode, Level, T1, T2, _, Subnodes),
or(T2 = single, T2 = repeat), not((Level = 1))
context: []
return: Return
generate: []
subgoals: [forall(X, Subnodes, t_goal(process.precondition, [Process, X], [], ...), Result),
[(Result = [], Return = []),
(T1 = and, [construct_function(Result, and, And),
distribute_function(And, Return))],
(else, [construct_function(or, Result, Or),
distribute_function(Or, Return))]]

name: flow_rule

goal.achieved: flow

source_model: (flow(Flowid, From, To, Type, Flowname),
flowtype(Flowid, Role, Flowtype),
or((agent(From, _, _), Type = 1),
or((process(From, _), process(To, _))),
or(timer(From, _), timer(To, _))))
precondition: true
generate: []
subgoals: [t_goal(flow_types, Types, [Flowname, 1, ...]]

name: flow_types_rule

goal.achieved: flow_types

source_model: Types
precondition: true
context: [Flow, Current]
return: []
generate: [class(Flow, role(Current, Firstotype))]
subgoals: [first(Types, Firstotype), rest(Types, Restypes),
[(Restypes = [], true),
(else, [Next is Current + 1, t_goal(flow_types, Restypes, [Flow, Next, ...)])]]
```
### Translation Rules from PLD to ECML

<table>
<thead>
<tr>
<th>name</th>
<th>externalevents_rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal_achieved</td>
<td>externalevents</td>
</tr>
<tr>
<td>source_model</td>
<td>(flow(<em>From,To,1,Flowname),agent(From,</em>__,Agentname))</td>
</tr>
<tr>
<td>precondition</td>
<td>true</td>
</tr>
<tr>
<td>generate</td>
<td>[externalevent(Flowname,operations([insert(Flowname)]))]</td>
</tr>
<tr>
<td>subgoals</td>
<td>[]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>local_pld_variable_rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal_achieved</td>
<td>local_pld_variable</td>
</tr>
<tr>
<td>source_model</td>
<td>process(Processid,___,0,Pname)</td>
</tr>
<tr>
<td>precondition</td>
<td>true</td>
</tr>
<tr>
<td>generate</td>
<td>[variable(Processstype,Pname)]</td>
</tr>
<tr>
<td>subgoals</td>
<td>[combine_texts([Pname,'_data'],Processstype)]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>local_variable_type_rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal_achieved</td>
<td>local_variable_type</td>
</tr>
<tr>
<td>source_model</td>
<td>(variable(Process,Variable,Type),</td>
</tr>
<tr>
<td></td>
<td>process(Process,___,__Pname))</td>
</tr>
<tr>
<td>precondition</td>
<td>true</td>
</tr>
<tr>
<td>generate</td>
<td>[type(Processstype,role(Variable,Type))]</td>
</tr>
<tr>
<td>subgoals</td>
<td>[combine_texts([Pname,'_data'],Processstype)]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>processrule_undecomposed_process</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal_achieved</td>
<td>process</td>
</tr>
<tr>
<td>source_model</td>
<td>(process(Pid,___,0),</td>
</tr>
<tr>
<td></td>
<td>in_port_table(Pid,Nodes,___),first(Nodes,Node),rest(Nodes,[]),</td>
</tr>
<tr>
<td></td>
<td>pld(<strong><strong>,Pid,start,</strong><em>,To,</em></strong>,___))</td>
</tr>
<tr>
<td>precondition</td>
<td>true</td>
</tr>
<tr>
<td>generate</td>
<td>[dynamic_law(Pid,Precondition,sublaws(seq,Pld),true,[],[])]</td>
</tr>
<tr>
<td>subgoals</td>
<td>[l_goal(____,To,[]),pld,</td>
</tr>
<tr>
<td></td>
<td>t_goal(process.precondition,[Pid,Node],[],Precondition)]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>assignmentrule</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal_achieved</td>
<td>assignment</td>
</tr>
<tr>
<td>source_model</td>
<td>Id</td>
</tr>
<tr>
<td>precondition</td>
<td>(pld(Id,Process,assign_to,___,assign(Var,Expr)),</td>
</tr>
<tr>
<td></td>
<td>process(Process,___,Pn))</td>
</tr>
<tr>
<td>context</td>
<td>[]</td>
</tr>
<tr>
<td>return</td>
<td>Sequenceids</td>
</tr>
<tr>
<td>generate</td>
<td>[dynamic_law(Id,true,operations([update([([Pn.Var,Nexpr]]))]),true,[],[])]</td>
</tr>
<tr>
<td>subgoals</td>
<td>[parse(Expr,Tree),l_goal(expression,Tree,Pn,Nexpr),</td>
</tr>
<tr>
<td></td>
<td>t_goal(____,To,[]),Restsequence,combine_lists([Id,Restsequence],Sequenceids)]</td>
</tr>
</tbody>
</table>
Appendix D. Translation Rules for PPP to ECML Translation

name = receiverule

name = receivevariables_rule

given achieved = receive

given achieved = receivevariables

given achieved = readfromuser

source_model = Varlist

source_model = Varlist

source_model = Id

precondition = (pld(Id,Process,receive,...To,......,receive(...Flowid,Varlist)),
flow(Flowid,From,...,Type,Flowname),process(Process,...,Pname)
or((agent(From,......),Type=1),process(From,......)))

context = []

return = Sequences

generate = [dynamic_law(Id,3X:Flowname,operations(Operations),true,[],[],[])]

subgoals = [t_goal(receivevariables,Varlist,[Pname,X,1],Update),
combine_lists([[update(Update)],[delete(Y:Flowname,[Y])],Operations],
t_goal(_,To,[],Restsequence),combine_lists([[Id],Restsequence],Sequenceids)]

context = [Process,Xvar,Current]

return = Updates

generate = []

subgoals = [first(Varlist,par(Var,Type)),Next is Current+1,rest(Varlist,Restvar),
t_goal(receivevariables,Restvar,[Process,Xvar,Next],Rupdates),
combine_lists([[Process,Var,Xvar,Current]],Rupdates),Update)

context = [Process,Xvar,Current]

return = []

generate = []

subgoals = []

context = []

return = Sequences

generate = [dynamic_law(Id,true,operations(Operations),true,[],[],[])]

subgoals = [forall(X,Varlist,t_goal(readvariables,X,Process,..),Operations),
t_goal(_,To,[],Restsequence),combine_lists([[Id],Restsequence],Sequenceids)]
name: readvariablesrule

goal_achieved: readvariables

source_model: par(Var,Type)

precondition: process(Pid,......,Pname)

context: Pid

return: read_variable(Pname,Var)

generate: []

subgoals: []

name: sendrule

goal_achieved: send

source_model: Id

precondition: (pid(Id,Process,send,...,To,......,send(...Flowid,Exprlist)),

process(Process,......,Pname),

flow(Flowid,....,Flowto,....,Flowname),not(agent(Flowto,......)))

context: []

return: Sequenceids

generate: [dynamic_law(Id,true,operations([insert(X,Flowname,Inserts)]),true,[],[],[])]

subgoals: [t_goal(senddata,Exprlist,[Pname,X,1],Inserts),

t_goal(,...,To,[]),Restsequence,combine_lists([[Id],Restsequence],Sequenceids)]

name: senddata_rule

goal_achieved: senddata

source_model: Exprlist

precondition: not(Exprlist=[]) [Process,Var,Current]

context: [Process,Variable,Current]

return: Inserts

generate: []

subgoals: [first(Exprlist,Expr),first(Inserts,(Var,Current,Newexpr)),

parse(Expr,Tree),t_goal(expression,Tree,Process,Newexpr),

rest(Inserts,Rinserts),Next is Current+1,rest(Exprlist,Restexpr),

l_goal(senddata,Restexpr,[Process,Var,Next],Rinserts)]

name: senddata_rule

goal_achieved: senddata

source_model: Exprlist

precondition: Exprlist=[]

context: [Process,Variable,Current]

return: []

generate: []

subgoals: []
### Appendix D. Translation Rules for PPP to ECML Translation

<table>
<thead>
<tr>
<th>name</th>
<th>sendrule_query</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal_achieved</td>
<td>send</td>
</tr>
<tr>
<td>source_model</td>
<td>Id</td>
</tr>
<tr>
<td>precondition</td>
<td>(pld(Id,Process,send,...To,......,send(...Flowid,Exprlist)),</td>
</tr>
<tr>
<td></td>
<td>process(Process,......,Procname),</td>
</tr>
<tr>
<td></td>
<td>flow(Flowid,...Flowto,...Flowname),agent(Flowto,......))</td>
</tr>
<tr>
<td>context</td>
<td>[]</td>
</tr>
<tr>
<td>return</td>
<td>Sequenceids</td>
</tr>
<tr>
<td>generate</td>
<td>[dynamic_law(Id,true,operations(Outputs),true,[],[],[])</td>
</tr>
<tr>
<td></td>
<td>forall(X,Exprlist,t_goal(outputexpr,X,Procname,..),Outputs),</td>
</tr>
<tr>
<td></td>
<td>t_goal(...To,[],Restsequence),combine_lists([[Id,Restsequence],Sequenceids])</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>outputexprule</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal_achieved</td>
<td>outputexpr</td>
</tr>
<tr>
<td>source_model</td>
<td>Expr</td>
</tr>
<tr>
<td>precondition</td>
<td>true</td>
</tr>
<tr>
<td>context</td>
<td>Procname</td>
</tr>
<tr>
<td>return</td>
<td>query(e(Nexpr))</td>
</tr>
<tr>
<td>generate</td>
<td>[]</td>
</tr>
<tr>
<td>subgoals</td>
<td>[parse(Expr,Tree),t_goal(expression,Tree,Procname,Nexpr)]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>choiceerule_has_alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal_achieved</td>
<td>choice</td>
</tr>
<tr>
<td>source_model</td>
<td>Id</td>
</tr>
<tr>
<td>precondition</td>
<td>(pld(Id,Process,choice,...To,Right,......,choice),</td>
</tr>
<tr>
<td></td>
<td>pld(Right,......,Rright,......),not(Rright=' '),</td>
</tr>
<tr>
<td></td>
<td>process(Process,......,Procname))</td>
</tr>
<tr>
<td>context</td>
<td>[]</td>
</tr>
<tr>
<td>return</td>
<td>Sequenceids</td>
</tr>
<tr>
<td>generate</td>
<td>[dynamic_law(Id,true,sublaws(excseq,Excseq),true,[],[],[])</td>
</tr>
<tr>
<td></td>
<td>[t_goal(alternative,Right,[],Excseq),</td>
</tr>
<tr>
<td></td>
<td>t_goal(...To,[],Restsequence),combine_lists([[Id,Restsequence],Sequenceids])</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>alternativerule_not_receive</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal_achieved</td>
<td>alternative</td>
</tr>
<tr>
<td>source_model</td>
<td>Id</td>
</tr>
<tr>
<td>precondition</td>
<td>(pld(Id,Process,alternative,...To,Right,......,alternative(Cond)),</td>
</tr>
<tr>
<td></td>
<td>not(Cond=yes),not(Cond=else),process(Process,......,Procname))</td>
</tr>
<tr>
<td>context</td>
<td>Conditions</td>
</tr>
<tr>
<td>return</td>
<td>Alternativeids</td>
</tr>
<tr>
<td>generate</td>
<td>[dynamic_law(Id,Precondition,sublaws(seq,Sequence),true,[],[],[])</td>
</tr>
<tr>
<td></td>
<td>[t_goal(...To,[],Sequence),</td>
</tr>
<tr>
<td></td>
<td>parse(Cond,Tree),t_goal(condition,Tree,Procname,Precondition),</td>
</tr>
<tr>
<td></td>
<td>combine_lists([[Precondition],Conditions],Newconditions),</td>
</tr>
<tr>
<td></td>
<td>t_goal(alternative,Right,Newconditions,Restalternativeids),</td>
</tr>
<tr>
<td></td>
<td>combine_lists([[Id,Restalternativeids],Alternativeids])</td>
</tr>
</tbody>
</table>
Appendix D. Translation Rules for PPP to ECML Translation

```
name          choicerule_one_alternative
goal_achieved choice
source_model  Id
precondition  (pld(Id,Process,choice,...,To,Right,...,choice),
               pld(Right,...,Alto,'...','...','...',alternative(Cond)),
               process(Process,...,Procname))
context       []
return        Sequenceids
generate      [dynamic_law(Id,Id,sublaws(excseq,[Right,Newpld]),true,[],[],[]),
               dynamic_law(Right,Ncond,sublaws(seq,Sublaws),true,[],[],[]),
               dynamic_law(Newpld,not(Ncond),operations([]),true,[],[],[]),
               parse(Cond,Tree),t_goal(condition,Tree,Procname,Ncond),
               new_id(pld,Newpld),
               t_goal(...,Alto,[]),Sublaws),
               t_goal(...,To,[]),Restsequence),
               combine_lists([[Id],Restsequence],Sequenceids)]
```

```
name          looprule
goal_achieved loop
source_model  Id
precondition  (pld(Id,Process,loop,...,To,Right,...,loop(Cond)),
               process(Process,...,Procname))
context       []
return        Sequenceids
generate      [dynamic_law(Id,Precondition,sublaws(repseq,Repseq),true,[],[],[]),
               dynamic_law(Superid,Id,sublaws(excseq,[Id,Dummyid]),true,[],[],[]),
               dynamic_law(Dummyid,true,operations([]),true,[],[],[]),
               t_goal(...,Right,[]),Repseq,new_id(pld,Superid),new_id(pld,Dummyid),
               t_goal(...,To,[]),Restsequence),
               combine_lists([[Id],Restsequence],Sequenceids)]
```

```
name          additionrule
goal_achieved expression
source_model  expression([Expr,addop('+'),Term])
precondition  true
context       Procname
return        add(E,T)
generate      []
subgoals       [t_goal(expression,Expr,Procname,E),
               t_goal(term,Term,Procname,T)]
```
<table>
<thead>
<tr>
<th>name</th>
<th>subtractionrule</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal_achieved</td>
<td>expression</td>
</tr>
<tr>
<td>source_model</td>
<td>expression([Expr,addop(`-'),Term])</td>
</tr>
<tr>
<td>precondition</td>
<td>true</td>
</tr>
<tr>
<td>context</td>
<td>Procname</td>
</tr>
<tr>
<td>return</td>
<td>sub(E,T)</td>
</tr>
<tr>
<td>generate</td>
<td>[]</td>
</tr>
<tr>
<td>subgoals</td>
<td>[l_goal(expression,Expr,Procname,E), l_goal(term,Term,Procname,T)]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>simpletermrule</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal_achieved</td>
<td>expression</td>
</tr>
<tr>
<td>source_model</td>
<td>expression([Term])</td>
</tr>
<tr>
<td>precondition</td>
<td>true</td>
</tr>
<tr>
<td>context</td>
<td>Procname</td>
</tr>
<tr>
<td>return</td>
<td>T</td>
</tr>
<tr>
<td>generate</td>
<td>[]</td>
</tr>
<tr>
<td>subgoals</td>
<td>[l_goal(term,Term,Procname,T)]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>multiplyrule</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal_achieved</td>
<td>term</td>
</tr>
<tr>
<td>source_model</td>
<td>term([Term,mulop(`*'),Factor])</td>
</tr>
<tr>
<td>precondition</td>
<td>true</td>
</tr>
<tr>
<td>context</td>
<td>Procname</td>
</tr>
<tr>
<td>return</td>
<td>mul(T,F)</td>
</tr>
<tr>
<td>generate</td>
<td>[]</td>
</tr>
<tr>
<td>subgoals</td>
<td>[l_goal(term,Term,Procname,T), l_goal(factor,Factor,Procname,F)]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>divisionrule</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal_achieved</td>
<td>term</td>
</tr>
<tr>
<td>source_model</td>
<td>term([Term,mulop(`/'),Factor])</td>
</tr>
<tr>
<td>precondition</td>
<td>true</td>
</tr>
<tr>
<td>context</td>
<td>Procname</td>
</tr>
<tr>
<td>return</td>
<td>div(T,F)</td>
</tr>
<tr>
<td>generate</td>
<td>[]</td>
</tr>
<tr>
<td>subgoals</td>
<td>[l_goal(term,Term,Procname,T), l_goal(factor,Factor,Procname,F)]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>simplefactorrule</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal_achieved</td>
<td>term</td>
</tr>
<tr>
<td>source_model</td>
<td>term([Factor])</td>
</tr>
<tr>
<td>precondition</td>
<td>true</td>
</tr>
<tr>
<td>context</td>
<td>Procname</td>
</tr>
<tr>
<td>return</td>
<td>F</td>
</tr>
<tr>
<td>generate</td>
<td>[]</td>
</tr>
<tr>
<td>subgoals</td>
<td>[l_goal(factor,Factor,Procname,F)]</td>
</tr>
</tbody>
</table>
Appendix D. Translation Rules for PPP to ECML Translation

- **name**: simplevariable
- **goal_achieved**: factor
- **source_model**: factor([variable(Var)])
- **precondition**: true
- **context**: Proc
- **return**: Proc.Var
- **generate**: []
- **subgoals**: []

- **name**: simpleconstant
- **goal_achieved**: factor
- **source_model**: factor([constant(Const)])
- **precondition**: true
- **context**: Proc
- **return**: Const
- **generate**: []
- **subgoals**: []

- **name**: factor_is_expression
- **goal_achieved**: factor
- **source_model**: factor(['(',Expr,')'])
- **precondition**: true
- **context**: Proc
- **return**: E
- **generate**: []
- **subgoals**: [t.goal(expression,Expr,Proc,E)]

- **name**: conjunction
- **goal_achieved**: condition
- **source_model**: condition([Condition1.conn('AND'),Condition2])
- **precondition**: true
- **context**: Proc
- **return**: and(C1,C2)
- **generate**: []
- **subgoals**: [t.goal(condition,Condition1,Procname,C1), t.goal(condition,Condition2,Procname,C2)]

- **name**: disjunction
- **goal_achieved**: condition
- **source_model**: condition([Condition1.conn('OR'),Condition2])
- **precondition**: true
- **context**: Proc
- **return**: or(C1,C2)
- **generate**: []
- **subgoals**: [t.goal(condition,Condition1,Procname,C1), t.goal(condition,Condition2,Procname,C2)]
name: negation_rule
goal_achieved: condition
source_model: condition(['NOT(', 'Condition, ,')'])
precondition: true
context: Procname
return: not(C)
generate: []
subgoals: [l_goal(condition, Condition, Procname, C)]

name: condition_in_parentheses_rule
goal_achieved: condition
source_model: condition(['(', 'Condition, ,')'])
precondition: true
context: Procname
return: Cond
generate: []
subgoals: [l_goal(condition, Condition, Procname, Cond)]

name: condition_is_comparison_rule
goal_achieved: condition
source_model: condition([Comparison])
precondition: true
context: Procname
return: C
generate: []
subgoals: [l_goal(comparison, Comparison, Procname, C)]

name: comparison_rule
goal_achieved: comparison
source_model: comparison([Expression1, relop(Relop), Expression2])
precondition: true
context: Procname
return: Comparison
generate: []
subgoals: [l_goal(expression, Expression1, Procname, E1),
         l_goal(expression, Expression2, Procname, E2),
         [(Relop=\'=', Rel=equal), (Relop=\'<', Rel=lt),
          (Relop=\'>', Rel=gt), (Relop=\'>=', Rel=geq), (else, Rel=leq)],
         construct_function(Re, [E1, E2][Comparison])]
Appendix E

TRQL Grammar

The syntax of the trace query language TRQL is given. The use of the different queries is currently by means of predicates expressed in PROLOG. As an example, for the query 'FIND LAST CHANGES FOR Variable WHERE Variable> 0', a predicate change(last,Variable,Variable>0) is called. It is also possible to access the trace directly through the use of the relations from the trace schema.

The State Component View

\[
\text{Statecomponentview} ::= \text{FIND Occurrence CHANGES FOR Statecompref} \\
\text{WHERE Changecondition} \mid \text{FIND Navigate CHANGE FOR Statecompref} \\
\text{Occurrence} ::= \text{ALL} \mid \text{LAST} \mid \text{FIRST} \\
\text{Navigate} ::= \text{PREVIOUS} \mid \text{NEXT} \\
\text{Statecompref} ::= \text{Statecompid} \mid \text{Lawapplicationview} \mid \text{Referesview} \mid \text{Externaleventview} \mid \text{Statetlawview} \\
\text{Statecompid} ::= \text{<Variable>}\mid \text{<Variable>},\text{<Rolename>}\mid \text{<Tempvariable>},\text{<Rolename>},\text{Componentspec} \mid \text{<Tempvariable>},\text{Componentspec},\text{<Classname>} \\
\text{Componentspec} ::= \text{free} \mid \text{Operationtype},\text{Valuecondition},\text{not} \mid \text{Connective} \mid \text{Changecondition} \\
\text{Freeformula} ::= \text{See specification in grammar for ECML} \\
\text{Connective} ::= \text{and} \mid \text{or} \mid \text{xor} \mid \text{implies} \\
\text{Operationtype} ::= \text{insert} \mid \text{delete} \mid \text{update} \mid \text{query} \mid \text{input} \\
\text{Valuecondition} ::= \text{Relation(Term,Term)} \\
\text{Relation} ::= \text{equal} \mid \text{lt} \mid \cdots \\
\text{Term} ::= \text{Statecompvale} \mid \text{<Constant>} \quad \text{% ECML-constant} \\
\text{Statecompvale} ::= \text{<Variable>}\mid \text{<Variable>},\text{<Rolename>}\mid \text{<Tempvariable>},\text{<Rolename>}
\]
The External Event View

Externaleventview ::= FIND Occurrence OCCURRENCES OF <Eventref>
                    | FIND Navigate OCCURRENCE OF <Eventref>

The Law Application View

Lawapplicationview ::= FIND Occurrence APPLICATIONS OF Lawref
                     | FIND Navigate APPLICATIONS OF Lawref
Lawref ::= <Lawidentifier>|(Statecomponentview)
         | (Refersview)|(Contextview)

The State Law View

Statelawview ::= FIND Occurrence VIOLATIONS OF <Lawidentifier>
                 | FIND Navigate VIOLATIONS of <Lawidentifier>

The Context View

Contextview ::= Superlawview|Concurrencyview
Superlawview ::= FIND Superocc OF Law
Superocc ::= ALLSUPER|SUPER
Concurrencyview ::= FIND CONCURRENT OF Law
Law ::= (Lawapplicationview)|(Statecomponentview)|(Contextview)
       | (Refersview)

The Refers View

Refersview ::= Referforwardview|Referbackwardview
Referforwardview ::= FIND REFERENCES-TO Laworevent
Referbackwardview ::= FIND IS-REFERRED-BY Law
Laworevent ::= (Externaleventview) | Law
The State View

Stateview ::= FIND Occurrence STATES WHERE Statecondition|
            FIND Navigatestates STATE

Navigatesates ::= PREVIOUS|NEXT|CURRENT|INITIAL

Statecondition ::= EXTERNALEVENT|EXTERNALEVENT(<Eventref>)|
                  STATE.LAW|Operationtype|INITIATED(<Lawidentifier>)|
                  TERMINATED(<Lawidentifier>)|CHANGED(Statecompvalue)|
                  Connective(Statecondition,Statecondition)|not(Statecondition)

Allevents ::= ALLEVENTS Stateview

Historicalvalues ::= FIND VALUE OF Statecompexpression IN (Stateview)

Statecompexpression ::= Statecompvalue|hyp(Statecompvalue=Value)|
                       Function(Statecompexpression,Statecompexpression)

Function ::= add|sub|⋯

Fromspecification ::= FROM STATE Stateview

Tospecification ::= TO STATE Stateview
Appendix F

TRQL Queries in Relational Calculus

We specify the data retrieved by TRQL queries through specifications similar to relational calculus.

We do not intend to give a complete specification of TRQL, but concentrate on the central aspects of query evaluations. The trace relations referred to in the specifications below were described in Chapter 9. They are as follows:

```
dynamicLaw(Id, From, To, PreconditionValues, Inputs, Outputs)
externalEvent(Id, From, To)
stateLaw(Id, From, To, ReferencedValues)
stateChange(Id, To, ReferencedValues, StateComponent, Value, Type)
```

Refer back to Chapter 9 if the meanings of these are unclear. We refer to tuples of these relations by \( x \in Relation \), and to the different attributes by \( x.\text{Attribute} \). We use obvious abbreviations for some of the attributes above. We assume that all queries refer to an execution period starting in state number \( \text{start} \), and ending in state number \( \text{end} \). The variable \( \text{current} \) is used to represent the state number for the current retrieved tuple in a navigation query, i.e. when 'next' or 'previous' is used in the query.

For each query, certain variables are given. The specifications are labeled with values of non-terminals from the TRQL grammar, to indicate which query they correspond to. The returned data are grouped in sets, or indicated by placing them in front of the '|' symbol.

State Component View

Let \( s \) denote reference to a state component in the query, and let \( c \) be the condition associated with the query. Further, \( \text{matches}(x, y) \) is defined as follows:
matches(x, y) ⇔ x = y ∨ x = a ∧ y = a.b ∨ x = a.b ∧ y = a ∨ instance(y, x)

Here, x is the reference to the state component in the query, and y is the reference to a state component from an instance of state_change. Matching was described informally in Chapter 9. Then we have the following specifications of queries in the state component view:

Occurrence=all:
\{(x, y) | (x ∈ dynamic.law ∨ x ∈ external.event) ∧ y ∈ state_change ∧
matches(s, y.scomp) ∧ y.to ≥ start ∧ y.to ≤ end ∧ x.id = y.id ∧ x.to = y.to ∧ holds(c)\}

Occurrence=first:
\{(x, y) | (x ∈ dynamic.law ∨ x ∈ external.event) ∧ y ∈ state_change ∧
matches(s, y.scomp) ∧ y.to ≥ start ∧ y.to ≤ end ∧ x.id = y.id ∧ x.to = y.to ∧ holds(c) ∧ ¬∃u(u ∈ state_change ∧ matches(s, u.scomp) ∧ u.to < y.to ∧ u.to ≥ start)\}

Occurrence=last:
\{(x, y) | (x ∈ dynamic.law ∨ x ∈ external.event) ∧ y ∈ state_change ∧
matches(s, y.scomp) ∧ y.to ≥ start ∧ y.to ≤ end ∧ x.id = y.id ∧ x.to = y.to ∧ holds(c) ∧ ¬∃u(u ∈ state_change ∧ matches(s, u.scomp) ∧ u.to > y.to ∧ u.to ≤ end)\}

Navigate=next:
\{(x, y) | (x ∈ dynamic.law ∨ x ∈ external.event) ∧ y ∈ state_change ∧
matches(s, y.scomp) ∧ y.to > current ∧ y.to ≤ end ∧ x.id = y.id ∧ x.to = y.to ∧ holds(c) ∧ ¬∃u(u ∈ state_change ∧ matches(s, u.scomp) ∧ u.to < y.to ∧ u.to > current)\}

Navigate=previous:
\{(x, y) | (x ∈ dynamic.law ∨ x ∈ external.event) ∧ y ∈ state_change ∧
matches(s, y.scomp) ∧ y.to ≥ start ∧ y.to < current ∧ x.id = y.id ∧ x.to = y.to ∧ holds(c) ∧ ¬∃u(u ∈ state_change ∧ matches(s, u.scomp) ∧ u.to > y.to ∧ u.to < current)\}

Note that we have omitted the specification of condition evaluation, since this only involves rather trivial comparisons.

**External Event View**

Here, ev is the reference to an external event given in the query.

Occurrence=all:
\{\{(x, y) | x ∈ external.event ∧ x.id = ev ∧ x.to ≥ start ∧ x.to ≤ end ∧ y = \{u | u ∈ state_change ∧ u.to = x.to ∧ u.id = x.id\}\}\}

Occurrence=first:
\{\{(x, y) | x ∈ external.event ∧ x.id = ev ∧ x.to ≥ start ∧ x.to ≤ end ∧ y = \{u | u ∈ state_change ∧ u.to = x.to ∧ u.id = x.id\} ∧ ¬∃v(v ∈ external.event ∧ v.id = ev ∧ v.to ≥ start ∧ v.to < x.to)\}\}
Law Application View

Here, \( l \) is the reference to a dynamic law given in the query. The first five specifications apply for the case that \( l \) is simple, while the latter five for the case that \( l \) is complex.

Occurrence=all:
\[
\{(x,y) \mid x \in \text{dynamic-law} \land x.id = l \land x.to \geq \text{start} \land x.to \leq \text{end} \land y = \{u \mid u \in \text{state-change} \land u.to = x.to \land u.id = x.id\}\}
\]

Occurrence=first:
\[
(x,y) \mid x \in \text{dynamic-law} \land x.id = l \land x.to \geq \text{start} \land x.to \leq \text{end} \land y = \{u \mid u \in \text{state-change} \land u.to = x.to \land u.id = x.id\} \land \neg \exists v (v \in \text{dynamic-law} \land v.id = l \land v.to \geq \text{start} \land v.to < x.to)
\]

Occurrence=last:
\[
(x,y) \mid x \in \text{dynamic-law} \land x.id = l \land x.to \geq \text{start} \land x.to \leq \text{end} \land y = \{u \mid u \in \text{state-change} \land u.to = x.to \land u.id = x.id\} \land \neg \exists v (v \in \text{dynamic-law} \land v.id = l \land v.to \leq \text{end} \land v.to > x.to)
\]

Navigate=next:
\[
(x,y) \mid x \in \text{dynamic-law} \land x.id = l \land x.to > \text{current} \land x.to \leq \text{end} \land y = \{u \mid u \in \text{state-change} \land u.to = x.to \land u.id = x.id\} \land \neg \exists v (v \in \text{dynamic-law} \land v.id = l \land v.to > \text{current} \land v.to < x.to)
\]

Navigate=previous:
\[
(x,y) \mid x \in \text{dynamic-law} \land x.id = l \land x.to \geq \text{start} \land x.to \leq \text{current} \land y = \{u \mid u \in \text{state-change} \land u.to = x.to \land u.id = x.id\} \land \neg \exists v (v \in \text{dynamic-law} \land v.id = l \land v.to < \text{current} \land v.to > x.to)
\]

Occurrence=all:
\[
\{x \mid x \in \text{dynamic-law} \land x.id = l \land x.to \geq \text{start} \land x.to \leq \text{end}\}
\]
Occurrence=first:

\[ x \mid x \in \text{dynamic}\_law \land x.id = l \land x.to \geq \text{start} \land x.to \leq \text{end} \land \neg \exists v \in \text{dynamic}\_law \land v.id = l \land v.to \geq \text{start} \land v.to < x.to \]

Occurrence=last:

\[ x \mid x \in \text{dynamic}\_law \land x.id = l \land x.to \geq \text{start} \land x.to \leq \text{end} \land \neg \exists v \in \text{dynamic}\_law \land v.id = l \land v.to \leq \text{end} \land v.to > x.to \]

Navigate=next:

\[ x \mid x \in \text{dynamic}\_law \land x.id = l \land x.to \geq \text{current} \land x.to \leq \text{end} \land \neg \exists v \in \text{dynamic}\_law \land v.id = l \land v.to > \text{current} \land v.to < x.to \]

Navigate=previous:

\[ x \mid x \in \text{dynamic}\_law \land x.id = l \land x.to \geq \text{start} \land x.to \leq \text{current} \land \neg \exists v \in \text{dynamic}\_law \land v.id = l \land v.to < \text{current} \land v.to > x.to \]

Law Context View

Here, \( l \) is assumed to be a law application, i.e. \( l \in \text{dynamic}\_law \). We further assume the existence of a superlaw relation \( \text{super}(l, l') \) which holds if \( l' \) is the superlaw of \( l \). The predicate \( \text{allsuper}(l, l') \) holds if \( l' \) is an ascendant of \( l \) in the law structure.

\[ \text{allsuper}(l, l') \Leftrightarrow (\text{super}(l, l') \lor \exists l''(\text{super}(l'', l') \land \text{allsuper}(l'', l'))) \]

\[ \text{allsuper}(l) = \{ l' \mid \text{allsuper}(l, l') \} \]

Superocc=super:

\[ x \mid x \in \text{dynamic}\_law \land \text{super}(l.id, x.id) \land x.from \leq l.from \land x.to \geq l.to \]

Superocc=allsuper:

\[ \{ x \mid x \in \text{dynamic}\_law \land x.id \in \text{allsuper}(l.id) \land x.from \leq l.from \land x.to \geq l.to \} \]

Concurrent applications:

\[ \{ x \mid x \in \text{dynamic}\_law \land \text{super}(l.id, s) \land \text{super}(x.id, s) \land x \neq l \land (x.from \geq l.from \land x.from < l.to \lor x.to > l.from \land x.to \leq l.to \lor x.from < l.from \land x.to > l.to) \} \]

Refers View

In the backwards direction, \( l \) is a given law application, while \( c \) is the set of associated tuples of \( \text{state}\_change \), i.e. \( l \in \text{dynamic}\_law \land c = \{ s \mid s \in \text{state}\_change \land x.id = s.id \land x.to = s.to \} \). For a complex law, \( c \) is empty. In the forwards direction, \( l \) is either an application of a simple dynamic law, or an occurrence of an external event.

The specifications for the backward and forward direction are given below. Note that for
the forwards direction, one must assure that no other dynamic law application has changed the same state component.

Backwards:

\[
\{ (x,y) \mid (u,v) \in l.prec \lor z \in c \land (u,v) \in z.ref) \land (x \in dynamic.law \lor x \in external.event) \land y \in state.change \land matches(u,y.scomp) \land y.to \leq l.from \land x.id = y.id \land x.to = y.to \land \exists w \in state.change \land matches(u,w.scomp) \land w.to > y.to \land w.to \leq l.from) \}
\]

Forwards:

\[
\{ (x,y) \mid z \in c \land u = z.scomp \land v = z.value \land x \in dynamic.law \land ((u,v) \in x.prec \lor \exists s \in state.change \land s.id = x.id \land s.to = x.to \land (u,v) \in s.ref) \land x.from > l.to \land \lnot \exists w'(l' \in dynamic.law \land w' \in state.change \land w'.id = l'.id \land w'.to = l'.to \land w'.scomp = u \land l'.to < x.from \land l'.to > l.to) \land y = \{ w \mid (w \in state.change \land w.id = x.id \land w.to = x.to) \}\}
\]

State Law View

Here, sl is the reference to a state law given in the query.

Occurrence=all:

\[
\{ (x,y) \mid x \in state.law \land x.id = sl \land x.to \geq start \land x.to \leq end \land y = \{ u \mid u \in state.change \land u.to = x.to \land u.id = x.id \}\}
\]

Occurrence=first:

\[
(x,y) \mid x \in state.law \land x.id = sl \land x.to \geq start \land x.to \leq end \land y = \{ u \mid u \in state.change \land u.to = x.to \land u.id = x.id \} \land \lnot \exists v \in state.law \land v.id = sl \land v.to \geq start \land v.to < x.to
\]

Occurrence=last:

\[
(x,y) \mid x \in state.law \land x.id = sl \land x.to \geq start \land x.to \leq end \land y = \{ u \mid u \in state.change \land u.to = x.to \land u.id = x.id \} \land \lnot \exists v \in state.law \land v.id = sl \land v.to \leq end \land v.to > x.to
\]

Navigate=next:

\[
(x,y) \mid x \in state.law \land x.id = sl \land x.to > current \land x.to \leq end \land y = \{ u \mid u \in state.change \land u.to = x.to \land u.id = x.id \} \land \lnot \exists v \in state.law \land v.id = sl \land v.to > current \land v.to < x.to
\]

Navigate=previous:

\[
(x,y) \mid x \in state.law \land x.id = sl \land x.to \geq current \land x.to \leq start \land y = \{ u \mid u \in state.change \land u.to = x.to \land u.id = x.id \} \land \lnot \exists v \in state.law \land v.id = sl \land v.to < current \land v.to > x.to
\]
State View

Here, we only give the specification for the case that all events related to a state $n$ are to be retrieved.

The retrieved information includes all dynamic laws initiated or terminated, and any external events or state laws.

\[
\begin{align*}
&\{(x,y) \mid x \in \text{dynamic\_law} \land \text{simple}(l) \land (l.\text{from} = n \lor l.\text{to} = n) \land y = \{u \mid u \in \text{state\_change} \land u.\text{id} = x.\text{id} \land u.\text{to} = x.\text{to}\}\} \cup \\
&\{x \mid x \in \text{dynamic\_law} \land \text{complex}(l) \land (l.\text{from} = n \lor l.\text{to} = n)\} \cup \\
&\{(x,y) \mid x \in \text{external\_event} \land (l.\text{from} = n \lor l.\text{to} = n) \land y = \{u \mid u \in \text{state\_change} \land u.\text{id} = x.\text{id} \land u.\text{to} = x.\text{to}\}\} \cup \\
&\{x \mid x \in \text{state\_law} \land (l.\text{from} = n \lor l.\text{to} = n)\}
\end{align*}
\]
Bibliography


293


