Hicons

A General Diagrammatic Framework for Hierarchical Modelling

Guttorm Sindre
Trondheim, July 1990

Information Systems Group
Dept. of Electrical Engineering and Computer Science
The Norwegian Institute of Technology
The University of Trondheim
N-7034 Trondheim, Norway
Abstract

This thesis is based on the assumption that hierarchical structure is essential for the human ability to construct large complex systems such as information systems in an efficient way. The notion of hierarchical structure is discussed, and the support for hierarchical abstraction mechanisms in current information systems engineering languages is surveyed.

Then **Hicons**, a general diagrammatic framework for hierarchical modelling, is developed. A special emphasis is placed on diagrammatic representation, where two major styles are identified: the *onion style* and the *tree style*. Of these, the onion style is considered the most interesting, and thus, this notation is given most attention.

The applicability of the framework and the suggested diagrammatic representation is evaluated by looking at the possibility to couple it to other languages and by comparing it to a similar framework developed by others. In particular, we discuss the possibilities of using the framework with the modelling languages of PPM and TEMPORA, where comparisons are made by means of an already existing solution to the IFIP Conference Example.
Preface

About the Thesis

This is a thesis submitted to the Norwegian Institute of Technology for the doctoral degree “doktor ingeniør”. The work has been carried out at the Information Systems Group, Dept. of Electrical Engineering and Computer Science, The Norwegian Institute of Technology, The University of Trondheim, Norway, under the supervision of Prof. Arne Sølvberg.

The thesis is structured in six parts:

I: Problem Statement

The assumption about the importance of hierarchical abstraction mechanisms is discussed (chapter 1). Some weaknesses of the current practice of information systems engineering are identified (chapter 2). Following from this, the objective of the work is stated (chapter 3).

II: Preliminaries on Hierarchies

The necessary preliminaries on hierarchies are given. These include the discussion of the notions of hierarchies and hierarchical relations (chapter 4), structuring principles (chapter 5), constructivity (chapter 6), and a state-of-the-art survey of the support of hierarchical abstraction mechanisms in current information systems specification languages (chapter 7). This part is mainly based on literature study (but the chapter on constructivity also draws upon experiences from previous research by ourselves).

III: The Hicon Framework

In this part we develop our language framework. First the linguistic criteria on which we base our choices are clarified (chapter 8). Then we discuss the conceptual basis of the framework (chapter 9) and present the diagrammatic constructs (chapter 10).

IV: Evaluation

In this part we try to assess the applicability of the Hicon framework. First we discuss its applicability in general, looking briefly at the possibilities of coupling it to different kinds of languages and comparing it with the similar Higraph framework (chapter 11). Then we discuss in more detail the possibilities for using Hicons together with the external languages of PPM (chapter 12) and TEMPORA (chapter 13).

V: Visions and Achievements

This part discusses our visions for the future of the Hicon framework, and sums up what has been achieved this far. First we discuss how method and tool support could be provided for the framework (chapter 14). Then we conclude the thesis by summing up what has been done and making suggestions for further work (chapter 15).
VI: Appendices

This part contains background material on our previous research on constructivity (A), on the meta-level language $\theta$ (B), and on the TEMPORA project (C).

Acknowledgements

In connection with the work on this thesis I would like to thank the following people:

- My supervisor, Prof. Arne Sølvberg, for his encouragement and constructive criticism, and for all that he has taught me during my participation in the research projects DAISEE, RHAPSODY, and TEMPORA at the Information Systems Group here at the Norwegian Institute of Technology.

- All my colleagues at the Division of Computer Science, as well as partners in the TEMPORA project, for the interesting discussions and social activities. Some specific thanks to Bård Helge Hofstad, for helping me with the lay-out and document processing tools. Also, some extra thanks to Andreas Lothe Opdahl, who has been my office mate for these years, and with whom I have discussed some of the major ideas of my thesis from their infant stage.

- The undergraduate students that I have guided (or misguided) during the period of this doctoral work (As stated by Robert Heinlein: "The teacher who does not learn from his students, does not teach.").

In the spring term of 1988 I was a research assistant at the Division of Computer Science. From the summer of 1988 my work has been financed by a grant from NTNF. During this period, I have also been a part time employee of SINTEF (SINTEF-B).

Trondheim, July 1990.

Guttorm Sindre
Contents

I Problem Statement xv

1 A Philosophical Outlook 3

2 The Problems of IS Engineering 7
  2.1 Terminology .......................... 7
  2.2 On a Buzzword: Software Crisis .... 8
    2.2.1 The Software Crisis .......... 8
    2.2.2 Why IS Engineering is Difficult 9
  2.3 Weaknesses in the Current Practice . 10
    2.3.1 Rigid Phase-Distinctions ...... 11
    2.3.2 Target System Orientation .... 13
    2.3.3 Concepts Which Lack Generality 14
    2.3.4 Lack of Abstractive Power ... 14

3 Objectives 17
  3.1 Idealistic Objective: Pan-Presumptionism .... 17
    3.1.1 Non-Presumptionism: a Dead-End Street 17
    3.1.2 The Hierarchical Presumption .... 18
    3.1.3 The Road Towards Pan-Presumptionism 18
  3.2 More Realistic Objectives ........ 18

II Preliminaries on Hierarchies 21

4 Basic Hierarchy Concepts 25
  4.1 What Is a Hierarchy? .............. 25
CONTENTS

4.1.1 Hierarchical: A Question of Definition .................. 25
4.1.2 Hierarchical: a Question of Degree .................. 29

4.2 Four Standard Relations .................................. 31
4.2.1 Set-Theoretic Definitions ............................. 31
4.2.2 How Good are the Suggested Relations? ............... 33

4.3 Hierarchies and Pan-Presumptionism ...................... 35

5 Structuring Principles ........................................ 37
5.1 Structuring Principles and Aggregation Principles ....... 37
5.2 Single and Multiple Principle Approaches ................ 41
5.3 Data-Orientation and Real World Orientation .............. 42
5.4 Structuring and Pan-Presumptionism ...................... 43

6 Constructivity .................................................. 45
6.1 The Fundamental Principle .................................. 45
6.2 Constructivity in BNM and PPM ......................... 47
6.2.1 BNM .............................................. 47
6.2.2 PPM .............................................. 52
6.2.3 Experiences Made ................................ 54
6.3 Constructivity and Pan-Presumptionism .................. 55

7 State-of-the-Art Survey ....................................... 57
7.1 Our Focus of Comparison ................................... 57
7.2 Language Survey ............................................... 58
7.2.1 ER .............................................. 59
7.2.2 PhM .............................................. 59
7.2.3 NIAM ............................................ 60
7.2.4 ERT ............................................. 61
7.2.5 SHM+ ........................................... 63
7.2.6 DFD ............................................. 65
7.2.7 PrM ............................................. 65
7.2.8 Petri nets ....................................... 66
7.2.9 Statecharts ....................................... 67
CONTENTS

7.2.10 DADES ........................................ 68
7.2.11 CIM ........................................... 68
7.2.12 ERAE ......................................... 69
7.2.13 Semantic Networks .............................. 71
7.2.14 Higraphs ........................................ 73
7.3 Summing up .......................................... 73
  7.3.1 Concepts ...................................... 73
  7.3.2 Diagrammatic Representation ................. 74
7.4 State-of-the-Art and Pan-Presumptionism ........ 75

III The Hicon Framework .................................. 77

8 Language Criteria ...................................... 81
  8.1 Language Quality .................................. 81
    8.1.1 What is Language Quality? ................... 81
    8.1.2 How Can Language Quality Be Measured? ...... 82
    8.1.3 A Framework for Language Quality Criteria .. 82
  8.2 Conceptual Criteria ............................... 83
    8.2.1 Perceptibility ................................ 83
    8.2.2 Expressive Power .............................. 83
    8.2.3 Expressive Economy ............................ 84
    8.2.4 Method/Tool Potential ......................... 85
  8.3 External Representation Criteria .................. 85
    8.3.1 Perceptibility ................................ 85
    8.3.2 Expressive Power .............................. 87
    8.3.3 Expressive Economy ............................ 87
    8.3.4 Method/Tool Potential ......................... 88
  8.4 Examples ......................................... 88
    8.4.1 Perceptibility ................................ 89
    8.4.2 Expressive Power .............................. 92
    8.4.3 Expressive Economy ............................ 93
CONTENTS

8.4.4 Criteria Dependency ...................................... 95
8.5 Language Criteria and Pan-Presumptionism .................. 97

9 Conceptual Basis ............................................. 99
  9.1 A Meta-Level Language .................................... 99
  9.2 Constructs to be Predefined ................................. 100
    9.2.1 Nodes, Names, and Labels .............................. 100
    9.2.2 Classes, Instances, and Masses ....................... 100
    9.2.3 Hierarchical Constructs .............................. 101
    9.2.4 Non-Totality ........................................ 102
    9.2.5 Overlapping ......................................... 103
    9.2.6 Logical Connectives and Operators ................... 104
    9.2.7 User-Defined Relations ............................... 104
  9.3 Conceptual Basis and Pan-Presumptionism .................. 104

10 Diagrammatic Representation ................................ 107
  10.1 Introduction ............................................ 107
  10.2 Onion Notation ......................................... 109
    10.2.1 Sources of Inspiration ............................... 112
    10.2.2 Nodes and Connectives ............................... 113
    10.2.3 Some Diagrammatic Tricks ............................ 117
    10.2.4 User-Defined Constructs ............................. 127
  10.3 Tree Notation ........................................... 129
    10.3.1 Nodes and Composition .............................. 129
  10.4 Diagrams and Pan-Presumptionism ......................... 131

IV Evaluation .................................................. 133

11 General Applicability .................................... 137
  11.1 Adding Hicons to Existing Languages ..................... 137
    11.1.1 SHM+ ............................................. 138
    11.1.2 DFD ............................................. 139
CONTENTS

11.1.3 Petri Nets ...................................................... 142
11.1.4 ER ............................................................. 144
11.1.5 STD ........................................................... 147
11.1.6 Semantic Networks ........................................... 149
11.1.7 ERAE .......................................................... 150
11.1.8 Summing up ................................................... 151
11.2 A Comparison with Higraphs ................................. 151
  11.2.1 Something More than Higraphs ............................ 152
  11.2.2 Elegance ..................................................... 152
  11.2.3 Conclusion .................................................. 153

12 Hicons and PPM .................................................. 155
  12.1 Statics: PhM .................................................... 155
    12.1.1 PhM as of Today ......................................... 155
    12.1.2 PhM with Hicons ........................................ 157
  12.2 Dynamics: PrM ................................................ 160
    12.2.1 PrM as of Today ......................................... 164
    12.2.2 PrM with Hicons ........................................ 167
    12.2.3 The IFIP Conference Example ........................... 176
  12.3 Effects of Introducing Hicons in PPM ...................... 182
    12.3.1 Gains .................................................... 182
    12.3.2 Threats .................................................. 183
    12.3.3 Conclusion ................................................. 183

13 Hicons and TEMPORA ........................................... 185
  13.1 Introduction ................................................ 185
  13.2 Hicons for ERT and EPL .................................... 185
  13.3 The Hierarchical Treatment of Rules ....................... 190
    13.3.1 Coupling Rules to Other Models ...................... 191
    13.3.2 Rule Hierarchies ..................................... 194
  13.4 Conclusion .................................................. 198
V  Visions and Achievements  

14 Method and Tool Issues  

14.1 Methodological Flexibility  
14.2 The Hicon CASE-Shell  
14.3 The Future: One Path, Not Two  

15 Conclusions  

15.1 Claimed Contributions  
15.2 Main Conclusions  
15.3 Directions for Further Work  

VI  Appendices  

A  A Description of θ  

A.1 The Use of Relations in θ  
A.2 The θ Interpreter  
A.3 Defining Useful Relations in θ  
A.4 How We Would Use θ  

B  The TEMPORA Project  

B.1 Introduction to the TEMPORA Project  
  B.1.1 Partners and Background  
  B.1.2 The Objectives of TEMPORA  
B.2 The External Languages  
  B.2.1 ERT  
  B.2.2 External Process Language  
  B.2.3 The External Rule Language
List of Figures

2.1 The Waterfall paradigm ........................................ 12

4.1 Six graphs ......................................................... 28
4.2 Two graphs with hierarchical tendencies ....................... 28
4.3 Two general digraphs ............................................. 29
4.4 Two weighted digraphs .......................................... 30
4.5 Hierarchical and non-hierarchical relations .................... 31
4.6 Association with a single child ................................ 33

5.1 Three aggregation principles ...................................... 39
5.2 Three separate hierarchies ...................................... 41

6.1 A Behaviour Network, before and after abstraction .......... 49
6.2 An STD for the network ......................................... 49
6.3 The PrM port symbols ........................................... 53
6.4 A PrM diagram, before and after abstraction ................. 54
6.5 The STD for the simple guano packing network ................ 54

7.1 An ER diagram ................................................... 59
7.2 Phenomena and information objects ............................. 60
7.3 Hierarchic constructs of PhM ................................... 60
7.4 A NIAM diagram .................................................. 61
7.5 Aggregation in ERT .............................................. 62
7.6 Generalization in ERT .......................................... 63
7.7 Structural hierarchies in SHM+ .................................. 63
7.8 Dynamic hierarchies in SHM+ ................................... 64
7.9 Database actions .................................................. 64
LIST OF FIGURES

7.10 Vague composition in DFD ..................................... 65
7.11 Aggregation and generalization in PrM ........................... 66
7.12 Aggregation and generalization in Petri nets .................... 67
7.13 Generalization and aggregation in statecharts ................... 67
7.14 Overlapping states ............................................. 68
7.15 Generalization and association in ERAE .......................... 70
7.16 Contexts in ERAE ............................................. 70
7.17 A semantic network ........................................... 72
7.18 A Conceptual Graph ............................................ 72

8.1 Connectivity and emphasis ....................................... 86
8.2 Venn diagrams .................................................. 89
8.3 Entity class and value class symbols in PhM and ERT ............ 90
8.4 Emphasis in ERT ............................................... 91
8.5 Crossing lines misleading emphasis ............................... 92
8.6 A graph to illustrate expressive economy ........................ 93
8.7 Few symbols (left) or expressive economy (right) ............... 96
8.8 Connectivity of value classes in ERT (left) and PhM (right) ... 98

9.1 Dealing with overlapping nodes .................................. 104

10.1 Tree style vs. onion style ...................................... 108
10.2 Two notations for hierarchies ................................... 110
10.3 Adding on non-hierarchical relations ............................ 111
10.4 Generalization and aggregation in Higraphs ..................... 112
10.5 Onion notation connective distinction ............................ 114
10.6 Two alternatives for the plain node/vague composition icon .... 116
10.7 The other four Hicons .......................................... 116
10.8 Node composition ............................................... 116
10.9 Space loss on combining different Hicons ........................ 117
10.10 Space loss with two subnodes .................................. 118
10.11 Stretching for more efficient packing .......................... 118
10.12 Reshaped symbols for efficient packing ........................ 119
<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.13</td>
<td>Node concatenation</td>
<td>120</td>
</tr>
<tr>
<td>10.14</td>
<td>A non-total partition</td>
<td>120</td>
</tr>
<tr>
<td>10.15</td>
<td>Overlapping through node concatenation</td>
<td>122</td>
</tr>
<tr>
<td>10.16</td>
<td>Overlapping with a Higraph-like technique</td>
<td>122</td>
</tr>
<tr>
<td>10.17</td>
<td>Intersection and meaning: The Higraph notation</td>
<td>124</td>
</tr>
<tr>
<td>10.18</td>
<td>The combined area/curve convention</td>
<td>124</td>
</tr>
<tr>
<td>10.19</td>
<td>Unnamed overlaps</td>
<td>125</td>
</tr>
<tr>
<td>10.20</td>
<td>Negation and inconsistency</td>
<td>126</td>
</tr>
<tr>
<td>10.21</td>
<td>The <em>nothing</em> node and its use</td>
<td>127</td>
</tr>
<tr>
<td>10.22</td>
<td>Tree notation</td>
<td>130</td>
</tr>
<tr>
<td>10.23</td>
<td>Tree notation and overlapping</td>
<td>131</td>
</tr>
<tr>
<td>11.1</td>
<td>Traditional and Hiconion SHM+ aggregation</td>
<td>139</td>
</tr>
<tr>
<td>11.2</td>
<td>Objects and operations: alternative representations</td>
<td>140</td>
</tr>
<tr>
<td>11.3</td>
<td>Several operations on one structure</td>
<td>141</td>
</tr>
<tr>
<td>11.4</td>
<td>Aggregation in Petri nets</td>
<td>143</td>
</tr>
<tr>
<td>11.5</td>
<td>Generalization in Petri nets</td>
<td>144</td>
</tr>
<tr>
<td>11.6</td>
<td>A simple ER diagram</td>
<td>145</td>
</tr>
<tr>
<td>11.7</td>
<td>Higraph extension for the ERD</td>
<td>146</td>
</tr>
<tr>
<td>11.8</td>
<td>Hiconion equivalent to the Higraph extension</td>
<td>146</td>
</tr>
<tr>
<td>11.9</td>
<td>A simple ST diagram</td>
<td>147</td>
</tr>
<tr>
<td>11.10</td>
<td>Higraph extension of the ST diagram</td>
<td>148</td>
</tr>
<tr>
<td>11.11</td>
<td>Hiconion for the Higraph extension</td>
<td>148</td>
</tr>
<tr>
<td>11.12</td>
<td>Some semantic networks with Hiconions</td>
<td>150</td>
</tr>
<tr>
<td>11.13</td>
<td>Relationship composition in Higraphs?</td>
<td>152</td>
</tr>
<tr>
<td>12.1</td>
<td>A PhM Diagram for the IFIP example</td>
<td>158</td>
</tr>
<tr>
<td>12.2</td>
<td>The properties of Response</td>
<td>158</td>
</tr>
<tr>
<td>12.3</td>
<td>Hiconion for the previous PhM</td>
<td>161</td>
</tr>
<tr>
<td>12.4</td>
<td>Slightly zoomed-out Hiconion for the same system</td>
<td>162</td>
</tr>
<tr>
<td>12.5</td>
<td>The properties of Response</td>
<td>163</td>
</tr>
<tr>
<td>12.6</td>
<td>Datatype decomposition</td>
<td>163</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

12.7 The PrM port symbols .............................................. 165
12.8 A small composite port example ................................... 166
12.9 PrM for the IFIP Conference Case ................................. 168
12.10 The decomposition of P1 ........................................... 169
12.11 The decomposition of P2 ........................................... 169
12.12 BNM at the lowest level ........................................... 170
12.13 Notational simplification for events .............................. 172
12.14 Increased clarity with Hicons .................................... 172
12.15 Increased flexibility with Hicons ................................. 173
12.16 Sequencing and timing .............................................. 174
12.17 Conditions for a choice ............................................ 175
12.18 The overview diagram in Hiconions ............................... 178
12.19 Identification of processes ....................................... 179
12.20 Decompositions of P2 .............................................. 179
12.21 The decomposition of P1 .......................................... 180
12.22 Final description of P1.3 .......................................... 181

13.1 Overview ERT diagram for the IFIP example ...................... 186
13.2 The properties of Response in ERT ................................. 187
13.3 Hiconion for the overview ERT diagram ........................... 188
13.4 Hiconion for the values of Response .............................. 189
13.5 Simulating the quality relation by introducing meta-levels .... 189
13.6 The two alternatives for hierarchical structuring in TEMPORA 191
13.7 Rules connected to ERT Hiconions ................................ 192
13.8 Rules connected to EPL Hiconions ................................ 193
13.9 The refers-to relation ................................................. 196
13.10 A little goal hierarchy ............................................. 197
13.11 An overrule hierarchy .............................................. 197

B.1 The TEMPORA application modelling languages .................. 219
B.2 ERT symbols ......................................................... 220
B.3 A Process Hierarchy Diagram ...................................... 220
List of Tables

6.1 The analysis of the path without loops .................. 50
6.2 The analysis of the path with only the t3-loop .......... 51
6.3 The analysis of the path with both loops ............... 52
8.1 Table for the JRJ example ............................. 94
Part I

Problem Statement
Part Introduction

This part clarifies the main philosophical assumptions of this thesis, our views on the problems of information systems engineering, and what we intend to do to contribute to the reduction of some of these problems.

Chapter 1 states the basic philosophical assumption of the thesis, namely that hierarchic abstraction mechanisms are essential for supporting the human perception of complex systems (like for instance information systems).

Chapter 2 describes some of the main problems of information systems engineering as of today and discusses to what extent these problems can be solved. Some difficulties are identified as being inherent in the nature of information systems themselves, whereas others seem to be the result of bad practice. Among the latter are the lack of flexibility and the poor support for hierarchies found in many current approaches for information systems development.

Chapter 3 states what we will try to achieve in this thesis, namely a general diagrammatic framework for hierarchical modelling. Admitting that this framework is mainly interesting as a step towards an idealistic long term goal, we also describe our vision of this goal.
Chapter 1

A Philosophical Outlook

Looking out through the window of my office on a sunny day when it is sort of painful to be sitting in front of the screen writing a thesis, I can see a park. Beautiful bikini dressed female students are lying on the lawn, more or less eagerly reading for their next exam, occasionally changing their positions to escape from the moving shadows of the huge trees. Having passed my Physics exam I know, of course, that everything can be broken down to atoms and photons, Trees, grass, paths, human beings — just an enormous collection of tiny particles. Emotions — love is a chemical reaction in the brain, says the physicist, just like hatred, happiness, sorrow or anything else. Some say, according to this, that the universe is flat. Nothing above, nothing below, no higher purpose.

Be these arguments what they may. It would be impossible to perceive the whole park in terms of atoms and make sense of it. Reductionism (the opinion that the world is essentially flat) might be a valid philosophical point of view, but a human being always perceiving its surroundings according to such principles would hardly be the fittest for survival. As will be argued in the following, we need to reason in terms of hierarchical abstraction mechanisms.

This thesis is going to present a general diagrammatic framework for hierarchical modelling, with the long-term ambition of improving the practice of information systems engineering. This ambition is based on the following assumptions:

- Hierarchies are essential for the human understanding of complex systems, and

- thinking in terms of hierarchical constructs such as aggregation and generalization is very natural to the human being.

- Information systems are complex systems because they must reflect the part of the real world which they process information about, and

- thus, a proper support for hierarchical constructs is an essential requirement throughout the entire information system development process.

It is easy to find support in literature for the importance of hierarchies, both
• in nature,
• in human thinking, and
• in the field of information systems engineering.

The importance of hierarchies in nature is pointed out in works such as [5], [75], [86], [117], [119], for instance by means of statements like

• "Hierarchical order is nowhere more striking than in biological systems." [42]

• "Hierarchical organization is so universal in the biological world that we usually pass it off as the natural way to achieve simplicity or efficiency in a large collection of interacting elements." [87]

• "It is a commonplace observation that nature loves hierarchies." [99]

These works emphasize how hierarchical structure is essential in reducing the evolution time and increasing the stability of complex systems, both natural and artificial.

The importance of hierarchies in human thinking is recognized for instance by Sowa in [104]\(^1\) and emphasized even more strongly by Whyte in [120] (quotation):

"The immense scope of hierarchical classification is clear. It is the most powerful method of classification used by the human brain-mind in ordering experience, observations, entities, and information. Though not yet definitely established as such by neuro-physiology and psychology, hierarchical classification probably represents the prime mode of co-ordination or organization (i) of cortical processes, (ii) of their mental correlates, and (iii) of the expressions of these symbolisms and languages. As a reminder of its great scope I cite the hierarchical classification of

• Numbers, scales, times, positions, crystal forms, symmetry elements, and groups.

• Symbolisms, sentences, and languages of all kinds.

• Logical types, concepts, principles, information, quantities, and abstractions of many other sorts.

The use of hierarchical ordering must be as old as human thought, conscious and unconscious,..."

Within the field of information systems engineering, there is also a significant support for the importance of hierarchical structuring, as indicated by the following quotations:
• [112] “Hierarchical notions pervade computing. Hierarchy appears in the organization of program units, in the organization of data, and in the organization of information flow during processing.”

• [122]: “...there are times when understanding can come only through the suppression of detail”

• [113]: “Anyone who has ever attempted the design of a large system realizes the importance of having a well-defined structure — any structure — as a tool for organizing thinking.” and later in the same article: “To this author, structural advantages of the pure tree structure over alternate structures are of immense value in practice, making the pure tree structure the ideal candidate for such a choice.”

• [78] “One of the most significant advances in the practice of computer system development and in the description of computer systems is unquestionably the notion of abstraction,...” and “A hierarchical design makes it possible to associate precisely the right level of complexity with the abstraction at each level.”

• [62]: “In order that large systems be designed in a reasonable time, a structured design methodology has to be employed. This involves the use of many levels of abstraction...”

Support for the importance of hierarchies is also implicit from the fact that many specification languages for information systems provide some hierarchical abstraction mechanisms, as will be shown in chapter 7. Still, there might be people who doubt that hierarchies are as important as claimed above. There are several formalisms used in connection with information systems development that are not particularly hierarchical, for instance the relational model, or state transition diagrams. Some might even maintain that these flat formalisms do very well. However, research has indicated that they do not (as stated by [28] (the relational model) and [48] (state-transition diagrams)). Thus, we feel pretty confident in our assumptions about the importance of hierarchical abstraction mechanisms.
Chapter 2

The Problems of IS Engineering

2.1 Terminology

The lack of an agreed terminology is a problem in information systems research. Thus, we find it convenient to start by defining some essential terms which will be frequently used in the rest of the work:

**IS:** information system: a system for the storage, retrieval, and processing of information within some organization. An IS will consist of automated as well as manual parts.

**CIS:** computerized information system; the part of an IS which is made up of computer hardware and software.

**method:** a set of guidelines for reaching a certain goal. In our case, the goal is the construction of an information system which satisfies the information handling needs of the target organization(s), and the system must be delivered within some time and cost limits.

**model:** a specific representation of some system, or parts of a system, at any level of abstraction. The system may be an already existing one, or one only considered for construction.

**language:** a collection of symbols (the *alphabet* of the language) together with some rules for how these can be put together to form statements (*syntax*) and some explicit or understood definitions of what different statements mean (*semantics*). If the language is formal (i.e. translatable to some logic), we can also use the word *formalism*.

**approach:** a language and the methods given for modelling systems with this language.

**verification:** checking the consistency and completeness of a model.

**validation:** checking that the model is in accordance with the actual intentions and needs of the customer.
Most noteworthy in the above definitions is the restriction on the use of the word "model". Many authors use the word "model" also when talking about a language in general rather than a specific representation of some system in this language (for instance, "the ER model", "the Jackson model" etc.). This practice may be confusing and will be avoided in this work. Thus, we would say "Fig. X shows an ER model of the X Library", but "the ER language has some severe weaknesses". Also, many authors prefer the word "methodology" instead of "method", but the former, meaning the science or study of methods, is definitely a bit exaggerated when talking about a set of steps suggested for modelling something in one particular language.

Having made precise our definitions of the above terms, we are ready for the more serious problems that are facing the IS engineering discipline, problems for which many people find it appropriate to use the word "crisis".

2.2 On a Buzzword: Software Crisis

2.2.1 The Software Crisis

Probably, there are thousands of software science works written to date with some section headed by the title "The Software Crisis". Is there really a crisis, and if so, how does it manifest itself? The points made by authors writing about the software crisis are mainly the following:

- the productivity increase in software development has been very disappointing compared to other engineering disciplines, for instance hardware development

- information systems engineering is still a very labour intensive activity compared to other engineering disciplines

- the methods used are much more vague than in other disciplines,

- time and cost limits are often broken

- maintenance costs on software systems are very high, sometimes as much as 80% of the total costs

The standard follow-up in such works is a statement that the problems result from the fact that the current methods for software development are not good enough. The more ambitious go on with proposing some method which they believe will address the problems. Lots of such methods have emerged, but none seem to have had any fundamental success. Why is this?
2.2. ON A BUZZWORD: SOFTWARE CRISIS

2.2.2 Why IS Engineering is Difficult

Confronted with the "software crisis" we agree very much with those authors who stress that IS engineering is fundamentally more difficult than other kinds of engineering (e.g. [94], [60], [20]). These authors maintain that the comparison of software development with for instance hardware development is rather unfair, because

• IS problems are often wicked problems ([94], [60]). The main characteristics of wicked problems are the following:

  - wicked problems have no definite formulation
  - every formulation of a wicked problem is in effect also some formulation of a solution
  - trying to solve a wicked problem, it is impossible to know when to stop, and
  - attempted solutions are untestable and generally unrepeatable
  - any wicked problem can be considered a symptom of another problem, so one is never certain that one is addressing it at the right level
  - every wicked problem is essentially unique

Thus, it is impossible to be completely rational about solving a wicked problem (like building a complex information system) — one has to be more or less intuitive. This does not mean that one has to go about in a completely ad hoc manner, but any method used should allow for intuition and accept the wickedness of the problem.

• information systems are more complex than other man-manufactured systems ([20]):

  - Information systems usually have a much larger state space than other kinds of systems, even hardware.
  - There is no repetition of components. Making a RAM twice as big, one can simply double the number of components, whereas an information system twice as big will necessarily mean the introduction of lots of new components.

• there are few firm principles ([20]). Whereas a discipline like physics can lean on the laws of nature, information systems have to adapt to the unpredictability of human behaviour, and the chaos of institutions in our society.

• information systems are more often required to change ([60], [20]). First of all they are easier to change (or at least believed to be). Second they are under a higher pressure to change because they among other things encode the most often changing parts of an organization, like business policies, working routines etc.
software constructs are invisible and impossible to visualize ([20]) Since visualization is very important for the human understanding of complex problems, this is a major drawback.

Thus, the term "software crisis" is really a manifestation of the fact that customer requirements for information systems are more ambitious than what the developers are able to cover with sound systematic approaches. Due to the wickedness of the problems, one cannot say the an information system is right or wrong, only that it is good or bad. Moreover, it seems very difficult to give any general definition of what a good information system is.

But why does not other engineering disciplines experience the same problems? Take for instance house construction. Imagine that the customers wanted very flexible homes (as they probably would, given a realistic opportunity) which could do things such as extending or shrinking the dining room and its table according to the number of guests for dinner, calling an ambulance if its inhabitant suddenly becomes so ill that he is incapable of making the call himself, and changing the music played according to the mood of the inhabitants. The reason why the housing branch is not faced with loads of such requirements is that the customers understand that it would be very expensive. Moreover, to run such a flexible home, one would need... an information system!

This suggests that the wicked problems will always be pushed onto the information systems engineers, wherever they might occur. Consequently, information systems engineering will always be more difficult than other engineering, and strict systematic approaches will never become generally applicable. The software crisis will never disappear.

Still, software science is not pointless — undoubtedly, there are things that can be improved. Some difficulties are inherent in the nature of software engineering, others are due to bad methods. In the next section we will discuss what we believe to be the major weaknesses of current practice in IS development.

### 2.3 Weaknesses in the Current Practice

As stated in the previous section, some of the difficulties in IS development are more or less unavoidable, whereas others are due to bad practice. Most approaches make very rigid presumptions about how to view the real world, in that the language supports a limited set of concepts (for instance: entities, relationships; processes, stores, flows; objects, messages; places, transitions; states, transitions; tasks, agents, roles; etc.), and in that the method prescribes a rigid list of steps to be undertaken to develop a system. In some cases, such rigidity may be helpful, but obviously, the developer is forced to wear some blinkers when approaching the problem. Consequently, some aspects of the problem are emphasized, whereas others are neglected. The neglected ones might not always be the least important.

---

[20] calls the former essential difficulties, the latter accidents.
2.3. *WEAKNESSES IN THE CURRENT PRACTICE*

Thus, the presumptions of the approach might soon become inappropriate. The dangers of too rigid presumptions are recognized for instance in [76]:

- "... the system architect need an intuition for good design architecture, which is often defined by a variety of buzzwords... They provide a foundation and framework of concepts that can be used to evaluate designs and solutions, but... Trying to adhere to these concepts religiously can lead to schedule disruption and inability to solve problems not accounted for by the concepts."

We believe that the most common methodological weaknesses in current commercial IS development are the following:

- rigid phase-distinctions,
- target system orientation,
- modelling languages which are not general enough, and
- which provide too weak support for abstraction mechanisms.

In the following these four points will be discussed in more detail.

### 2.3.1 Rigid Phase-Distinctions

The conventional life-cycle model for IS development has been the so-called *Waterfall paradigm* ([96], [14], [1], [29], [15]) splits the process of developing information systems into phases. The Waterfall paradigm is illustrated in fig. 2.1, which has been adapted from [29]. According to [29] most standard methods for IS development follow some variation of the Waterfall paradigm, although there are a variety of names for the stages. As stated in [15], the Waterfall paradigm removed some of the major problems of the code-and-test and stagewise methods used earlier, the improvement from the stagewise methods being the introduction of feedback loops to correct the result of an earlier phase if found inappropriate later in the development. These major benefits of the Waterfall paradigm compared to previous techniques are listed in [29]:

- encouraging the specification of requirements before design,
- encouraging the planning of component interaction before coding,
- facilitating the management of the project by introducing check-points,
- leading to the generation of a series of documents that can later be used to help testing and maintenance,
- reducing development and maintenance costs due to the four reasons above, and
- enabling the organization that will develop the system to be more structured
Additionally, the Waterfall paradigm has an advantage also in that it is very widely used. Thus, system developers from different organizations can easily communicate by means of it.

In spite of its nice properties, and in spite of the progress made compared to the more ad hoc approaches used earlier, the conventional life-cycle model has received much criticism from the beginning of the 80’s, manifested by articles such as [73] (Life Cycle Concept Considered Harmful), [39] (Stop the Life-Cycle, I want to get off), and [7]. Weaknesses like the following have been pointed out:

- The phases are artificial constructs, one specific kind of project management strategy imposed on software development ([7], [73]). Actually the development of an IS is one continuous process, and since ISs are very diverse, one specific strategy cannot be appropriate for all projects.

- The conventional life-cycle model provides very little help for bridging the communication gap between analysts and end-users ([73], [15]). Thus, it works badly for interactive end-user applications.

- Rigidifying thinking the conventional approach tends to result in systems not very responsive to change ([73], [39]).

Moreover, it has been experienced that system development is very often late and more expensive than planned ([29], [39]), and when a software system is finally delivered, the user requirements might have changed so much that it is no longer satisfactory.
To mend the weaknesses of the conventional approach, new approaches have been suggested, such as evolutionary prototyping, automatic software synthesis [85], [8], rapid throwaway prototyping [40], [15], [73], incremental development [51], and software reuse [57]. However, these new approaches also have their weaknesses, as pointed out by Boehm in [15], stating that flexibility is necessary for being able to deal with the diversity of software development projects. His spiral model gives room for development along the lines of Waterfall, prototyping, automatic software synthesis, or combinations of these. The use of the spiral paradigm this far has shown promising results, although Boehm admits that there are some challenges which must be further addressed.

2.3.2 Target System Orientation

Another criticism that has been raised against traditional information systems development is that it is target system oriented, i.e. it concentrates very much on possible computerized solutions rather than on the actual needs of the customer. An example of this could be the approaches that use data modelling early in the analysis phase. A data model shows how information is going to be represented in the database, and information not supposed to go into the database would not be modelled. Thus, one only concentrates on what the computer does, not what the organization around the computerized information system is doing.

More recent works have called for a problem oriented approach, where the environment of the computer system is given as much attention as the system itself, the first goal of the development being the capturing of the objectives of the customer organization, regardless of whether these are going to be satisfied by the computer, by human beings or by both. These thoughts are supported for instance by

- the literature on ERAE ([32] [33] [46] [34]), particularly [46]: “Traditionally, RE (i.e. requirements engineering) focuses on the target system,... We call these requirements “computer-oriented”. Some of the dissatisfactions with computer systems may be traced to a basic weakness of this approach: it fails to pay enough attention to the environment of the system. Perhaps, the most revealing symptom is the users’ complaint that they were not made aware of the organizational changes required to run the system.”

- [98]: “Most of the software crisis is rooted in too narrow a focus on the software component. Instead, the entire system’s scope should be considered.”

- [67]: “...IS problems cannot be reduced within a short period of time. Most causes of problems, as our study shows, are social in nature, and their removal may require intricate learning and organizational changes, which are slow in coming.”

---

2This is the term used by [29]. [73] calls it simply prototyping, and [15] uses the term evolutionary development.
3Again the term used by [29], whereas [15] calls this the transform model.
4Boehm’s suggestion [15], called the spiral model, allows for the use of a series of throwaway prototypes, leading finally to full-scale implementation.
Undoubtedly, the environment or world oriented view is becoming more and more important, since software systems are applied to increasingly complex tasks, more and more in interaction with their users. As also stated in [98]: “The focus of software has shifted from efficient use of hardware resources to efficient use of human resources.” Thus, a good information system is only achieved through a successful cooperation between the computerized parts and their environment.

2.3.3 Concepts Which Lack Generality

Quite a lot of approaches and modelling languages have been developed in the field of information systems engineering. However, none seem to have achieved general acceptance, and the success in solving the difficulties of the discipline has been limited. The following observations can be made:

- most approaches lack generality — they are suitable for some kinds of problems and domains, but not for others

- the main reason for this seems to be that each approach has its specific blinkers. With this we mean the following:

  - The languages have constructs which force the analyst to perceive the real world in some specific way, for instance as objects communicating by means of messages, as processes, flows, and data stores, or as entities, relationships, and attributes. In other words, each approach forces the analyst to perceive the real world in some specific way, emphasizing some of its features, and neglecting others.

  - The more an analyst works with one specific language, the more his thinking will be influenced by this, and his awareness of the aspects of the real world that do not fit in might be diminished. Thus, any language can potentially have a kind of brain-washing effect.

  - For those kinds of problems for which the approach is suitable, the neglect of features that are not covered can have a positive effect, in that one more easily manages to concentrate on the relevant and important information. However, generally it is hard to know what information is relevant (due to the wickedness of information system problems). Thus, the neglect of certain features of the real world can be very dangerous.

Consequently, a language should be as general and flexible as possible. However, what are the general concepts in the world? This is a controversial philosophical issue which we do not want to discuss in any depth.

2.3.4 Lack of Abstractive Power

The discussion in the beginning of this chapter indicated that the major problem in information systems engineering is to deal with the enormous complexity. As
stated in chapter 1, humans overcome the complexity of the world by making abstractions. This suggests that an important feature of any language used for IS development should be its abstraction mechanisms. However, lots of widely used modelling languages provide amazingly little when it comes to abstraction mechanisms. This weakness is recognized by several authors, for instance

- [13]: "..., current languages and notations for software development are inadequate for supporting the abstraction paradigm, especially at the early stages: Particularly noteworthy was the absence of a wide-spectrum notation to avoid excessive translation steps in development."

Examples of widely used languages with poor support for abstraction are standard Petri nets [90], traditional state transition diagrams (see for instance [54]), the relational language [26], and the entity relationship language [24]. The experience of various researchers (for instance Harel [48] [49]) suggests that these languages would be more suitable for IS modelling if they were extended with some hierarchical constructs. This work is based on a strong belief that a systematic strengthening of the abstraction mechanisms provided within standard IS modelling might lead to major improvements of the practice.
Chapter 3

Objectives

3.1 Idealistic Objective: Pan-Presumptionism

3.1.1 Non-Presumptionism: a Dead-End Street

Flexibility in problem-solving is essential in IS engineering, due to the human factors involved. It is believed that by reducing this flexibility to an inappropriate extent, the presumptionism of many current approaches is an obstacle for progress in the field.

According to this belief we might go to the other extreme — an approach without any presumptions at all. However, this is impossible. Any method and any language implies some presumptions. Thus, having an approach totally free of presumptions would mean to have no approach at all, inventing a new one fit for the specific problem for every new development job. For philosophers this might be all right, but engineers are bound to adapt to certain demands for efficiency. Inventing a new approach for every development job would not give us that efficiency, neither is it likely that it will give us better end-products. Developing an information system without any fixed ideas about how it should be done would be tedious and unsystematic — as stated in [15] the ad hoc methods used in the earliest days of software development were much worse than the those used today. So clearly we need to make some presumptions, we need to have some fixed ideas. What we want is to find a point of balance — making enough presumptions for the approach to be systematic and efficient, but not so many that its flexibility is severely reduced.

Consequently, the essential question is the following: What presumptions should be made? We believe that one should try to make the most generally acceptable and most necessary presumptions first, whereas presumptions of a more limiting nature should be relaxed. Still, this is not a very substantial answer. What presumptions are necessary? What presumptions are general?
3.1.2 The Hierarchical Presumption

The basic philosophy of this work was stated in chapter 1. We believe that abstraction mechanisms are essential for dealing with the specification of information systems. Thus, supporting hierarchical constructs in our language is a necessary presumption, in that system complexity cannot otherwise be properly dealt with.

There are many kinds of hierarchies, so what hierarchical constructs would we want to include? Our requirement is that they must be general, so that they can be used to represent hierarchies everywhere, i.e. independent of the application domain, the level of detail, or the nature of the problem. The presumption made by introducing hierarchical constructs in the language should not indirectly imply lots of other presumptions, limiting the applicability of the language.

3.1.3 The Road Towards Pan-Presumptionism

So far you might have got the impression that we are very negative towards the presumptionistic attitude of current approaches, stating that they all tell lies. However, this is not the case; one could rather say that they all have the potential to express the truth, only different parts or aspects of it.

What we want is an approach which can cope with all these different aspects — an approach which is able to deal with any presumption. We will term such an approach pan-presumptionistic. At the moment there is no approach for IS development that can be called pan-presumptionistic. Neither will there be so by the conclusion of this thesis. Pan-presumptionism is a distant goal, a kind of research philosophy, rather than something which can be realized in a couple of years.

The truly pan-presumptionistic approach would have to provide a very powerful framework through which the user could define exactly the concepts that she finds appropriate, their syntax and semantics, and their external representation. The pan-presumptionistic tool would be a kind of CASE-shell, powerful enough to carry out routines to facilitate the verification and validation of the specifications, as well as giving guidance in the creative task of modelling. Moreover, it should have facilities for project management, version control, software reuse etc. — like everything else, these would have to be very flexible. This means that the tool support must be extremely powerful, by far exceeding anything on the market today.

3.2 More Realistic Objectives

As suggested above, pan-presumptionism (or something close to it) is a very ambitious goal which will definitely not be achieved in this thesis. We have to limit ourselves to something more realistic. The objectives of this work are as follows:

- We will concentrate mainly on language, not on method and tools.
3.2. MORE REALISTIC OBJECTIVES

- We will concentrate mainly on hierarchical constructs. This does not mean that we are going for a language unable to express anything else — to the contrary, the reason why we do not go into detail about other constructs is that we want as much generality as possible, the idea being that the framework should contain a base language allowing for the definition of any concept that the user might want.

- The idea, thus, is to arrive at a language framework featuring some general hierarchical constructs which can
  - be coupled to existing languages whose current support for hierarchies is found too weak, or
  - be the basis for a CASE-shell based approach in which these hierarchical constructs are provided as default, whereas other concepts can be defined by the user at request.

For this language framework we will consider both the conceptual issues, and the external representation, where we will emphasize the use of diagrams.

Not achieving anything which can be called pan-presumptionistic, we will at least try to avoid rigid presumptions, and hopefully, the language framework presented can be a little step on the road towards pan-presumptionism.\footnote{If asked "What good will such a language framework be?" we would have to admit that on its own, it cannot be used for developing any IS. But one should remember that Thomas A. Edison, when asked about the use of a new invention, often answered "What good is a baby?" [16]. This goes also for the language framework to be presented here: On its own, as it stands today, it does not give you much, but coupled to another, already existing modelling language, or extended with a CASE-shell for the definition of additional concepts, it might grow up to be something powerful.}
Part II

Preliminaries on Hierarchies
Part Introduction

This part provides some preliminaries on hierarchies.

Chapter 4 discusses the notions of hierarchy and hierarchical relation. Some graph-theoretic definitions are provided, and some general set-theoretic hierarchic constructs are identified.

Chapter 5 discusses different ways of structuring the knowledge in information systems modelling, illustrating how the same knowledge can be represented in different ways according to what kind of concept in the language which is treated as the most central.

Chapter 6 describes the idea of constructivity, the ability to check the consistency between different hierarchical levels in some model. Supporting constructivity is essential to any method heavily based on hierarchical decomposition, and here we summarize the experience made from work that we have done on constructivity for some specific modelling languages.

Chapter 7 takes a look at several current modelling languages, investigating what hierarchic constructs they support, and how these are represented externally. In particular, the survey focuses on diagrammatic representation of hierarchical constructs.
Chapter 4

Basic Hierarchy Concepts

This chapter defines some basic hierarchical concepts which we will need in the rest
of the work. First we discuss what a hierarchy is. Then we discuss some general
constructs on which a language for hierarchies can be based.

4.1 What Is a Hierarchy?

In this section we will discuss what a hierarchy is. The first subsection discusses
the possibilities for arriving at a precise definition of “hierarchy” in terms of graph
theory. As will be seen, however, it is difficult to come up with a definition which
is precise and at the same time satisfactory to our very general purposes. The
second subsection thus argues that being hierarchical is very much a question of
degree.

4.1.1 Hierarchical: A Question of Definition

In [99] [42] a hierarchy is defined rather vaguely as any collection of Chinese boxes
(wher each box can contain several smaller boxes). [75] refrains from giving any
exact definition of what a hierarchy is, but lists some properties which all hierar-
chies should have, namely “vertical arrangement of subsystems which comprise the
overall system, priority of action or right of intervention of the higher level subsys-
tems, and dependence of the higher level subsystems upon the actual performance
of the lower levels. More precise definitions are given in [5] and [22].

Some works also identify different kinds of hierarchical systems. [5] distinguishes
formally between division hierarchies and control hierarchies. [75] operates with
three notions of hierarchical levels, namely strata (levels of description or abstraction),
layers (levels of decision complexity), and echelons (organizational levels).
All in all it seems that the word “hierarchy” may be used in rather different ways
by different authors — as stated in [104] some use it indiscriminately for any partial
ordering, whereas the above definitions require something more.

Having very general purposes, we agree with [75] that it is difficult to come up
with a strict and precise definition distinguishing hierarchical systems from other systems. However, we also need to make clear what we are talking about. Thus, we need some kind of definition of what a hierarchy is.

To this end it is very illuminating to look at the definition presented by Bunge in [22] (quotation):

\[ H \text{ is a hierarchy if and only if it is an ordered triple } H = <S, b, D> \]

where \( S \) is a nonempty set, \( b \) a distinguished element of \( S \) and \( D \) a binary relation in \( S \) such that

- \( S \) has a single beginner, namely \( b \). (That is, \( H \) has one and only one supreme commander.)
- \( b \) stands in some power of \( D \) to every other member of \( S \). (That is, no matter how low in the hierarchy an element of \( S \) may stand, it is still under the command of the beginner.)
- For any given element \( y \) of \( S \) except \( b \), there is exactly one other element \( x \) of \( S \) such that \( D_{xy} \). (That is, every member has a single boss.)
- \( D \) is transitive and antisymmetric. (Togetherness but no back talking.)
- \( D \) represents (mirrors) domination or power. (That is, \( S \) is not merely a partially ordered set with a first element: the behaviour of each element of \( S \) save its beginner is ultimately determined by its superiors.)

(end quotation)

As pointed out by Bunge, this definition does two things:

- The first four points state what a hierarchy is in a graph-theoretic sense, namely a strict tree-structure.\(^1\)
- The fifth point introduces an extra requirement on the nature of the relations (i.e. edges) between the nodes, namely that they represent domination or power.

Thus, Bunge makes the important point that whether something is a hierarchy or not cannot be determined by graph-theoretic considerations alone, which is in accordance with our own opinion. However, Bunge's definition might be a little too strict to our purposes:

- the graph-theoretic demands are very limiting. In real life it often happens that a node can have more than one boss, or even that there are cycles in the

\(^1\)To be precise, it is an open-ended directed graph whose underlying graph (i.e. the undirected parallel of a directed graph) is a tree, since trees, graph-theoretically, are undirected graphs. For an introduction to graph theory, including definitions of graphs (directed and undirected), trees, and underlying graphs, see for instance [121].
4.1. WHAT IS A HIERARCHY?

graph, and still many people might consider the system to be of a hierarchical nature.

- the requirement that nodes are related by domination severely limits the scope of hierarchical systems — as stated by Bunge himself reciprocal action, rather than unidirectional action, seems to be the rule in nature (which leads Bunge to the conclusion that it is misleading to speak of hierarchies in nature: "Hierarchical structures are found in society, e.g. in armies and in old-fashioned universities; but there are no cases of hierarchy in physics or in biology"). Since we want to be able to model practically anything, we have to recognize other kinds of hierarchical relations in addition to domination or power.

To achieve more generality, we will allow more general graphs to be considered as hierarchical systems. But it will also be useful to have a specific term for those systems which satisfy the rather restrictive requirements stated above. In this work we will use the following terminology:

**strictly hierarchical graph**: a digraph whose underlying graph is a tree, and for which there is one specific vertex b from which all other vertices can be reached (this is the distinguished element of Bunge's definition).

**weakly hierarchical graph**: a connected acyclic digraph which deviates from the former in that there is no distinguished element and/or in that its underlying graph is cyclic. Mathematically, this class of graphs are called DAGs (directed acyclic graphs).

**cyclic hierarchical graph**: a cyclic digraph.

Obviously, the latter two notions should be used carefully — there is no point in calling any graph a hierarchy. Thus, even if we allow some DAGs, and maybe even some cycles, we should still require that a graph is pretty close to being a strict hierarchy if calling it hierarchical.

The meaning of our suggested terminology can be visualized by fig. 4.1. Of these graphs (a) would not be a hierarchy because it is not connected (but it might be two hierarchies), and (b) would not be a hierarchy because the edges are not directed. (c) on the other hand, is the kind of graph which satisfies Bunge's requirements, i.e. it is a strict hierarchy. (d) would not be accepted as a hierarchy according to Bunge's definition because the underlying graph has a cycle (i.e. the middle element at the lowest level has two bosses), but we might call it a weak hierarchy. Similarly, (e) does not have one distinguished element — there are two elements on top which do not control each other. This could also be a weak hierarchy in our terminology. Finally, (f) contains a cycle and is thus clearly excluded by Bunge's definition, whereas we could call it a cyclic hierarchy (because although containing a cycle, the graph is not very far from being a strict hierarchy).

The motivation for removing some of the restrictions of Bunge's definition is that we want to be as general as possible, and clearly many people might feel that sys-
tems are hierarchical even when they are not strictly hierarchical. This is exemplified by the two graphs of fig. 4.2, where (a) breaks the single boss requirement, and (b) breaks the antisymmetry requirement. If the edges denote the relation like "is the boss of"\(^2\), it is still likely that both systems will be considered as hierarchical. Moreover, our definition has not required that the relations denoted by the edges be transitive. Clearly, most hierarchical relations are transitive (e.g. if A is the boss of B, and B the boss of C, it also makes sense to say that A is the boss of C), but there is no point in rejecting cases where this does not apply (e.g. if A is the parent of B, and B the parent of C, it will not make sense to say that A is the parent of C, and still people might feel that "parent of" is a typically hierarchical relation).

Having loosened up Bunge’s graph-theoretic restrictions it might seem that we may end up calling any kind of directed graph a hierarchy. However, this is not our intention. We still need some requirement corresponding to the fifth point of Bunge's definition. *Dominance or power* is obviously too narrow for our purposes. Still we need to make some restriction on the semantics of the relation denoted by the edges. This is not easy, and can only be dealt with when we have discussed in

\(^2\)in (b) Bo and Dan might for instance supervise two different business areas, both working on both.
the next subsection what it means to be more or less hierarchical.

4.1.2 Hierarchical: a Question of Degree

If one takes everything into consideration, a description of a real world situation will be a general graph rather than a hierarchy. Depicting something as a strict hierarchy will therefore be a simplification of the reality, and this simplification may be more or less appropriate, depending on the distance between the hierarchy depicted and the actual situation.

How can such distance be measured? Given a general connected digraph, how would you answer the question “How close is this graph to being a hierarchy?”? From a general connected digraph, a strict hierarchy can be obtained by cutting some edges, so a first attempt could be to count how many edges one would have to cut, or rather the ratio of cut edges to the total number of edges. With this approach we would say that the digraph of fig. 4.3(a) is obviously closer to being hierarchical than the one of (b), since in (a) we have to cut only 2 out of 10 edges, whereas in (b) we have to cut 4 out of 10. However, the soundness of this kind of computation relies on the assumption that all links are equally important, which need of course not be true. To deal with other cases, each edge must be assigned a weight, signalling its importance. In fig. 4.4 weights have been assigned, and now it is (b) which is closest to being a hierarchy, because the minimum cut has a total of only 5 weights, whereas the same number for (a) is 12.

If the given relation is “is the boss of” weights should reflect the degree of influence that the supervisor has over the subordinate; if the graphs are call graphs for a software system, importance will depend on the frequency of calls. Generally, importance is a very problematic notion, and we will not enter any further discussions of it in this work. Instead we will only conclude that:

- a general directed graph can be more or less hierarchical, and
CHAPTER 4. BASIC HIERARCHY CONCEPTS

Figure 4.4: Two weighted digraphs

- its closeness to a strict hierarchy is dependent on:
  - the structure of the graph, and
  - the importance of individual connections.

Being hierarchical is thus a question of degree. Not only specific graph models of the real world can be evaluated according to this; we can also compare different kinds of relations. Obviously, some relations, like “is the boss of”, tend to result in rather hierarchical graphs, whereas for instance “loves” is not likely to do so. In fig. 4.5, (a) is a plausible picture of “who is the boss of who in Dept. X” and (b) is a plausible picture of “who loves who in Dept. X”. As can be seen, (a) is almost a strict hierarchy, whereas (b) is not even connected (and thus very far from being a hierarchy). That the “who loves who in Dept. X” should form a hierarchy, like in (c), seems pretty unlikely because a relation like “loves” is inherently non-hierarchical (as opposed to for instance “is the boss of”). Consequently, even if the situation in (c) occurred, one might not feel that this is a hierarchy. Thus,

- Some kinds of relations are hierarchical of nature, and others are not.
- For the former it might be interesting to simplify the presentation of some knowledge by cutting edges to obtain hierarchies.
- For the latter, it seems that such an approach would make no sense, as it would be confusing rather than enlightening to present them as hierarchical.

Still, we have basically only made it clear that we want to deal with more situations than what falls under the rather strict definition of Bunge — in fact we want to be able to deal with almost any situation where something like a hierarchical abstraction construct occurs. In the next section we will identify some constructs which are necessary in a language with such ambitions.
4.2. *FOUR STANDARD RELATIONS*

4.2.1 Set-Theoretic Definitions

There is a vast number of hierarchies that one might want to model, and these have rather diverse properties. Imagine organization hierarchies, definition hierarchies, goal hierarchies, file system hierarchies, operating system process hierarchies. Trying to capture the specific properties of all these, one could easily end up creating a hierarchical language just as complicated as the real world itself. Instead, we must try to identify some *general* hierarchical relations which can contribute to the structuring of many different kinds of knowledge.

By and large, such relations have already been found. Work in the field of semantic data modelling ([55], [88], [91]) and semantic networks ([36]) has lead to the identification of four standard constructs:

- classification,
- aggregation,
- association, and
- generalization.

Since this list of four items, as well as the list of the latter three, might have to be mentioned quite often in the rest of this work, we define the abbreviations

\[ \text{CAGA} \overset{\text{abr}}{=} \text{classification, aggregation, generalization, and association}; \]

\[ \text{AGA} \overset{\text{abr}}{=} \text{aggregation, generalization, and association}. \]
The four constructs have the following meaning [91]:

**classification**: specific instances are considered as a higher level object type via the *is-instance-of* relationship (for example, “Rod Stewart” and “Mick Jagger” are specific instances of “singers”).

**aggregation**: an object is related to the components that make it up via the *is-part-of* relationship (for example, a bicycle has wheels, a seat, a frame, handlebars etc.).

**generalization**: similar object types are abstracted into a higher level object type via the *is-a*³ relationship (for example, an employee is a person).

**association**: several object types are considered as a higher level set object-type via the *is-a-member-of* relationship (for example, the sets “men” and “women” are members of the set “sex-groups”). Association is also likely to be encountered under the names of *membership* (for instance in [91]), *grouping* (for instance in [55], or *collection* (for instance in [45]).

It can be noticed that *classification* is sort of orthogonal to the other three — whereas the others construct bigger things from smaller things (on the same meta-level), classification results in a shift of meta-level, in accordance with the philosophical notions of *intension* and *extension* ([23], [31]). The *intension* of “man” is the property of being a man, whereas the extension of “man” (in any specific world, at any specific time) will be the set of all existing men (in that specific world, at that specific time). Going one meta-level higher from “man”, one can get to “species”, of whose extension “man” is a member (in this particular world, at this particular time). One does not have to go much higher until there are only very abstract notions like “words” and “concepts”, so it is of limited interest to use many meta-levels in a model. Moreover, meta-level reasoning requires a very powerful language, as the one presented in [11].

For the other three constructs, the complicated notions of intension and extension are unnecessary, and rather straightforward set-theoretic definitions can be provided:

- **aggregation** corresponds to the *Cartesian product*: If the set A is said to be an aggregation of the sets A₁, ..., Aₙ this means that A ⊆ A₁ × ... × Aₙ, i.e. each element of A consists of one element from each of A₁, ..., Aₙ.

- **generalization** corresponds to *union*: If the set A is a generalization of the sets A₁, ..., Aₙ, this means that A ⊆ A₁ ∪ ... ∪ Aₙ.

- **association** corresponds to *membership* (i.e. embracing by set brackets): If the set A is an association of the sets A₁, ..., Aₙ this means that A = { A₁, ..., Aₙ }.

However, as indicated by [55], some works may use these terms somewhat differently:

³There are a few authors that use “is-a” for classification...
4.2. FOUR STANDARD RELATIONS

- some languages (like for instance SDM [47] and TAXIS [77]) represent aggregations by means of attributes (instead of cross product type construction).

- some languages have identified several kinds of generalization, for instance the language defined in [111] provides one is-a relationship called subset and one called generalization. The latter is used when the supertype is totally partitioned by its subtypes.

- it is often useful to define association in terms of the powerset operator. As suggested both by [55] and [88], association is commonly used for constructing sets of objects of the same type. Consider the example of fig. 4.6 (taken from [55]), where the *-node denotes the association of the “person” node, meaning that the former is a subset of the powerset of the set of persons (i.e. each committee will have some group of members taken from the set of persons). Since we do not want to express at an abstract level the exact members of each committee, and since all members are persons, the association operator will have only one child in this case (whereas “men” and “women” being members of “sex-groups” earlier in this chapter signalled a use of association with several children).

However, these differences are merely of a representational kind and do not give us any problems, as will be shown in chapter 9.

4.2.2 How Good are the Suggested Relations?

Strengths

As indicated by [55] [88] [91] quite a lot of modelling languages provide at least some of the CAGA constructs, and the effects of introducing such constructs are positively described. [88] reports improvements in expressive economy, integrity maintenance, modelling flexibility, and modelling efficiency. But why is it that languages tend to predefine exactly CAGA and not any other hierarchical relations (like “is the boss of”). The reason is obviously their
• *generality* and
• *intuitivity*.

The generality of CAGA can be accounted for by the fact that they are substantial. Whereas relations like "is-the-father-of" and "is-the-boss-of" contain substantives "father" and "boss", whose semantics clearly limit the applicability of the relations, "is-part-of" uses the semantically very anonymous substantive "part". Anything can be a "part" of something — the set of potential fathers is much more limited. The substantives "instance", "subset", "member" are similarly weak in semantic content. Defined in terms of sets, with no commitment as to what these sets shall contain, these abstraction mechanisms should be able to cover any application area. Thus, they can be useful in organization modelling, process modelling, data modelling, hardware modelling, and so on, and so forth — which was exactly what we wanted.

Moreover, CAGA are obviously very intuitive abstraction mechanisms, which must be why they have become so popular in the first place. We find it natural to think of things as being put together from smaller parts (aggregation), as being of a specific type (classification), as being members of groups (association) which can have smaller subgroups (generalization). People are even being trained pretty much in using such hierarchies in school, for instance learning languages (aggregation: assembling words from letters, sentences from words, etc., classification: distinguishing between word classes, identifying phrases as subject, predicate, direct object, indirect object etc., generalization: different kinds of sentences, substantives, verbs etc., association: memorizing lists of prepositions demanding a certain case in German), learning biology, learning mathematics — whatever!

**Weaknesses**

However, there are also some weaknesses to be mentioned:

• The set-oriented definition of CAGA cause some limitations on their use.

• Also, there are hierarchies which are certainly of interest in information systems modelling which are not covered by the CAGA scheme.

As can be seen from the set-theoretic definitions given in this chapter, classification means to move up one meta-level, from an instance to a type. The other three are set-level constructs. Thus, there are two problems:

• What do we do about instances?

• What do we do about masses?

**Instances** are not necessarily such a big problem. The association construct is trivially applicable, since it can produce a set of instances just as easily as a
set of sets ("Peter, Patricia, and Joey are members of the Party Committee"). Moreover, if we treat instances like sets with only one member (like Quine does in [92]), the definition of aggregation just presented is also trivially applicable ("The car # 346 was constructed from the chassis # 9213, the carrossery # 2134, and the engine #905"), and so is generalization (with the limitation that it only seems to be useful in situations where the general notion is a variable: "Joey's murderer must have been either Peter or Patricia", in which case "Joey's murderer" can be said to be a generalization of "Peter" and "Patricia").

Another question is hierarchical relations between instances (like "father-of", "boss-of"). It is difficult to know which instance level relations people might want, and we cannot define an enormous amount of them in advance. The wisest thing for a general framework might be to provide a generic relation construct from which the users can define all the relations that they need. This is addressed in more detail in chapter 9, and in chapter 14 we will look at some hierarchical relations between rules (which are, of course, instances).

b. Mass Concepts As Sowa points out in [105] the set-oriented way of thinking which permeates so many information systems models work well for things that are countable, whereas there are problems for the so-called mass nouns, like water, love, money. Again it appears that the notions of AGA are applicable ("Chocolate is made of cocoa, sugar and milk" (aggregation), "Milk and water are both liquids" (generalization), "Milk and Water are members of the set Liquids" (association)). However, the problem is that we cannot use the set based definitions presented earlier in this chapter. There are two possible ways out of this:

- We can go for a more general definition of AGA which is not at all based on sets (but for instance on types).

- We can use the definitions already suggested and add some special tactics for dealing with masses.

This question will be further discussed in chapter 9.

4.3 Hierarchies and Pan-Preumptionism

As we said in the previous chapter, language is presumption. So how can we define a language of hierarchical constructs and still call our approach pan-preumptionistic? We will argue as follows:

- As indicated in this chapter, the hierarchical relations considered are very general. They only consider sets, with no commitment as to what these sets contain.

- Concentrating on sets is of course a presumption in itself. However, we have promised to support also instances and mass concepts.
• Finally, we have argued that abstraction mechanisms are essential to the human perception of complex system. If hierarchies are inevitable, it is not supporting them which is the most inappropriate presumption that a language can make.

Still, of course, we need to ensure that the hierarchical constructs defined are as general as possible, not excluding any particular way of perceiving the world.
Chapter 5

Structuring Principles

5.1 Structuring Principles and Aggregation Principles

As we have already stated, it is of major importance to be able to structure the knowledge in some way when modelling complex systems. It might be advantageous to achieve a kind of structuring which is somehow uniform throughout any model. To do this, one needs a structuring principle.

Generally, we define a structuring principle to be some rule or assumption concerning how knowledge should be structured. This is a very vague definition — we observe that

- a structuring principle can be more or less detailed: on a high level one for instance has the choice between structuring the information hierarchically, or in a general network. Most approaches take a far more detailed attitude towards structuring: deciding what is going to be decomposed, and how. For instance, structured analysis implies that the things to be decomposed are processes (maybe also stores and flows), and an additional suggestion might be that the hierarchy of processes should not be deeper than 4 levels, and the maximum number of processes in one decomposition 7.

- a structuring principle might be more or less rigid — in some approaches one can override the standard structuring principle if one wants to, in others this is impossible.

Since this thesis concentrates on hierarchies, our interest here is of course directed towards structuring principles which have a clearly hierarchical nature. Moreover, since we are more into language than method in this thesis, we are more occupied with the question of what can be decomposed, than the question of how it should be done. Consequently, the rest of this chapter will basically discuss what we call aggregation principles.

As stated in the previous chapter, aggregation means to build larger components
of a system by assembling smaller ones. Going for a certain aggregation principle thus implies decision concerning

- what kind of components to aggregate
- how other kinds of components (if any) will be connected to the hierarchical structure

Fights between the supporters of different aggregation principles can often be rather heated. As we will show, the aggregation principle is a very important feature of an approach, so this is very understandable. Some possible aggregation principles are the following:

- object-orientation
- process-orientation
- agent-orientation

Objects are the things subject to processing, processes are the actions performed, and agents are the ones who perform the actions (i.e. either organizational units, persons\(^1\), or machines). Clearly, these three approaches concentrate on different aspects of reality, but it is easy to be mistaken about the difference\(^2\). The following points can be made:

- The difference between different orientations is not necessarily that they use different concepts. All of the above three could potentially support all the concepts object, process, and agent.
- Neither does the difference have to be that they model different kinds of knowledge. Modelling a piece of the real world in all three, it might happen that all models represent exactly the same knowledge. It is not unlikely that it would not, but our point here is that the main distinction between the orientations does not lie here.
- The difference is one of structuring (as pointed out for instance in [6] on the difference between object-orientation and process-orientation), i.e. the three approaches structure the knowledge (which might be the same) differently.

For an illustration of this, consider fig. 5.1. Here we have a model with 9 agents (A1–A9), 9 objects\(^3\) (O1–O9), and 9 processes (P1–P9). To make things as comprehensive as possible we simply assume that A1 is responsible for P1 which is performed on O1, A2 for P2 performed on O2 and so on.

Fig. 5.1(a) shows an object-oriented structuring. The following points can be made:

\(^1\)or animals, or maybe even plants...
\(^2\)At least I used to be, some years ago.
\(^3\)Actually, the term object means an ADT plus the operations that it owns. In this example, the objects are only ADTs (since the operations are covered by the process concept).
Figure 5.1: Three aggregation principles
• Basically, only objects are decomposed. This means that the higher level composite processes P1–P3, and the higher level composite agents A1–A3 might not be shown, which has been indicated by parentheses and dotted lines.

• Every process must be connected to some specific object, the one that owns it (i.e. the object that it works on).

• Agents (if at all included) should be connected to the objects that they are responsible for.

• Consequently, the object structure determines the structure of the overall model, with the following results:
  
  – Mother-daughter relationships between objects will be emphasized, whereas similar relationships for processes and agents will be suppressed.
  
  – Communication between agents and/or processes must be shown in other ways. If the representation language is partly diagrammatic, this would probably mean that relations between objects will be possible to draw, whereas relations between agents and/or processes only have a textual representation.

All in all, one can conclude that objects (i.e. the items on which the work takes place) are given higher importance than the actual actions themselves, or the ones performing the actions.

The process-oriented version is shown in fig. 5.1(b). The situation is the same as for object-orientation, only that now it is processes that are decomposed. Agents are attached to the processes they perform, and objects to the processes that work on them. The agent-oriented version of fig. 5.1(c) emphasizes the agents similarly: processes and objects are connected to the agents responsible, and only agents are decomposed.

All in all, each approach emphasizes some specific aspect of reality at the cost of others. However, we said above that the same information could be represented in any orientation, so what is the practical impact of the structuring difference?

We believe the practical impact to be the following:

• A natural way of modelling is to start by getting down the basic structure, then afterwards adding more information to this structure.

• Consequently, choosing one specific aggregation principle will also very much determine the order in which things are to be done during system specification. For instance, if using an object-oriented approach, one will more or less be forced to specifying the objects first, adding other things to these later.

• Since specification cannot be decoupled from knowledge acquisition, the choice of aggregation principle will thus be important for the task of validation, which many believe to be the key problem of information system engineering.
5.2. Single and Multiple Principle Approaches

In the previous section we only gave examples of approaches using one aggregation principle, structuring all the knowledge in one hierarchy. However, this is not the only way of doing things. It is also possible to entertain several aggregation principles in the same approach, and in fact, this is a pretty ordinary thing to do. The most common example is approaches which supply one language for modelling statics and another one for dynamics. If at all hierarchical, these will be likely to support one static hierarchy (which might be called entity-oriented)\(^4\) and one dynamic (which is likely to be process-oriented).

Taking a multiple principle view, the example in the previous section can be redone with three hierarchies: one for objects, one for processes, and one for agents. This is shown in fig. 5.2. With this approach, non-hierarchical information (like which agent performs which process, which object does the process work on) must be encoded as links between the different hierarchies — these are depicted as dotted lines in the figure.

Obviously, both approaches have their pros and cons. The following arguments can be made in favour of a single hierarchy approach:

- All the information is structured according to one hierarchy. This might make things simpler, both for tools and users.

- Often, the goal is to arrive at the lowest level specification (upon which

\(^{4}\)To call a something like a hierarchical ER model object-oriented would be too dangerous, since object-orientation usually means something more: encapsulation, functional completeness etc.
higher levels might even be thrown away). In this case, it might be pointless
to support more than one decomposition hierarchy.

However, an important drawback, at least from our pan-presumptionistic view-
point, is that

- using only one hierarchy, we have to emphasize one specific aspect of reality
  at the cost of others.

whereas

- using several hierarchies, we can be more open to several aspects of reality.
  Also, we avoid the problem of having to do things in one specific order
  (which, as argued in the previous section, often permeates single hierarchy
  approaches).

However,

- having several hierarchies might be far more chaotic, especially since we also
  have to be able to define relations between components of different hierar-
  chies. Undoubtedly, the increased flexibility comes at a cost — namely that
  the approach requires a much stronger tool support to be practical.

Since the systems we need to deal with are usually very complex networks rather
than neat hierarchies, a hierarchical presentation necessarily implies some major
simplifications. Often it is stated that the important thing is not exactly how
things are structured, only that they are structured in one way or another, and
that this structuring is understandable to those involved.

### 5.3 Data-Orientation and Real World Orientation

Some languages are made for modelling data. For some other languages, it is
claimed that they can be used for modelling the real world. Often it is rather
vague what this means, but obviously, the difference between data-modelling and
real-world-modelling has some relevance to our discussion of aggregation principles.

The following points can be made:

- Data-orientation and real world orientation are not simply additional alter-
natives to object-orientation and process-orientation. Rather these categories
  are orthogonal to the previously mentioned — an object oriented approach
  can take either a real world point of view or a data modelling point of view.
Data-orientation and real world orientation are not aggregation principles according to our definition earlier in this chapter. The distinction decides what is going to be modelled, rather than how it is going to be structured. In fact, data orientation or real world orientation in its own respect does not imply any structuring at all.

However, the choice between data orientation and real world orientation will necessarily have an impact on how the different hierarchical might be used — this is most heavily experienced for aggregation in static models.

The difference in the use of aggregation can be summarized as follows:

In a data model, an object is considered as identical to the information about it. Thus, if the relevant information about an employee is his social security number, name, address, position, and salary, it would be said that employee is an aggregation of these properties. Similarly, if there is a binary relation "is the boss of" between managers and ordinary employees, one might say that the relation is an aggregation of the two (since each instance of it might be uniquely defined by its manager instance and employee instance. All in all:

- modelling data one might regard an object as an aggregation of its properties, and
- relations as aggregations of the things they relate.

Modelling objects in the real world, on the other hand, it would be awkward to consider human beings as aggregations of numbers and names. One would rather say, for instance, that a human being consists of a mind and a body, the body again consisting of different parts: head, arms, legs, torso... However, names and numbers must also be modelled. Thus, one must be able to distinguish between an object and the information about this object. All in all:

- modelling the real world one cannot regard an object as an aggregation of its properties — however, the information about an object (which is itself an object) can be aggregated from these.
- Thus, one must be able to distinguish between an object and the information about it.

The conclusion is that a construct which is considered an aggregate of certain others when information objects are concerned, might not be considered likewise when the actual real world objects are concerned.

5.4 Structuring and Pan-Presumptionism

Some people might be in favour of approaches supporting several aggregation principles at the same time. Others may prefer one particular principle, for instance
the object-oriented one. As stated above, both have their advantages and disadvantages, and it may be difficult to make a choice. However, in this work we do not need to make a choice. The idea of pan-presumptionism is just the opposite:

- Our framework should work both for data modelling and real world modelling.
- Our framework should be able to support any kind of aggregation principle.
- Our framework should be able to support any number of different hierarchies in some specification.
- Moreover, it must allow for the specification of relations across the basic hierarchical structure, both within some specific hierarchy and between different hierarchies.

Thus, our framework can be used by those who prefer an object-oriented structuring as well as warm supporters of functional decomposition, by people following a single hierarchy approach as well as those using multiple hierarchies. Most of this chapter has been about aggregation, but of course a similar flexibility will be strived for when the other hierarchical constructs are concerned.

One particularly interesting feature for a flexible approach would be the ability to translate automatically between different ways of structuring the knowledge. Clearly, if enough information has been encoded, this is only a matter of different ways of presentation. The possibility of seeing things from several different angles can undoubtedly be very helpful to the essential task of validation. It is more problematic to provide any sensible restructuring if important information is missing — for instance, it will often be difficult to transform a purely object-oriented specification to a purely process-oriented one and vice versa, because one has omitted some links that the other needs. To make such translations possible the person writing the specification might be forced to introduce some links which are felt to be specificational overhead (which will of course be unpopular).
Chapter 6

Constructivity

6.1 The Fundamental Principle

The notion of constructivity was brought into the field of information systems engineering by Langefors [61], who defined what he called the fundamental principle of systems work. This principle amounted to the following (quotation):

Partition the systems work into separate parts, a through d:

a. Definition of the system as a set of parts:
   List all parts from which the system is regarded as built-up.

b. Definition of the system structure:
   Define all interconnections which make up the system by joining its parts together.

c. Definition of the systems parts:
   For each single part (or group of similar parts) separately define its properties as required by the system work at hand and do this in a format as specified by the way the systems structure is defined (in task b).

d. Determination of the properties of the system:
   Use the definitions as produced by the tasks a, b, and all separate tasks c, all taken together. Compare with specifications wanted for the system and repeat a, b, c, and d until satisfied.

(end quotation)

It is the point d in the list above that involves constructivity. Basically, it means to derive the properties of a system based on the properties of its subsystems, and then check if the derived properties are the same as those specified for the system earlier (if any). The derivation can be called abstraction, and a specific subsystem structure is said to be constructive if such an abstraction is possible. Constructivity is necessary when we want to check the consistency of a hierarchical specification, i.e. to check whether decompositions are correct.

Some other important points made by Langefors are the following:
• The principle divides tasks into two sets: one concerned with the whole system (a, b, d) and one concerned with its parts (c, where each part can be treated separately).

• Hence it gives a natural way for dividing the work among several people (which is essential for large, complex systems).

• The principle works equally well for top-down and bottom-up development.

• The principle is most efficient when it can be mathematically formalized, but one should stick to it also when this is not possible.

Although the ideas of Langefors are more than 20 years old, little has happened when it comes to automating the fundamental principle in commercial information systems development tools. Considering the enormous potential such an automation would have, it seems natural to believe that the main reason for this is the tremendous difficulty of the task. Some major problems are:

• The principle requires that we are able to decompose the system into independent subsystems (which can then be dealt with separately).
  
  – Due to the complexity of the world, it is very hard to fulfil this requirement, and independence can never be completely verified (since it is always possible that we have forgotten some connection).

  – Moreover, interconnections between the parts on high levels of abstraction will be very complex, so that it is impossible to perceive them in any sensible way. This point is specifically made by Bubenko [21], who states that one should not care about specifying interconnections at higher levels when doing top-down specification. The goal should rather be to arrive at a set of business activities, as complete as possible, which can then be the basis for bottom-up design.

• The formal languages sufficiently expressive for the specification of complex systems are not fully decidable. Thus, consistency checking might be impossible, hierarchical or not.

• Finally, constructivity is merely hierarchical verification. Although important enough, this means that the fundamental principle does not address (at least not directly) the problem of validation, which is equally crucial to successful system development.

Even though the criticism raised concerning the applicability of the fundamental principle may be correct, this does not mean that constructivity is not a nice feature to have. Obviously,

• languages and methods that tend to give constructive subsystem structures are highly preferable to those that do not, and

• any serious approach based on hierarchical decomposition should have some automated support for constructivity built into its specification tools.
For a more concrete illustration of the possibilities and problems connected to constructivity, we summarize some personal experiences from previous research in the next section.

6.2 Constructivity in BNM and PPM

6.2.1 BNM

BNM (Behaviour Network Model, [60]) is a language for describing information system structure and behaviour — an example diagram is shown in fig. 6.1. The language uses Sölvbergs Phenomenon Model [107] as its static part, coupled with an extended Petri net formalism for dynamic modelling. This coupling is shown by edges between places in the Petri net and phenomenon classes. The token of a place can either be an element of a phenomenon class (the edge is annotated with \( "\in" \)) or it can be the whole class (the edge is annotated with \( "=\)\).

The Petri nets of BNM differ from standard Petri nets in that

- tokens are named variables. Class variables have capital letters and element variables have small letters.

- there are two kinds of input places to a transition: consumption places and reference places. For the former, a token is consumed when a transition fires, whereas the latter is not consumed.

- transitions are allowed to take time.\(^1\)

- transitions have pre- and postconditions in predicate logic. For a transition to fire, its precondition must be true, and by the firing its postcondition will become true.

Otherwise, the BNM semantics are in accordance with standard Petri net semantics.

An algorithm for abstracting a network of BNM transitions to one higher level transition whose pre- and postconditions would be the conditions of the network as a whole (i.e. performing Længefors' step d above) was outlined by Kung in 1986 (as accounted for in [60], chapter 16, sections 5 and 6) and elaborated further in [100].

The algorithm goes through these three main steps:

1. find all possible transition sequences

2. propagate variable updates along each such sequence to eliminate variables of internal places from the expression of external output values

\(^1\)The early reports on BNM operate with zero time transitions, but as mentioned in ch. 7 it is necessary to allow transitions to take time if they are going to be decomposed. This is also assumed in the BNM simulation tool RAPTUS.
3. build higher level network based on 1 and 2.

To illustrate the algorithm in some more detail, we give an example. The network of fig. 6.1(a) describes the detailed behaviour for handling a batch of orders. For the sake of simplicity we have not shown the attributes of the phenomenon classes. These are indicated by the ordinary point notation following variables in the conditions of the transitions:

**t1:** pre: \( m = "\text{Handle Orders}" \)
post: \( \diamond X = W \)

**t2:** pre: \( X = \emptyset \)
post:

**t3:** pre: \( (\exists x)(\exists s)(x \in X \land s \in S \land x.P\# = s.P\# \land x.Q \leq s.Q) \)
post: \( \diamond X = X \setminus \{x\} \land \diamond s.Q = s.Q - x.Q \land y.C\# = x.C\# \land y.P\# = x.P\# \land y.Q = x.Q \)

**t4:** pre: \( (\exists x)(\exists s)(x \in X \land s \in S \land x.P\# = s.P\# \land x.Q \leq s.Q) \)
post: \( \diamond X = X \setminus \{x\} \land z.C\# = x.C\# \land z.P\# = x.P\# \land z.Q = x.Q \)

The network with its conditions can be explained as follows:

- When the message "Handle Orders" is given, the list of orders to be processed is copied from the variable \( W \) to the variable \( X \) (\( t1 \) does not consume \( P2 \) because \( P2 \) is only a reference place, as indicated by the dotted line).

- As long as there are still orders to process, an arbitrary order will be picked and processed. If it can be satisfied (i.e. the requested part exists and the quantity in the store is sufficient), then \( t3 \) will fire, creating an entry for the order in the pack list. Otherwise \( t4 \) will fire, creating a rest order. Both \( t3 \) and \( t4 \) take orders away from the set \( X \) one by one \( (\diamond X = X \setminus \{x\}) \), where the circle is the temporal next operator, i.e. \( X \) in the next state is equal to \( X \) in the previous state minus the element \( x \).

- When \( X \) finally becomes empty, \( t2 \) will fire. The firing of \( t2 \) removes the token from \( P3 \), and consequently, nothing more can happen in the network until a new "Handle Orders" message arrives, i.e. execution halts.

Recognizing that the states of the two output places \( P5 \) and \( P6 \) have no impact on what can happen next in the network, we can derive automatically the state transition diagram of fig. 6.2. Concentrating on one execution of the network (i.e. not going around and around the dotted arc resulting from new “Handle Orders” messages), we can establish that there are basically four different ways of going through this graph from top to bottom: either going directly through it (i.e. skipping both loops), including only the \( t3 \) loop (all orders can be satisfied), including only the \( t4 \) loop (no orders can be satisfied), or including both loops (satisfy some orders, others not).
Figure 6.1: A Behaviour Network, before and after abstraction

Figure 6.2: An STD for the network
Table 6.1: The analysis of the path without loops

<table>
<thead>
<tr>
<th></th>
<th>PC</th>
<th>SC</th>
<th>PA</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>m = &quot;H.O.&quot;</td>
<td>X = W</td>
<td>STOP</td>
</tr>
<tr>
<td>t2</td>
<td>W = ∅</td>
<td>STOP</td>
<td>STOP</td>
</tr>
</tbody>
</table>

The next step of the algorithm amounts to looking at each of these possible paths through the state-transition graph in order to eliminate internal variables. This analysis is performed by using a table with 3 columns, PC (path condition), SC (state change), and PA (path assertion). The first will contain preconditions for the transitions of the path, the second state changes on internal variables, and the third the assertion of external output. For the alternative where both loops are skipped the analysis is very simple, as illustrated in table a-tab1.

The main clue of the path analysis is illustrated in the PC entry of t2 here. The precondition of t2 has been given as X = ∅. However, before we get to t2 in the transition sequence, we have executed t1, for which we have recorded the state change X = W. Whenever new conditions or state changes are to be inserted, we have to consult the last state change of all variables involved. Thus, we find that X in the precondition of t2 can be replaced with W, so that the precondition of t2 becomes W = ∅. Thus, we have managed to eliminate the internal variable X.

The analysis of the path with only the t3 loop is a little more complex. Since the number of orders must be regarded as finite but arbitrary we first establish an expression for the first run-through of the loop, and then we extrapolate this to final run-through (the n-th). The x'es are thus indexed according to their completely arbitrary order of processing. In table 6.2 we have used the following abbreviations:

- \( U = \bigcup_{i=1}^{n-1} \{x_i\} \),
- \( V = \bigcup_{i=1}^{n} \{x_i\} \),
- \( M = \sum_{i=1}^{n-1} x_i.Q \),
- \( N = \sum_{i=1}^{n} x_i.Q \), and
- \( \ldots = y_1.C \# = x_1.C \# \wedge y_1.P \# = x_1.P \# \wedge y_1.Q = x_1.Q \) and \( y_n.C \# = x_n.C \# \wedge y_n.P \# = x_n.P \# \wedge y_n.Q = x_n.Q \), respectively.

The analysis for the other single loop path is very similar to the one just considered, only a little bit simpler (since we satisfy no orders, nothing is subtracted from the store). Thus, we do not bother to show it. The double loop path, on the other hand, is rather problematic. The state change on X can be extrapolated as easily as before, since t3 and t4 update this variable in the same way. For s.Q, on the other hand, the state change cannot be extrapolated exactly. Whenever an \( x_i \) is taken by t3, \( x_i.Q \) is subtracted from s.Q, whereas nothing is subtracted when it is
### 6.2. Constructivity in BNM and PPM

<table>
<thead>
<tr>
<th></th>
<th>PC</th>
<th>SC</th>
<th>PA</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>m = &quot;H.O.&quot;</td>
<td>X = W</td>
<td></td>
</tr>
<tr>
<td>1st t3</td>
<td>((\exists x_1)((\exists s)) (x_1 \in W \land s \in S\land x_1.P# = s.P#\lor x_1.Q \leq s.Q))</td>
<td>X = W {x_1} \land s.Q = s.Q - x_1.Q</td>
<td>((\exists y_1))(...)</td>
</tr>
<tr>
<td>nth t3</td>
<td>((\exists x_n)((\exists s)) (x_n \in W \land s \in S\land x_n.P# = s.P#\lor x_n.Q \leq s.Q - M))</td>
<td>X = W {V} \land s.Q = s.Q - N</td>
<td>((\exists y_n))(...)</td>
</tr>
<tr>
<td>t2</td>
<td>W {V} = 0</td>
<td>STOP</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: The analysis of the path with only the t3-loop

taken by t4. Since the choice between t3 and t4 is itself dependent on the value of s.Q, an exact expression for how many \(x_i\) are satisfied and how many are not cannot be established, neither can we give an exact expression for the value of s.Q when the execution of the network halts — all we can say is that it must be somewhere between the old value and 0. The analysis is shown in Table 6.3.

The final step is to establish the higher level network with pre- and postcondition. By some symbolic manipulation, we are able to obtain the following formulae:

\((\forall x)(x \in W \rightarrow ((\exists y)(...) \lor (\exists z)(...))\)

and

\((\forall s)(s \in S \rightarrow (\sum_{z \in W \land z.P\# = s.P\#} x.Q \leq s.Q \rightarrow s.Q = s.Q - \sum_{z \in W \land z.P\# = s.P\#} x.Q \land (\sum_{z \in W \land z.P\# = s.P\#} x.Q \leq s.Q \rightarrow \min(s.Q - \sum_{z \in W \land z.P\# = s.P\#} x.Q, 0)))\)

for the general case where both loops are taken. Since these expressions also cover the more special cases where one or both loops are omitted, it can be used as the postcondition for the higher level transition t shown in fig. 6.1(b). The precondition of this higher level network will be the same as the precondition of the first transition executed, i.e. m = "Handle Orders".

As can be seen, there is a certain vagueness in the expressions, due to the problems just mentioned with the double loop path. Still, the expressions obtained are quite informative about what the network does, so evidently it is possible to do some consistency checking even in cases where we cannot express the outcome of the execution of a network exactly.

The work accounted for in [100] resulted in the implementation of a Prolog prototype for BNM abstraction, sophisticated enough to handle cases such as the order handling problem above.
Table 6.3: The analysis of the path with both loops

6.2.2 PPM

PPM ([60] [9] [83] [64]) is a language for information systems modelling. It has sublanguages: PhM (Sölvbergs Phenomenon Model) for static modelling and PrM, a formalized extension of DFD, for dynamic modelling. In addition to DFD, PrM provides the following constructs:

- logical connectives over input flows,
- logical connectives over output flows,
- triggering marks on input flows,
- termination marks on output flows,
- formal execution semantics, and
- decision tables for describing process behaviour.\(^2\)

The logical connectives (usually called ports) define the logical relations between the inputs and outputs of a process, respectively. The port symbols are shown in fig. 6.3. From these one can construct more complicated logical relations on the input and output by putting ports inside each other. The semantics is as follows:

---
\(^2\)At least this was the formalism used for describing process behaviour at the time when the work on abstraction for this language was performed.
6.2. CONSTRUCTIVITY IN BNM AND PPM

Figure 6.3: The PrM port symbols

- For any incarnation of a process
  - a single flow is satisfied if it occurs,
  - an AND port is satisfied if all its immediate subparts are satisfied,
  - an XOR port is satisfied if one and only one of its immediate subparts is satisfied,
  - an OR port is satisfied if at least one of its immediate subparts is satisfied,
  - a REP port is satisfied if all its immediate subparts are satisfied a number of times, and
  - a COND port is satisfied if all its immediate subparts are satisfied, or if none of its immediate subparts are satisfied.

- Every incarnation of a process must behave such that its outermost input port and its outermost output port are satisfied.

The algorithms for PrM [101] were more complicated than those for BNM, due to the greater complexity of the former, and due to the introduction of some intricate tactics for eliminating redundancy in the first step of the analysis. However, nothing fundamentally new was introduced, and no new problems occurred. Thus, we only give a very simple example of abstraction for PrM.

Fig. 6.4(a) shows some of the activities of a guano packing factory. Due to the simplicity of this network we have not bothered to add decision tables — we merely write variable updates along the concerned flows. The network can be explained as follows:

- A guano shipment arrives at P1 (and triggers this process).
- The guano is weighed and its volume is computed (the volume is always twice the weight). A message is sent to P2, informing this process about the guano volume.
- P2 is triggered by the receipt of the volume message and computes how many barrels P1 will need (volume/100). The necessary number of barrels is taken from the store and sent to P1 (which terminates P2). We assume that there are always enough barrels in the store.
CHAPTER 6. CONSTRUCTIVITY

![PrM Diagram](image)

(a) Pack guano

(b) Barrels

Figure 6.4: A PrM diagram, before and after abstraction

![STD Diagram](image)

Figure 6.5: The STD for the simple guano packing network

- P1 receives the barrels, packs the guano into them and sends the packed guano away. This terminates P1 (and thus execution of the network as a whole).

Due to the simplicity of this network, the state transition diagram will be a strict sequence (as shown in fig. 6.5). Here we easily see without using any table that "v" will be replaced with "2·w", which furthermore gives us that the number of barrels is "w/50". The higher level network, with the abstracted variable expressions is shown in fig. 6.4(b).

To date the algorithms for PrM abstraction have not been implemented, but this will soon happen in connection with the PPM2001 project, which is building a case-tool environment based on PPM.

6.2.3 Experiences Made

The major experiences of the two projects on abstraction were the following:
• The possibilities for automating the abstraction task seems promising. The prototype for BNM was able to deal with a fairly complicated network involving two loops. For PPM nothing was implemented (due to the larger complexity of the problem), but the algorithms were specified in so much detail that an implementation is clearly no problem.

• The reason for the promising features of the languages seem to be their formality.

• Some limitations were identified, in that exact descriptions could not be obtained for systems containing a sufficient degree of non-determinism (because of interacting loops etc.). In these cases, one would not be certain about discovering inconsistencies. But even in these cases, it seemed that the algorithms were able to derive some helpful information.

• For many kinds of systems, the network structures potentially causing problems will be very rare, and thus an automatic constructivity facility might be very helpful to ensure correct decomposition.

The constructivity support of a tool will of course be even stronger if analytic abstraction is combined with empiric approaches like simulation — the latter coming into action if exact verification is impossible.

The notion of constructivity is related to the notion of axiomatic program verification [38] [52], where every program statement is characterized by a pre- and postcondition, and the pre- and postcondition of the entire program is constructed from those of its statements. Just as with the approach shown above, the problem is to find invariants for loops. The problem becomes even harder for concurrent programs, and similarly for modelling approaches with separate processes, as we have to take the arbitrary interleaving between processes into account.

6.3 Constructivity and Pan-Presumptionism

As can be seen from the Langefors quotation, constructivity requires some description of interconnections and component properties. The general hierarchical language to be defined later in this thesis does not include such things, and thus, there is very little to construct. Thus, it is beyond the scope of this thesis to go into any detailed discussion about constructivity in the following.

However, interconnections and component properties must be defined sooner or later when modelling some system, so the pan-presumptionistic attitude does not make constructivity less important. Just like anything else, routines for automatic abstraction can be defined in the meta-level language accompanying our framework (as will be explained in chapter 9).
Chapter 7

State-of-the-Art Survey

7.1 Our Focus of Comparison

Examples of surveys of abstraction mechanisms in several languages are [55], [88], [45], and [93]. The first two are surveys of languages for semantic data modelling, and looks at all of the CAGA constructs. The latter two concentrate only on association, using a very detailed framework for investigating the exact semantics of this concept in several languages for data modelling and knowledge representation. Our focus is somewhat different from the one of the above works:

- The above works only deal with static hierarchies. For our purpose it is necessary to pay attention to all kinds of hierarchies; thus, we also have to look at dynamic modelling.

- Unlike [45] [93] we want to look at all kinds of abstraction mechanisms, not only association. Moreover, a very detailed look at semantic restrictions may not be very interesting to us because we want to end up with a framework as general as possible.

- The framework deals only with the conceptual basis of the languages, not the external representation. We have a need to consider also the external representation, with a special emphasis on diagrammatic constructs.

The main motivations for this comparison exercise are

- to find out how general a pan-presumptionistic framework would have to be, by looking at several languages with rather different viewpoints,

- to get inspiration for inventing some good diagrammatic representation for our framework, and finally,

- to find out if some of the current languages show some pan-presumptionistic potential.

Thus, this is not an evaluation of language quality (which will be discussed in chapter 8), only an investigation of what is provided, and how it is presented.
7.2 Language Survey

The languages were selected according to the following criteria:

- being somehow related to the field of information systems engineering,
- being (at least partly) diagrammatic,
- being very commonly used, or
- being particularly interesting for the purpose of our discussion.

Furthermore, we tried to include several different kinds of languages, in order to get a fairly broad picture of what the field of information systems engineering has to offer. The languages chosen are:

1. static languages:
   - ER [24], the classical Entity-Relationship language, which has inspired many other static languages, and even some dynamic ones (such as ERAE).
   - PhM [107] [60] [58], Sølvberg’s Phenomenon Model, an ER extension for real world modelling.
   - NIAM [114], a binary relationship language.
   - ERT [110], the Tempora project Entity-Relationship-Time language.

2. operational languages:
   - SHM+ [18], the conceptual modelling language of ACM/PCM (Active Component Modelling/ Passive Component Modelling).
   - DFD [44], the classical data flow diagram language.
   - PrM [83], the Process Modelling language — a formalized extension of DFD.
   - Petri nets [90].
   - Statecharts [48], an extension of ordinary state transition diagrams, partly motivated by the need to facilitate abstraction.

3. declarative languages:
   - DADES [81], a non-diagrammatic declarative language
   - CIM [43], a non-diagrammatic declarative language
   - ERAE [33], a declarative language including dynamic diagrams inspired by ER.

4. other languages:
   - semantic networks [36], a large family of graph-based languages for representing knowledge in a way which is somehow close to human thinking.
   - Higraphs [49], a general hierarchical language (for instance applied in Statecharts).
7.2. LANGUAGE SURVEY

7.2.1 ER

The Entity-Relationship Language [24] allows for the graphic representation of abstract sets, relationships between these sets, and attributes defined from both entity and relationship sets to printable value types. An example illustrating the diagrammatic representation is given in fig. 7.1, the rectangles being entity sets, the diamond a relationship, and the ovals attributes. As pointed out in [55], [88] the only abstraction mechanism provided in the original ER language is aggregation (but later works have extended the language to provide generalization and/or association), and the use of aggregation is rather restricted. It is provided through the relationship construct, which has exactly the cross product definition given in chapter 4. With reference to the discussion in chapter 5, the ER language can be described as a data modelling language, rather than a language for modelling the real world.

7.2.2 PhM

The PhM language [107] is an extension of the ER language. The following additions are interesting here:

- PhM distinguishes between real world phenomena and information objects, thus being able to support the modelling of reality as well as the modelling of data. An example is given in fig. 7.2 (adapted from [58]), where the rectangle denotes the real world phenomenon student, whereas the parallelogram denotes the information about students.

- Association is provided by means of a member relation, indicated by arcs leading from the group to the member, annotated with the word “member” (or possibly an abbreviation for this).

- Generalization of phenomena is provided by means of a subset relation, shown by arcs from the general to the special, annotated with the word “subset” (or possibly an abbreviation for this). When several specializations are attached to the same class, it is possible to add on information showing whether the subsets are disjoint, or even a total partition. This can be depicted by sticking the respective arcs together with a bow annotated by “ds” (disjoint subsets) or “tp” (total partition).
Figure 7.2: Phenomena and information objects

Figure 7.3: Hierarchic constructs of PhM

The diagrammatic representation of an association is shown in fig. 7.3(a), generalization in (b), and a total partition in (c).

7.2.3 NIAM

The NIAM language [114] [103] is a binary relationship language, which means that relationships that are trinary or more are not allowed. Relationships with more than two involved parts will thus have to be objectified (i.e. modelled as entity sets instead). In other respects, the NIAM language has many similarities with ER — the distinction between entities and printable values is reflected in NIAM through the concepts of lexical and non-lexical object types (LOT and NOLOT), where the former denote printable values and the latter abstract entities. Aggregation is provided by the relationship construct just like in ER, but NIAM also provides generalization through the subobject-type construct. The diagrammatic notation is rather different from that of ER; an example is shown in fig. 7.4 (which is a reduced and slightly modified version of an example from [79]). NOLOTs are shown as circles, relationships as two adjacent rectangles (to account for both ways
of reading the relationships, called *roles* in NIAM terminology — going from one NOLOT to another, the first relationship box is the one to be chosen). LOTs can be written in parentheses within the NOLOT which it refers to if there is a unique value for each NOLOT, otherwise they have to be written separately in dotted circles. Objectification is shown by the node labelled StudentSubject, and generalization by the edge leading from student to person (the fact that the former is a subtype of the latter is indicated by letting the arc go into the node). The bidirectional arrows on the roles indicate relationship cardinality constraints. The relation between Student and Subject is many-to-many, and the one between Person and Name one-to-many (i.e. a person only has one name, but several persons can have the same name). One-to-one relationships are indicated by a separate arrow on each role.

### 7.2.4 ERT

The ERT language was developed within the Tempora project [110]. Just like NIAM, ERT is a binary relationship language. Considering hierarchies, ERT provides the following:

- Aggregation is provided through the definition of complex objects. Diagrammatically, this is done as illustrated in fig. 7.5, i.e. by putting component objects inside the assembled ones. However, it must be noted that the complex object construct is not strictly limited to aggregation — by means of

---

1 The "T" stands for time and refers to the explicit attachment of time intervals to entity and relationship classes offered by the language. This aspect, however, is not of any interest to our discussion of hierarchic constructs.
Figure 7.5: Aggregation in ERT

the cardinalities of the relations between the components and the whole, situations of choice and repetition can also be modelled. Thus, the complex object construct itself might be called a vague composition construct, but relationship cardinalities eliminate the vagueness.

- Generalization is supported, and it is possible to require subsets to be disjoint and total. Diagrammatically, generalization is done by drawing an arrow from the special class to a circle (filled means total, non-filled means non-total) which is again connected with an arrow to the general class. Subsets connected to the same circle are disjoint. Fig. 7.6(a) shows “good customer” and “bad customer” as a total partition of “customer”, (b) as a non-total partition, and (c) is the case where the subsets need not be disjoint (i.e. some customers might be both good and bad).

Association is not provided (except for the possibility of modelling repetitions between a complex object and its components).
7.2. LANGUAGE SURVEY

![Diagram](image)

Figure 7.6: Generalization in ERT

![Diagram](image)

Figure 7.7: Structural hierarchies in SHM+

### 7.2.5 SHM+

SHM+ [18] has both a static and a dynamic part. The only structural component of the language is the object. Aggregation, generalization, and association are supported by means of different kinds of arcs, as shown in fig. 7.7 (adapted from [60]).

The dynamic part of SHM+ is very similar to the static. The symbols used are shown in fig. 7.8 (adapted from [60]). As can be seen, the static hierarchic constructs have been slightly modified in that:

- aggregation is given the meaning of sequence,
- generalization is given the meaning of choice, and
- association is given the meaning of repetition.

The three abstraction mechanisms provided can be used together with some operations to define actions. An example of this is the operation FORM-CLASS, shown in fig. 7.9 (adapted from [60]). Looking at this example, it becomes clear that the dynamics are in fact attached to the static structure. It is impossible to define hierarchies of operations as such — the dynamic hierarchies are completely
dependent on the static. Thus, it can be said that SHM+ has some object-oriented tendencies.\footnote{Again it is emphasized that some people mean something very special and rather restricted when using the word object orientation. We use the notion according to the distinction in aggregation principle outlined in chapter 5. Since SHM+ structures knowledge around its static components (objects) rather than around operations or agents or something else, it can be called object-oriented in this sense.} The dynamics also reveal the data-orientation of the approach already experienced when looking at the static parts — just like ER, SHM+ seems to be intended for data modelling and not for real world modelling.

The definition of AGA for dynamics done for SHM+ is quite interesting, but the

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure7.8.png}
\caption{Dynamic hierarchies in SHM+}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure7.9.png}
\caption{Database actions}
\end{figure}
Figure 7.10: Vague composition in DFD

requirement that aggregation is a strict sequence makes it difficult to deal with cases where there is parallelism, or where the sequence is not fixed.

7.2.6 DFD

Data Flow Diagrams [44], [30] consist of processes, stores, and flows. Hierarchies are constructed by decomposing these, according to the following rules:

- Stores can only be decomposed into lower level stores.
- Flows can only be decomposed into lower level flows.
- Processes can be decomposed to contain flows and stores as well as lower level processes.

Thus, almost any kind of dynamic hierarchy can be captured. However, the hierarchic construct of composition is very vague in that it is not stated whether it is aggregation, generalization, or association. Consider for instance the example of fig. 7.10. Is the process A an aggregation of B and C, or is it a generalization of the two? Both interpretations are possible, as will be shown in the following subsection on PrM.

Thus, the DFD notion of decomposition is vague and general compared to the more specific AGA constructs provided by SHM+. Sometimes the execution of a mother process will imply (i.e. consist of) one execution of each of the daughter processes, i.e. a typical aggregation. However, in other cases, the mother process will choose among various alternative daughter processes, indicating generalization, or require the repetition of some of its daughter processes, indicating some association.

7.2.7 PrM

The PrM language [83] is a formalized extension of DFD. The only extension specifically interesting to this chapter is the introduction of logical connectives between flows on the inputs and outputs of processes. By means of simple connectives like
AND and XOR, one can indicate whether two flows should both be received or sent by a process, or whether one should make a choice between them. Also, there is a REP connective for repetition, signalling that sending or receipt on a flow will occur several times for each process instance execution. These logical connectives may increase the precision of PrM decomposition as compared to the vagueness of DFD. This is exemplified in fig. 7.11 where AND yields aggregation (i.e. the set of A executions is a subset of the cross product of B and C executions) and XOR generalization (i.e. the set of A executions have two subsets, one consisting of B executions and one consisting of C executions).

However, it must be noted that the reason why this succeeds here is partly because the example diagram is extremely simple. In general, hierarchic relations will still be vague, in spite of the connectives. This does not signal any failure on behalf of the connectives, since their intended use is to relate flows logically, not to distinguish between different hierarchical relations for process decomposition.

7.2.8 Petri nets

The classical Petri net, as presented in [90] cannot be decomposed at all, i.e. they are completely flat, with no support for hierarchies whatsoever. This is inevitable by the fact that transitions are instantaneous, which makes it impossible to compose more complex networks (whose execution is bound to take time) into higher level transitions. However, there exists several more recent dialects of the Petri net language (for instance [70] [71] [106]) where the transitions are allowed to take time, and these approaches provide decomposition in a way not very different from that of a data flow diagram. Due to the formality and simplicity of Petri nets, it will often be possible to determine whether transitions are AND’ed or XOR’ed or repeated to form the higher level transition. Fig. 7.12 shows an example of aggregation and generalization corresponding to the PrM example of the previous subsection.

Since there is only a vague decomposition construct provided, the distinction is not easily seen and requires tedious inspection for complex networks. The situation considering only the representation of hierarchies is thus very much the same as
7.2. LANGUAGE SURVEY

Figure 7.12: Aggregation and generalization in Petri nets

(a)  
(b)

Figure 7.13: Generalization and aggregation in statecharts

in PrM.

7.2.9 Statecharts

The traditional state transitions diagrams have the same weakness as Petri nets in that they cannot be decomposed. The exponential blow-up in the number of states for complex systems has the effect that diagrams become difficult to understand and use. [48] addresses these difficulties and comes up with a new formalism, the statechart, which is an extension of the traditional state transition diagram.

The two basic hierarchic constructs provided are those of generalization, corresponding to a logical XOR (cf. fig. 7.13(a) where the composite state A is a generalization of the states B and C, i.e. if in the state A you are either in B or in C), and aggregation, corresponding to a logical AND (cf. fig 7.13(b) where the state A is an aggregation of B and C, i.e. if in the state A you are both in state B and C). The aggregation of states might often correspond to the aggregation of some physical component from smaller parts (i.e. if part 1 is in the state B and part 2 is in the state C, then the overall system is said to be in the state A in the example in the figure).
By its ability to compose states into higher level states, the statechart formalism avoids the pitfall of traditional state transition diagrams when it comes to poor abstraction facilities. The basic hierarchic constructs of fig. 7.13 makes it possible to structure the states in an AND/XOR tree. For generalization it is possible to show overlapping diagrammatically, as indicated in fig. 7.14, thus turning XORs into ORs. The diagram states that C is a subset both of A and B.

7.2.10 DADES

DADES [81] consists of three components: a formal language for the specification of IS requirements, a method for checking the logical consistency of the specifications, and a method for verifying the logical correctness of some design decisions. The formal specification language is declarative, which makes it possible to specify the input and output of an IS independently of storage and processing concerns, consists of domain declarations, relation schemes, assertion times, value sets of the attributes, and derivation rules. Although presenting many epoch-making ideas, the language has little to offer when it comes to abstraction mechanisms. The conceptual schema is based on the relational model, which is inherently flat. Also, there is no diagrammatic representation.

7.2.11 CIM

CIM (Conceptual Information Model, [43]) is a declarative language based on a time model and supports the declaration of entities, events, relationship functions, and data types. Conceptually, CIM is a predicate logic extension of the ER language, and thus, aggregation is supported in the same weak way as in ER. Generalization is supported through a generalization assertion construct, but this construct is only provided for entity sets, not for events. Thus, generalization is possible only for statics. There are no explicit constructs for association or classification. Neither is there any diagrammatic representation. Just like DADES, CIM featured some very important, path-breaking ideas, but the support for hierarchies is rather weak.
7.2. LANGUAGE SURVEY

7.2.12 ERAE

ERAE (Entity Relationship Attribute Event, [32], [33], [46], [37]) is pretty similar to CIM conceptually, in that both can be presented as a first-order logic version of an extended ER language. But ERAE, being a more recent invention, also provides some significant additions to the ideas presented by CIM. The ERAE language is partly diagrammatic, depicting entities as boxes, events as ovals, relations as arcs and attributes as lines. The diagrams (which ERAE calls declarations) are complemented by rules written in first order temporal logic (which ERAE calls statements).

ERAE supports the following abstraction mechanisms:

- Generalization through a predefined “isincluded” relation.
- Association, by means of the predefined “isin” relation.
- The context construct (accounted for in [37]) which seems to deal with aggregation (although it is not stated that this construct is identical to aggregation — it might be more in the direction of vague composition). A context is a syntactic block containing a set of declarations (i.e. some chunk of a diagram) and statements (i.e. temporal logic rules).

An example depicting association and generalization is given in fig. 7.15. It is also interesting to note that ERAE supports the explicit modelling of instances (double rectangles, whereas classes have simple rectangles). This is also included in fig. 7.15.

The context construct, present only in the most recent work on ERAE, is particularly interesting from our point of view. Quoting [37] “a context may be interpreted as a process, or an agent, or an object” — thus, it seems that the idea of the context is a step in the direction of pan-presumptionism. At least, contexts imply a major improvement of the abstractive power of the ERAE language:

- Contexts can be organized in hierarchies, one context containing other smaller context. Thus, support for aggregation is achieved.
- Statements can be embedded in contexts — thus, it is no longer the fact that the statements are doomed to reside in one big, flat rule base.
- The main purpose of contexts is to limit the visibility of names — statements within a context may refer only to names visible within this context. Contexts thus provide information hiding.

---

3It should be noticed that the ERAE terminology deviates somewhat from the one used here: the standard relationship construct is called “association” in ERAE, whereas the term “classification” is sometimes used for the “isin” construct (as signalled by “This relation is similar to the “is-a” link in semantic networks (at least when used for classification)” in [32]. However, the “isin” construct does not represent any shift in meta-level. As also stated in [32]: “A predefined relation, named “isin”, applies between a group and its members.” Obviously, the “isin” construct represents what is called association in our terminology.
Figure 7.15: Generalization and association in EERA

Figure 7.16: Contexts in EERA

Contexts can also communicate, either by means of shared events (i.e. communication by messages) or shared entities (i.e. communication by shared memory). This is illustrated in fig. 7.16 (which is adapted from [37]).

The visibility rules for EERA contexts are as follows:

1. any name declared in a context is visible in that context
2. value types and functions/predicates exclusively defined on value types are visible in all contexts
3. the types declared to interface with a context, as well as their attributes, are visible in that context
4. any name visible in the context C is visible in any context containing C

In the example in fig. 7.16, D, g (rule 1), XXX, YY (rule 2), plus events of type
EV and entities of type EN (rule 3) would be visible within C2. In a context containing both C1 and C2, all names in the diagram would be visible (rule 4).

We conclude that ERAE is a very interesting approach in that it provides quite powerful abstraction mechanisms in spite of being declarative.

7.2.13 Semantic Networks

A semantic network is a graph where the nodes are objects, situations, or lower level semantic networks, and the edges are binary relations between the nodes. Semantic networks [36] [19] [60] constitute a large family of languages with very diverse expressive power. The version found in [60] provides the following support for hierarchies:

- Generalization and association through the “subset” and “element” relations, both presented as arcs annotated with “s” and “e”, respectively. If subsets are required to be disjoint, “ds” is used, and if elements are required to be distinct (i.e. different from all other elements connected to the same higher level set), then “de” is used.

- Aggregation through the space concept. As stated above a network node can be a lower level network — such nodes are called spaces. These make it possible to bundle nodes and arcs together to form larger components. The internals of a space cannot be seen from outside.

The hierarchical constructs of the semantic network language considered are depicted in fig. 7.17. The space S bundles together the four nodes U, V, X, and Y. Notice that these nodes are variables. Nodes can also contain logical connectives, as indicated by the Negation node. The combination of variables and logical connectives gives us an expressive power comparable to first-order logic. The relations “agt” and “obj” are predefined semantic relations with the meaning of agent (i.e. someone performing some action) and object (i.e. something on which an action is performed). The meaning conveyed by the negated space in this example is that no person can write and review the same paper.

Sowa’s Conceptual Graphs formalism [104] can be said to be a special kind of semantic network language. The abstraction facilities provided are very much the same, although their external representation might be different. For instance, negation is represented in a way which is rather different from the semantic network example of fig. 7.17 where negation is depicted as an ordinary node named negation. Sowa instead puts a ¬-symbol just to the left of the proposition to be negated. Fig. 7.18 (adapted from [104]) shows the conceptual graph for “Every person has a mother”, where two nested negations have been used to achieve universal quantification.

Similarities between semantic networks and ERAE are not coincidental, as stated in [32] the latter was much inspired of the former. However, it should be noticed that ERAE contexts differ from semantic network spaces in that the visibility
Figure 7.17: A semantic network

Figure 7.18: A Conceptual Graph
relations are opposite. In a semantic network, the internals of some space can only be seen within this space, whereas in ERAE, everything in a context can be seen within any larger context containing the former.

### 7.2.14 Higraphs

The Higraph language [49] emerged when Harel realized that his generalization and aggregation icons for Statecharts ([48], cf. subsection 7.2.9 above) might be used for other things as well — in fact they can be filled with any kind of substance. In the Higraph formalism, a rounded rectangle (called a blob) denotes, simply, a set (which can be a set of agents, events, processes, gas pipes, cars, or anything else). Edges between blobs provide relations (which can be directed or undirected, and potentially of many different kinds). The hierarchic facilities are exactly the same as for Statecharts. The Higraph language is in fact rather apresumptionistic — one can use the formalism for state transition diagrams (as done in [48]), for ER diagrams (as suggested in [49]), or for something else.

### 7.3 Summing up

#### 7.3.1 Concepts

The survey has revealed the following:

- Most languages are strongly class or set based — the only languages allowing for the explicit modelling of instances are ERAE, and semantic networks.

- Classification is generally weakly supported, being merely implicit in all languages except possibly some dialects of semantic networks. This is not so strange, since classification is only useful for meta-level reasoning, which goes beyond the level of ambition of most current languages.

- Whereas static formalisms usually support some or all of AGA explicitly, operational languages tend to supply only the vague decomposition. The only exception to this is SHM+, but
  - the hierarchical structure of dynamics in SHM+ reflects the static structure, and
  - SHM+ has made a very limiting assumption in providing dynamic aggregation as strict sequence.

When it comes to generalization, some languages have constructs for expressing whether subsets are disjoint and/or total.

- From several angles there has been a trend towards a better support for hierarchies:
- The traditional declarative languages provided little when it comes to hierarchical structure. The more recent approach ERAE, achieves AGA by combining temporal logic with semantic networks. The hierarchical constructs of ERAE are thus very similar to those of semantic networks, except that
  * the ERAE context construct used for aggregation has the opposite visibility restrictions of traditional semantic network spaces.
- The traditional Petri nets were not composable due to their zero-time restriction on transitions. This has been remedied by Timed Petri nets.
- The traditional state-transition diagrams were completely flat. Now, Harel’s Statecharts have provided powerful support for aggregation and generalization.
  
- None of the languages consider mass entities (except for Conceptual Graphs [104], which is a special dialect of semantic networks).
  
- Only one of the languages (PhM) entertains constructs for explicitly distinguishing between real world objects and information about these.

7.3.2 Diagrammatic Representation

It can be concluded that there are basically two ways of representing hierarchical relations diagrammatically:

- By drawing arcs from the icons of the higher level things to the lower level things. Henceforth, this will be called the edge notation. The tree notation is used in SHM+, ER, PhM, ERT, ERAE, and semantic networks.

- By putting the icons of the lower level things inside the higher level things. Henceforth, this will be called the onion notation. The onion notation is used in ERT, DFD, PrM, Timed Petri nets, ERAE, and semantic networks.

In the edge notation distinction is made between AGA by

- using different kinds of arrows (SHM+), or
- annotating the arrows differently (PhM, ERAE, semantic networks),

and totality/disjointness, if at all shown diagrammatically, by

- annotation (PhM, semantic networks), or
- special symbols (ERT: filled and non-filled circles, cf. fig7-ert2)

In the onion notation distinction between AGA is either

- not made explicit (DFD, PrM, Petri nets),
or done by means of different icons (Higraphs, using a normal blob for generalization, and one divided by a dotted line for aggregation). Here disjoint and non-disjoint specializations can be shown by non-overlapping and overlapping blobs respectively.

Other languages use the onion notation only for one abstraction mechanism (aggregation in ERT, ERAE, semantic networks) and the edge notation for the others (if providing several).

7.4 State-of-the-Art and Pan-Presumptionism

The investigation of current languages has revealed that there are languages with apresumptionistic or even pan-presumptionistic tendencies:

- Semantic networks: Nodes in ordinary semantic networks can be absolutely anything — thus, semantic networks do not force the user to view the world in terms of some predefined classification of real world concepts.

- ERAE, being a combination of semantic networks and logic, has similar tendencies. It should be noted however, that ERAE has made some assumptions by introducing the concepts entity, relationship, event, attribute, and value. However, by the introduction of the context concept, the approach has increased its abstractive power — as well as its generality. Quoting from [37], “a context may be interpreted as a process, or an agent, or an object”. According to this, it seems that ERAE is able to deal with all the three typical aggregation principles that were discussed in chapter 5, making it possible to view the world from several different angles using the same primitive concepts.

- Higraphs: The blobs in Higraphs are simply sets, what they are sets of is left open. Harel has shown how these constructs can be used as abstraction mechanisms for rather different languages: a dynamic language (state-transition diagrams, [48]) and a static language (ER, [49]).

These languages will be important sources of inspiration for the framework to be defined in the next part.
Part III

The Hicon Framework
Part Introduction

In this part we present Hicons — a general diagrammatic framework with predefined hierarchical constructs. The framework is motivated from linguistic criteria which we make explicit before the actual definition of the constructs and their diagrammatic representation. Hicon is an amalgamated abbreviation of hierarchical construct (accounting for the conceptual dimension), and hierarchy icon (accounting for the diagrammatic dimension).

Chapter 8 draws up lots of general criteria for language quality. These are guidelines for the language definition in the two following chapters. The linguistic issues, especially those concerning diagrammatic representation, are illustrated by means of examples from existing languages.

Chapter 9 outlines the conceptual basis of our language. The conceptual basis will consist of a meta-language in which the user can define the necessary concepts, plus a set of predefined hierarchical constructs, i.e. Hicons.

Chapter 10 outlines how the hierarchical constructs, together with other knowledge, can be represented diagrammatically. Two major styles of diagrammatic representation of hierarchies are identified, namely the tree style and the onion style. The pros and cons of each are discussed. We give most attention to the onion style.
Chapter 8

Language Criteria

8.1 Language Quality

8.1.1 What is Language Quality?

Giving criteria for evaluating the quality of a language cannot be done in isolation from its purpose. A language good for poetry might not be a good language for software specification and vice versa. For the language framework to be presented in the two next chapters the following applies:

- the purpose of our language is to facilitate the communication among and between analysts, customers, and development tools (all these three should be able to produce and interpret statements in the language), in the process of representing knowledge about

  - some existing or imagined part of the real world, and/or
  - customer demands for a new information system

leading to

  - a better understanding of customer problems and needs,
  - a better understanding of solution possibilities, and
  - an information system matching as closely as possible the customer needs.

This statement about the purpose of our language is inspired by [60]. Concerning the medium, we assume that statements in the language are going to be made basically on paper or on computer screens.

\footnote{The discussion about the role of a conceptual model, chapter 15, section 3.}
8.1.2 How Can Language Quality Be Measured?

Undoubtedly, the best way to evaluate any language according to the above criteria is to give it a try in realistic information systems development, preferably several times, and compare its success with that of other languages used for problems of a similar nature. This is what we could call a result-oriented evaluation. This is impossible for the language that we are going to create in this report. A result-oriented evaluation would require the construction of a tool set to support the language, as well as finding system developer teams willing to use it. Moreover, as stated in chapter 3, the language to be presented will not be complete — it has to be coupled to something else to be useful for IS specification. Obviously, it might take a decade or more before large scale experimentation with the language is possible.

In the lack of results, we have to go for criteria which are language-oriented rather than result-oriented. Such criteria will to some extent be a matter of belief and taste, but also a matter of general linguistic and psychological knowledge [66] [4] [12] [124] [104]. It is important that we make our underlying criteria explicit, so that the motivations behind the language to be defined are made clear.

8.1.3 A Framework for Language Quality Criteria

To make the discussion more systematic, we identify several categories of criteria. First of all, one can distinguish two main kinds:

- criteria for the conceptual basis of the language, and
- criteria for the external representation of the language.

For each of these two parts, we identify the following four main groups of criteria:

perceptibility: how easy is it for human beings to grasp the language

expressive power: what is it possible to express in the language

expressive economy: how effectively can things be expressed in the language

method/tool potential: how easily does the language lend itself to proper method and tool support

Although we operate with the same subcategories for each, it is believed that distinguishing between the conceptual basis of a language and its external representation will result in a clearer discussion. The framework above is not necessarily orthogonal or complete; its purpose is to provide a nice grouping of the more detailed criteria to be presented.

In respective sections, we first deal with criteria for the conceptual basis, then for the external representation, and finally we illustrate some of the criteria by looking at some features of some of the languages evaluated in the previous chapter.
8.2 Conceptual Criteria

8.2.1 Perceptibility

The conceptual basis of the language should correspond as much as possible to the way human beings view reality. A language with a high perceptibility will typically be easier to learn and easier to remember than one with a lower perceptibility, which makes it easier for human beings to produce and understand statements in the language. The most important criteria for achieving a high perceptibility are believed to be the following:

- the concepts should be natural (i.e. supporting the way human beings prefer to reason about reality, or information systems).

- the concepts should be easily distinguished.

- the number of concepts should be reasonable. If the number of concepts has to be uncomfortably large (due to some other requirements), the concepts should be organized hierarchically, making it possible to approach the conceptual framework at different levels of sophistication. This hierarchical organization should in itself be rather natural, cf. the first item.

- the use of concepts should be uniform or consistent throughout the whole set of statements possible to express within the language. Using the same construct for opposite linguistic functions or different constructs for the same function depending on the context will tend to make the language confusing. For instance, English would undoubtedly be easier to learn if all substantives could have their plural form constructed simply by adding an "s" to the singular form.

8.2.2 Expressive Power

Ideally, the conceptual basis must be powerful enough to express anything we need to express. There is an infinite number of statements that we might need to make, and these have to be dealt with through a rather small number of concepts (due to the perceptibility requirement discussed in the previous subsection). This means that

- the concepts must be general rather than specialized (for instance one would rather provide a general action concept than one concept for swimming, one for running, one for type-writing, etc.), and

- the concepts must be composable, which means that we can make more complicated statements by putting them together. When only expressive power is concerned, it is an advantage if all thinkable combinations of statements are allowed, each yielding a separate meaning (but other criteria might pull the opposite way).
• the language must be *flexible in precision*:
  
  – To express precise knowledge we need precise constructs. This means that the language must be *formal*\(^2\) and *unambiguous*\(^3\).
  
  – At the same time, we need vague constructs for modelling vague knowledge. To fulfil both requirements, the vagueness must also be formalized (i.e. even the vague constructs must have a definite interpretation — the constructs are called vague because their interpretation is *wide* compared to the more definite constructs).

• the language must be flexible in the *level of detail*\(^4\)

  – statements must be *easily extendable* with other statement providing more details about the first.
  
  – At the same time, details must be *easily hidden*.

This means, in effect, that the language must have a substantial *abstractive power*, i.e. provide the necessary abstraction mechanisms among its concepts.

### 8.2.3 Expressive Economy

Whereas expressive power concerns what we are able to express, expressive economy concerns how briefly things can be expressed, i.e. how many conceptual constructs you need to assemble to make the statements you want to make. Introducing one concept for each possible statement would make every statement brief, but the number of concepts would be infinite. Since it is necessary to keep the number of concepts at a reasonable level (as stated in the subsection on perceptibility), a good expressive economy cannot be based on defining new concepts for everything. Instead

• the most *frequent* kinds of statements should be as brief as possible (whereas the less frequent could be more complex without significantly increasing the average modelling effort).

• The most *important* kinds of statements should be as brief as possible (whereas less important statements could be more complex without a similar decrease in the system reliability). For instance, an information system might have some statements that are very rare, but extremely important (e.g. urgent) if invoked. If these are complex, the time spent on processing them might be intolerable; moreover, being complicated they will more easily be wrongly specified.

\(^2\) which means having formal syntax and semantics...

\(^3\) which means that every statement has one unique syntactic and semantic interpretation...

\(^4\) which is related to the point about precision, but not necessarily the same: it is possible to tell a story including a lot of details using unprecise concepts, and it is also possible to recapitulate only the main events using very strict, formally defined concepts...
8.3. **EXTERNAL REPRESENTATION CRITERIA**

### 8.2.4 Method/Tool Potential

With the method/tool potential of a language we mean how well the conceptual language fits together with natural methods and tools for information systems development. Important factors here are:

- do the conceptual constructs facilitate *work division* (since information systems are complex, we cannot expect them to be developed by one single person, and maybe not even in one single place).

- does the language lend itself to *automatic reasoning*? This requires formality, but formality is not necessarily enough, since the reasoning must also be fairly efficient to be of practical use.

### 8.3 External Representation Criteria

#### 8.3.1 Perceptibility

The external representation should be easily perceived by human beings. In more detail this means:

- the external representation of different kinds of concepts should be *intuitive* in the sense that the symbol chosen for a particular concept somehow reflects that concept better than the symbol of another concept would have done.

- *symbol discrimination* should be easy. Since we have restricted ourselves to visual representation of the language statements, this means that it should be easy to see the difference between the various symbols of the language.

- the external language should be as *consistent* as possible, meaning that symbol use should be uniform, i.e. a symbol should not stand for one concept in one context and a completely different concept in another context.

- one should strive for *symbolic simplicity* — both concerning the primitive symbols of the language and the way they are supposed to be connected to form larger statements. If the symbols themselves are visually complex, specifications containing a lot of symbols will be even more complex, and thus difficult to read.

- the use of *emphasis* in the external language should be in accordance with the relative importance of the statements. Factors that have an important impact on visual emphasis are the following:
  
  - size (the big is more easily noticed than the small)
  - solidity (e.g. **bold** letters vs. ordinary letters, full lines vs. dotted lines, thick lines vs. thin lines, filled boxes vs. non-filled boxes)
  - difference from the ordinary pattern (e.g. *slanted* letters, a rare symbol will attract attention in a forest of ordinary ones)
Figure 8.1: Connectivity and emphasis

- foreground/background (if the background is white, things will be easier noticed the darker they are)
- colour (red attracts the eye more than other colours)
- change (blinking or moving symbols attract attention)
- pictures vs. text (pictures usually having a much higher perceptibility, information conveyed as such will be emphasized at the cost of information conveyed textually)
- position (looking at a diagram, people tend to start at its middle)
- connectivity (objects able to connect to many others will attract attention compared to objects making few connections, cf. fig. 8.1, where the circle shape at the lower left achieves visual prominence although there are other objects that are bigger, darker, or more in the middle of the diagram).

Emphasis is a very powerful mechanism for facilitating the understanding of some specification, but it can easily be overdone. Emphasizing too much, one will end up practically emphasizing nothing, and the models will only look confusing.

Undoubtedly, diagrammatic styles of representation have some important advantages over text and tables when it comes to perceptibility. [109] points out that diagrammatic visualization of knowledge makes the acquisition process more efficient, helping knowledge elicitation, completeness testing etc.
8.3.2 Expressive Power

Expressive power belongs to the conceptual basis, and the only requirement to the external representation in this respect is that it does not destroy it. Thus,

- every possible statement in the external language should have a unique representation in the conceptual basis (otherwise, the precision of the language will be destroyed), and

- every possible statement in the conceptual basis should have at least one external representation (otherwise, the generality might be destroyed).

- just like the concepts, the symbols of the language must be composable.

As indicated, only the transformation from externals to concepts needs to be unique — it is all right to have several alternative external representations for the same conceptual statement (for instance text or diagrams, trees or tables, etc.).

8.3.3 Expressive Economy

At the external level, expressive economy is concerned with how many symbols we need to use to express what we want to express. As the requirements of the previous subsection suggest, it will usually be the case that the things easily expressed conceptually will also be easily expressed externally, and the things which are complicated in the conceptual basis will also have to be complicated externally.

However, this should not fool anyone into believing that the expressive economy of the conceptual basis and the external representation of a language should be the same. In fact, a good external representation should always have an expressive economy much better than that of the conceptual basis. This is because the external representation has lots of possibilities that the conceptual basis does not have:

- omission of symbols that are understood in the context.\(^5\)

- special symbols can be defined for conceptual constructs which are frequent (or important).\(^6\)

- multiple mentionings of the same concept is unavoidable at the conceptual level. At the external level, such multiple mentionings can often be avoided.\(^7\)

---

\(^5\) An example of this from natural English: the external statement “John Rodriguez Jackson woke up, ate his breakfast, and went to play croquet with Jane, Raoul, Don, Liza, Sally, and Rod” corresponds on the conceptual level to what the linguists call the deep structure of the sentence, where John Rodriguez Jackson would have to be repeated as the subject of all the three actions in the sentence, and where the conjunction “and” would be present between each of the persons in the list at the end of the sentence. Thus, the external representation has omitted JRJ twice and “and” four times.

\(^6\) An example will be given in the next section.

\(^7\) An example will be given in the next section.
 CHAPTER 8. LANGUAGE CRITERIA

Of course, there are some pitfalls to be avoided. Making the right mistakes, one could decrease the expressive economy of an external language compared to the conceptual basis. Things to be avoided are:

- blank symbols, i.e. symbols that do not contain any information.
- external redundancy, i.e. showing the same concept in several different ways in the same external representation.\(^8\)

As will be indicated in the next section

- Diagrams have a significantly larger potential for expressive economy than tables or text.

Thus, an important aspect of expressive economy is the use of diagrams. It is impossible to convey everything diagrammatically\(^9\). Thus, the best thing to do for expressive economy is to try to express the frequent concepts diagrammatically and the less frequent textually.

### 8.3.4 Method/Tool Potential

The method potential of an external language depends very much on its perceptibility. The most important question for an external language is what tool support can be given for it:

- does the external language easily lend itself to man-machine communication?
- does it avoid features that are exhausting to look at (for instance excessive blinking)?
- does the external constructs allow for nice ways of information filtering, i.e. making various selections concerning what information to show and what to hide, and browsing, i.e. moving from concept to concept in a specification?
- does models in the external language fit into standard paper and screen formats (without making components too small to be readable)?

### 8.4 Examples

In this section we will illustrate some of the language quality guidelines mentioned in this chapter by means of examples, concentrating on external representation.

---

\(^8\)At the conceptual level, some degree of redundancy could often be helpful, for instance for consistency checking. A redundancy which is purely external, however, does not contribute to this, since specifications are always checked at the conceptual level, not at the external.

\(^9\)As for instance pointed out by Jacques Hagelstein (personal conversation, October 1988): "If you try to represent too much in your diagrams, they only become messy, and you lose the basic perceptive advantage of diagrammatic representation."
First we will give some examples on perceptibility, second some on expressive power, third some on expressive economy, and finally some examples illustrating cases where one has to choose between satisfying different criteria. Obviously, satisfying all the above criteria is very difficult, since different criteria often pull in different directions. Thus, it should be emphasized that although some of the examples point out criteria violation on the part of ERT, this does not mean that we think ERT is a particularly bad language. Surely, similar problems could have been pointed out in other languages, but we chose ERT because this is a language which we are particularly familiar with. Moreover, the criticism only applies to the external representation of ERT, not to its conceptual basis.

### 8.4.1 Perceptibility

**Union and Intersection: Intuitive Symbols**

Union and intersection are abstract mathematical concepts that cannot be directly visualized. However, it seems to be generally agreed that Venn diagrams (cf. fig. 8.2) represent these notions in a very intuitive diagrammatic way. But unfortunately, this kind of representation does not lend itself very well to representing complicated algebraic expressions. However, the symbols $\cup$ (union) and $\cap$ (intersection) are also quite appealing when it comes intuitive quality:

- The $\cup$-shape reflects several kinds of items for gathering things (like pots, vases, buckets etc.), and this corresponds well to the gathering nature of the union operation.

- The $\cap$-shape similarly reflects items for keeping things out (like umbrellas, roofs etc.) which corresponds well to the excluding nature of the intersection operation.

- Finally, since the two operations are somehow duals, it is also intuitively appealing that one is the other upside down in the external representation.
Value classes in PhM and ERT: Symbolic Discrimination

The languages PhM and ERT have been discussed in chapter 7. Their symbols for entity classes and value classes are shown in fig. 8.3. Undoubtedly, the visual discrimination between the rectangle and circle provided by PhM is very easy. However, for ERT both are depicted as rectangles, the only symbolic difference being a small black triangle on the value class. Here, visual discrimination will require a closer look, especially if diagrams are large (which necessarily means that the rectangles will be small, and the black triangles even smaller, due to limitations in screen and paper size). Thus, it must be said that when it comes to the visual discrimination between value classes and entity classes, PhM is the better of the two.

ERT: Misguiding Emphasis

Take a look at the ERT diagram of fig. 7.5 and tell me what attracts your eye before you start looking at anything in particular. If the emphasis factors stated earlier in this chapter were correct, one of the things that would stand forward in a diagram on a white background is blackness. But where is blackness applied? Does it seem that blackness is used in a conscious way to emphasize the most important constructs in the diagram at the cost of others? The answer is clearly no — looking at the ERT diagram we find that

- blackness is applied at the relationships as black squares in the middle of the link (or at the end for unary relationships).
- blackness it applied at value classes as small black triangles.

The most central construct of ERT, however, is the entity class (since the central concepts of the business domain modelled are supposed to be captured as entity classes). Thus, blackness emphasizes value classes and relationship classes at the cost of entity classes — leaving us with the conclusion that the use of emphasis in ERT is directly misguiding, cluttering the diagram with dysfunctional black spots. Probably, this will not be a serious problem to the experienced ERT modeller (who will have learnt to look through the emphasis induced by the black spots), but to novice users it is known to be dissatisfying (as for instance experienced in the work accounted for in [68]).

One way to reduce or remove the problem is to make the black spots as small as possible, or to use white squares and triangles instead, cf. fig. 8.4. (a) is the
current style of representation. In (b) the black spots have been made smaller; this diagram is less noisy (i.e. there is less confusing emphasis), but the visual discrimination between entity and value classes is more difficult. The same applies to (c), where squares and triangles have been made white. In (d) the symbol for value classes have been changed to remove the need for the triangles, and the relationship squares have been removed. All in all, there can be little doubt that the latter diagram has the highest perceptibility.

Intuitivity and Emphasis: Location and Crossing Lines

The three graphs of fig. 8.5 depict the same situations. Most people would find (a) much easier to grasp than (b) and (c). There are two reasons for this:

- in (a) the nodes which are connected to each other are always close to each other in the diagram, and this is intuitively pleasing.

- in (b) and (c) there are crossing lines which, and this attracts visual attention to points in the diagram where there is actually nothing of interest. Crosses tend to create a false impression of nodes — this is particularly evident in (c), where as much as four lines cross at the same point.

Of course, a sensible location of nodes so as to avoid crossing lines and make the picture intuitively pleasing is very much up to the person making specific statements in a language, rather than those defining the language. However, it should be noticed that the ease of avoiding crossed lines and locating nodes sensibly depends on the diagrammatic representation chosen for a language, as will be further discussed in chapter 10.
8.4.2 Expressive Power

Flows: Flexibility in Precision

Imagine three different languages including flow constructs. The languages be named A, B, and C and assume that they entertain the following concepts for flows:

A: flow

B: information flow, material flow

C: flow, information flow, material flow

Here, C has the best flexibility in precision. A only provides the concept of a flow, making it impossible to be precise if we in fact know whether the flow to be modelled contains information or material. B on the other hand, makes it impossible to be unprecise. If we just want to model a flow and are not sure whether it contains information or material, we run into problems.

Composability: Achieving Generality with Fewer Concepts

Assume that we feel a need to model processes, flows, stores, objects. All of these four can relate to either information or material, and for each there are instances and classes. Not utilizing composability, we must define the concepts information process instance, information process class, material process instance, material process class, information flow instance, ... plus the four unprecise terms process, flow, store, and object for a sufficient flexibility, a total of 20 concepts. Defining instead one concept for each of the four, as well as the concepts of information, material, class, and instance, we get away with only 8. Allowing these concepts to be composed, we can get the expressive power we want.
8.4.3 Expressive Economy

Diagrams, Text, and Tables

The diagram of fig. 8.6, describes part of the life situation of John Rodriguez Jackson and some of the people closely related to him. As can be seen, JRJ (and each of the other persons) need to be depicted only once in this diagrammatic representation. In a textual language like predicate logic, the same knowledge would be represented in the 8 predicates

- \( \text{loves}(\text{JRJ}, \text{Sally}) \)
- \( \text{is-married-to}(\text{JRJ}, \text{Liza}) \)
- \( \text{is-married-to}(\text{Liza}, \text{JRJ}) \)
- \( \text{is-the-father-of}(\text{JRJ}, \text{Jane}) \)
- \( \text{loves}(\text{Jane}, \text{Rod}) \)
- \( \text{loves}(\text{Jane}, \text{JRJ}) \)
- \( \text{is-the-boss-of}(\text{JRJ}, \text{Don}) \)
- \( \text{is-the-boss-of}(\text{Rod}, \text{JRJ}) \)

Here JRJ (and every other person for that sake) has to be mentioned in every conceptual construct in which he participates. Consequently, the expressive economy is poor compared to that of the diagram.

Table 8.1 shows a two-dimensional matrix representation of the situation. The advantage of single mentioning of names is retained, but the table representation has a problem when expressive economy is concerned in that it is bound to have an entry for every thinkable connection, whether existing or not. The blank entries of the table are examples of empty symbols, i.e. symbols that only use space in a representation without having any information content. In our example, 17 out of 25 entries are blank, resulting in a pretty bad expressive economy.
<table>
<thead>
<tr>
<th></th>
<th>JRJ</th>
<th>Sally</th>
<th>Liza</th>
<th>Jane</th>
<th>Rod</th>
<th>Don</th>
</tr>
</thead>
<tbody>
<tr>
<td>JRJ</td>
<td>loves</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sally</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Liza</td>
<td>married</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jane</td>
<td>loves</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rod</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Don</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8.1: Table for the JRJ example

Moreover, the tabular style of representation has some other general drawbacks which should not be forgotten:

- The tabular representation works well as long as there are only binary relations, and no relationships between relationships. Introducing n-ary relations in a diagram is fairly straightforward, and so is relationships between relationship. In a tabular representation, this would require the superimposition of several tables, which would make this kind of information much harder to grasp.

- The user-friendliness of tables is generally poorer than that of diagrams. A good example of this is the empiric comparison between decision trees, decision tables, and structured English in [115], where the decision trees (which are the only diagrammatic representation among the three) proved to be by far the best.

Relationships in ERT: Empty Symbols

Consider fig. 7.5 of chapter 7. What is the informative function of the black squares on the relationships? Undoubtedly, a relationship could also have been conveyed as a simple line between the two classes connected (as done, for instance, in ERAE). The only function the square might have, is that the way to read a relationship is determined by what half of the link the annotation is attached to — thus, it is “CUSTOMER places ORDER”, not “ORDER places CUSTOMER”. However, in general, it should be fairly easy to see what side the name is attached to even without the black square. Moreover, in many cases it is not very interesting to name a relationship — for instance all the relationships between entity classes and value classes in the particular diagram have been annotated with “has”. Thus, it must be concluded that the black square basically an informatively empty symbol in the diagram, doing nothing but decreasing diagram simplicity, and thus reducing the perceptibility of the language.
8.4. **EXAMPLES**

Aggregation in ERT: Diagram Redundancy

Consider again fig. 7.5 of chapter 7, in particular the representation of aggregation in the complex entity classes ORDER and ADDRESS. Here, aggregation is conveyed in two ways:

- the parts are put inside the aggregate
- there is a relation drawn between each part and the aggregate

The second is obviously redundant, since a component is necessarily related to its aggregate. The only defense for it might be the need to represent cardinalities on the relationship. But at least, in a model where cardinalities are not included, the relationships between an aggregate and its parts are completely redundant.

8.4.4 Criteria Dependency

As we have already admitted, the language quality guidelines stated in this chapter are far from independent of each other. Here we bring some examples illustrating how one nice feature often will have to be sacrificed in favour of another one which is found more important.

Intuitivity vs. Number of Symbols: Chinese and English

Chinese writing is iconographic, each word being represented by an icon which somehow reflects that word — for instance the symbol for “horse” bears visual resemblance to real world horses. The English word, as depicted in writing, has no such resemblance. Thus, Chinese writing is far more intuitive than English. However, intuitivity comes at a cost: the number of symbols — to write Chinese fairly well, one must know 3,000 symbols, and a real scholar would know about 40,000. English, on the other hand, gets away with using few symbols (i.e. letters). Composing these into words, English obtains a vocabulary large enough to be known as the richest of all human languages.

Intuitivity vs. Economy and Simplicity: Classes and Instances in ERAE

The ERAE language has been discussed in chapter 7. In the earliest versions of this language, the expression of entity classes and entity instances was done as follows:

- instances were depicted as single boxes, and
- classes were depicted as double boxes.

---

10 Which could of course have been done in some other way, but symbolic consistency might be an argument in favour of the current way of doing it.
This is an intuitive choice. A single box naturally signals a single element, and a double box similarly signals some kind of multiplicity. Still, in later versions this choice was turned upside down. Classes turned out to be represented much more frequently than instances, and consequently it would be bad for expressive economy and diagram simplicity if a class were to require the most complex representation. Thus, intuitivity was sacrificed on this particular issue, and the gain was a substantial increase in diagram simplicity.

Expressive Power vs. Simplicity: PrM and DFD

The conflict between expressive power and perceptibility is one of the basic conflicts in language design, so here it is very easy to find examples. The languages DFD and PrM were discussed in chapter 7. PrM being an extension of DFD providing possibilities for more detail and precision (which can, however, be omitted if one wants to) is trivially more expressive than DFD. On the other hand, PrM diagrams including ports are undoubtedly also more complicated than DFDs, and PrM will undoubtedly be a more difficult language to learn due to its larger number of concepts.

Expressive Economy vs. Number of Symbols: Arcs in DFD

Modelling something by means of data flow diagrams, it often happens that there is a flow from component A to component B, and at the same time a flow from component B to component A. In many cases, the two flows might even have the same content. In these cases, it is diagrammatically possible, even after naming, to depict the two flows as one bidirectional arc instead of two unidirectional arcs, cf. fig. 8.7. Note that this is possible without introducing a new concept “bidirectional flow” — it is straightforward to simply let the two unidirectional flows at the conceptual level be represented as one bidirectional flow symbol.

This trick with the bidirectional arc increases the number of defined symbols by one, but it enhances expressive economy and diagram simplicity in that one arc can be removed every time a bidirectional situation occurs. Since the bidirectional arc

---

11for instance “passenger list” both being taken from the store A and put back to the store A by the process B...
is very intuitive, and since it is of major importance for the perceptibility of DFDs that the number of arcs in the diagrams is kept reasonably small, the introduction of a bidirectional arrow seems pretty sensible.

**Expressive Economy vs. Emphasis: Value Classes in ERT and PhM**

The languages ERT and PhM have been discussed earlier in this chapter. Disregarding now the difference in symbols for value classes in the two languages, we also observe another difference in the diagrammatic representation\(^{12}\) of such classes (cf. fig. 8.8):

- ERT provides *shared* value classes, i.e. a value class need not be connected to one particular entity class. Thus, if we have entity classes CAR, BOAT, BIKE, TABLE, and CHAIR, and all these are supposed to have a WEIGHT, we can get away with drawing a single value class WEIGHT to which all the relevant entity classes can be connected.

- In PhM a value class belongs to one specific entity class\(^{13}\), this means that we have to draw five different WEIGHTs in the above case.

Whereas ERT gains expressive economy, it loses the emphasis that PhM applies to entity classes at the cost of values (by letting entity classes have a much larger connectivity). In ERT a value class can make as many connections as an entity class, which might give the intuitive impression that the two constructs are equally important. This is not true. Still, the ERT style of representation could be preferable in situations where shared value classes often occur, since the reduction of the number of nodes might result in simpler diagrams (unless the gain is eliminated by an increasing number of crossing lines).

### 8.5 Language Criteria and Pan-Preumptionism

This chapter has revealed that the criteria from which we seek guidance when about to create our language are very complex, and different criteria often pull in different directions. Thus, it is important to have some ideas concerning what priority we should give the different criteria.

With our pan-preumptionistic outlook it seems obvious that

- the expressive power must be sufficient

since reducing the expressive power would effectively mean to make a presumption (that some kinds of knowledge do not need to be represented). Consequently, expressive power gets the highest priority.

Following this decision, we have three problems:

\(^{12}\)The diagrammatic difference in this case also reflects a conceptual difference.

\(^{13}\)or relationship class
Figure 8.8: Connectivity of value classes in ERT (left) and PhM (right)

- increasing the expressive power often slows down the reasoning, i.e. reduces the method/tool potential of a language, and
- increasing the expressive power often reduces the perceptibility,

whereas the expressive economy is not necessarily made worse by an increasing expressive power (for instance, declarative languages are more expressive than operational languages, and for some kinds of knowledge they also have a far better expressive economy, being able to state by one sentence something which requires many sentences in an operational language). Thus, the main trade-off in emphasizing the expressive power is reduced method/tool potential and reduced perceptibility. However, slow reasoning could be speeded up by acquiring a faster computer, whereas nothing can be done about insufficient expressiveness. Our approach is meant to be one for the future, not one for today, which means that hardware will be even faster (and cheaper). Moreover, it is believed that a good user-interface in the development tool could remove the worst perceptibility problems resulting from a large expressive power.

Finally, it must be stressed that the need for a large expressive power reflects a problem which we cannot escape from anyway — the complexity of the real world, or as stated in chapter 2: the wickedness of IS problems. A language without a sufficient expressive power would not be able to address these problems.
Chapter 9

Conceptual Basis

9.1 A Meta-Level Language

As stated in chapter 3, we are not going to define a complete language for IS modelling, only a framework consisting basically of hierarchical constructs. This framework can either

- be connected to an already existing language to enhance its abstractive power, or

- be part of the basis of a CASE-shell environment, where the hierarchical constructs and possibly some other essential constructs are defined as default, whereas the user has to define the constructs found necessary in addition to the predefined ones.

In the latter case, we need a meta-level language to facilitate the definition of concepts, syntax and semantics. Designing such a language is a significant effort, so it would be nice if we could use something already existing. First-order logic (FOL) is an obvious first suggestion, to be able to distinguish between meta-levels it must be extended with a naming convention (quoting). [118] [89] describes the use of FOL as a meta-level architecture. However, [56] and [11] points out some weaknesses with FOL for representing meta-level architectures, each providing an alternative (Jiang's self-referential data model [56], Bergheim and Sandersen's $\theta$ [11]). Since we are not going to design any meta-level reasoning mechanism or do any kind of coding in this work, the exact choice of language for this purpose is not crucial — it would be better to wait and see what is on the market when an actual implementation of the Hicon framework is considered. For the purpose of illustration (in the appendices) we will use $\theta$. A description of $\theta$ is found in appendix A.
9.2 Constructs to be Predefined

9.2.1 Nodes, Names, and Labels

To be at all able to say something sensible about some hierarchical structure, we need some generic construct which can be composed and decomposed, and we must be able to give names to the components. We will call our generic construct a node (like in semantic networks).\(^1\)

Definition 9.1 (The Node Relation: ) \(\text{node}(\mathcal{R})\) is a unary relation expressing that \(\mathcal{R}\) is a node.\(^2\) A node can be any phenomenon: static or dynamic, concrete or abstract, instantaneous or with duration, space or time.

Definition 9.2 (The Naming Relation: ) \(\text{name}(\mathcal{R}, \mathcal{N})\) is a binary relation expressing that \(\mathcal{R}\) has the name \(\mathcal{N}\). The relation is generally allowed to be \(N:N\), i.e. several nodes can have the same name, and one node can have several names.

An unnamed node will simply be taken to mean something — formally speaking this will have the meaning of an uninstantiated variable. Since names are not necessarily unique, we need some separate label to identify nodes unambiguously.

Definition 9.3 (The Labelling Relation: ) \(\text{label}(\mathcal{R}, \mathcal{L})\) is a binary relation expressing that \(\mathcal{R}\) has the label \(\mathcal{L}\). The relation is a bijection (i.e. labels are unique).

Whereas names are supposed to be given by the user, the labels are unique identifiers that are automatically generated.

9.2.2 Classes, Instances, and Masses

In chapter 4 it was noted that mass concepts represent a problem to the set based definition of AGA. We can either go for a more general definition (e.g. type based) or we can use the set-based and try to adapt mass concepts to this. The first alternative might be the most elegant, but the problem is that a more general definition (for instance type based) will be not be very useful from an automation point of view. Moreover, it is likely that we usually will be modelling classes, and then the set-based definitions are essential. Consequently, we will go for the second alternative:

- The set-theoretic definitions discussed in chapter 4 will be used for AGA,
- instances will be adapted to these definitions (treated as sets with only one member), and

---

\(^1\)Words like object, entity, thing have the problem that their connotations are too static. Thus, node felt like the best choice.

\(^2\)In \(\theta \mathcal{R}\) will also be a relation.
• mass concepts will be adapted to these definitions in that masses are considered as *virtual sets of sets*. This means that a mass which can be measured in one or more dimensions can be split up in any partitioning of continuous intervals in these dimensions. Since this means that there will be an infinite number of possible sets for the mass, we cannot store the mass at such, but for any partitioning we can check if it is within the given limits (thus, "5 liters of water" is part of "9 liters of water" although we have not stored explicitly any fact saying that 9 liters of water is an aggregation of 4 liters of water and 5 liters of water). Other mass concepts cannot be measured in any dimension (for instance "love", "happiness") — although they are not the most likely candidates for being modelled, we can treat them in a simple way by taking the view that any such concept is a set of generic elements having the same name as itself. Thus, the statement "Jealousy is an aggregation of love and possessiveness" will result in a tuple (*love*, *possessiveness*) which corresponds to a generic element "jealousy". It can be questioned how sound this treatment is philosophically, but for us it is important only to do something so that the set-based definitions do not crash if a mass concept has to be modelled.

Definition 9.4 (The Class Relation: ) \( \text{class}(N) \) is a unary relation expressing that the node \( N \) denotes a class (whose name is the name of that node).

Definition 9.5 (The Instance Relation: ) \( \text{instance}(N) \) is a unary relation expressing that the node \( N \) denotes an instance (whose name is the name of that node).

Definition 9.6 (The Mass Relation: ) \( \text{mass}(N) \) is a unary relation expressing that the node \( N \) denotes a mass (whose name is the name of that node).

### 9.2.3 Hierarchical Constructs

In the preliminary chapters, five standard hierarchical constructs have been identified, namely CAGA plus vague composition. Classification will be taken care of by the *instance-of* relation (which is already predefined in \( \theta \)). AGA can be defined in the normal set-based way:

Definition 9.7 (The Aggregation Relation: ) \( \text{agg}(N, N_1, \ldots, N_k), \ k \geq 2, \) is a \( k+1 \)-ary relation expressing that

\[ N \subseteq N_1 \times \ldots \times N_k, \]

i.e. that \( N \) has exactly the subparts \( N_1, \ldots, N_k. \)

The nodes must either all be classes, all instances, or all masses.
Definition 9.8 (The Generalization Relation: ) \( \text{gen}(\mathcal{N}, \mathcal{N}_1, \ldots, \mathcal{N}_k), \ k \geq 2, \) is a \( k+1 \)-ary relation expressing that

\[ \mathcal{N} = \mathcal{N}_1 \cup \ldots \cup \mathcal{N}_k, \]

i.e. that \( \mathcal{N} \) can be specialized exactly into the disjoint subsets \( \mathcal{N}_1, \ldots, \mathcal{N}_k. \)

The nodes must either all be classes, all be instances, or all be masses.

As mentioned in chapter 4, there are two common types of association in semantic data modelling:

- providing the mother node with several daughter nodes, stating that the former is a set of the latter
- providing the mother node with only one daughter node, stating that the former is a subset of the powerset of the latter.

We will provide both:

Definition 9.9 (The Association Relation: ) \( \text{ass}(\mathcal{N}, \mathcal{N}_1, \ldots, \mathcal{N}_k), \ k \geq 2, \) is a \( k+1 \)-ary relation between nodes, meaning that

\[ \mathcal{N} = \{\mathcal{N}_1, \ldots, \mathcal{N}_k\}, \]

i.e. that \( \mathcal{N} \) is the set of \( \mathcal{N}_1, \ldots, \mathcal{N}_k. \)

\( \text{ass}(\mathcal{N}, \mathcal{M}), \) is a binary relation meaning that

\[ \mathcal{N} \subseteq \mathcal{M}, \]

i.e. that \( \mathcal{N} \) is a subset of the power set of \( \mathcal{M}. \)

The node \( \mathcal{N} \) must always be a class, whereas the other nodes can be anything.

Finally, vague composition can be any possible combination of AGA. This definition allows for very much, but that is just the point of the vague composition construct.

Definition 9.10 (The Vague Composition Relation: ) \( \text{vag}(\mathcal{N}, \mathcal{N}_1, \ldots, \mathcal{N}_k), \ k \geq 2, \) is a \( k+1 \)-ary relation between nodes meaning that \( \mathcal{N} \) can be obtained from \( \mathcal{N}_1, \ldots, \mathcal{N}_k \) by a finite number of applications of AGA.

9.2.4 Non-Totality

A non-total composition occurs when we know some subnodes but not all. However, the definitions given require that all subnodes are accounted for (i.e. we cannot put up \( \text{agg}(\text{"human body"}, \text{"arms"}, \text{"legs"}), \) \( \text{gen}(\text{"trees"}, \text{"spruce"}, \text{"pine"}), \) or
ass(“Belgian towns”, “Namur”, “Brugge”). This does not mean that we have any problem in dealing with non-total partitions. Remembering that an unnamed node has the function of an uninstantiated variable (i.e. something), we can simply add such a node if we know that we have not specified all subnodes (i.e. ass(“Belgian towns”, “Namur”, “Brugge”, “”)) would be all right because the empty node can instantiate not only to one node but also to a set of nodes, like the remaining Belgian towns).

9.2.5 Overlapping

Overlapping occurs when two subnodes of a node again have some common subnode. The definitions of aggregation and generalization assumes that the subnodes are disjoint (i.e. we cannot put up agg(“human body”, “skeleton”, “head”, “torso”, “arms”, “legs”) or gen(“trees”, “tall trees”, “short trees”, “dead trees”, “living trees”). But overlapping is easily accommodated. The daughter nodes are supposed to be disjoint only in participating in the same aggregation or generalization relation. Thus, we can write gen(“trees”, “tall trees”, “short trees”) and gen(“trees”, “dead trees”, “living trees”). This allows for potential overlapping.

It might seem that the above strategy does not make it possible to say something about overlapping as such (except for the fact that it might occur), and that we should therefore define a special overlap relation. However, this is not necessary. Our conceptual basis is in fact already strong enough for describing exactly how different nodes overlap. For an illustration of this, look at the situation of fig. 9.1, where we have used the SHM+ notation for aggregation. In (a), A is an aggregation of D, E, F, G. Since B is an aggregation of D and E, A is also an aggregation of B, F, and G, and since C is an aggregation of E, F, and G, A is also an aggregation of C and D. Thus, if specifying all possible (total and disjoint) aggregations, it will always be possible to handle an overlapping case. In our example, we would know that B and C have the common subpart E, and hence that B and C are overlapping parts of A. It should be noted also that we do not need to know exactly what the overlapping part is. If we just know that there is an overlap, but not what to name it, we can use an unnamed node. Thus, we can handle overlapping cases also when we do not know anything about the parts of the nodes that overlap, as indicated in fig. 9.1(b). Here, we only have the vague knowledge that B and C have some common subpart, and still we are able to treat the situation by means of disjoint node aggregation.

The same approach goes for generalization, of course (as well as association, should it be interesting to say anything in particular about overlaps for this construct). The pictures are not exactly pleasing when it comes to their perceptibility, for instance, people might find it unsatisfactory that three blank nodes have to be introduced for a proper treatment of the situation in (b). However, in the next chapter we will give some clues as to how we can present overlapping diagrammatically in a way which is much nicer than the one just seen.

3Associations are not disjoint by default, but if one wants to state something specifically about overlapping, this can always be done
9.2.6 Logical Connectives and Operators

Obviously, the logical connectives should be predefined in the conceptual basis. If necessary, the user can add on temporal operators and other modal operators. We will not bother about the definition of the standard logical connectives here, since their meaning is well known.

9.2.7 User-Defined Relations

Whenever a user feels a need to extend the framework, the concept missed can be defined as a relation in \( \Theta \). For instance, it is very likely that the user finds a need to distinguish between different types of nodes (such as "processes", "data stores", "agents", "resources" etc.). Any of these can be defined as unary relations on nodes. Nodes are supposed to cover those concepts which can be decomposed, or which have an autonomous existence. Also, one can define types of relations which go between nodes, either binary, trinary, anything. If we want to, we can distinguish between hierarchical and non-hierarchical relations, by providing one construct for each. Also, we can distinguish between directed and undirected relations, and one can declare the visibility and inheritance relations for defined constructs. Providing the distinctions which the user most likely will have to make will save her lots of work, in that the framework becomes easier to extend.

9.3 Conceptual Basis and Pan-Presumptionism

The choice of not giving a definite set of concepts, but rather providing a meta-level language by which new concepts can be defined, gives our approach a pan-presumptionistic profile. Having stated earlier that language is presumption, we
must of course admit that the meta-level language also contains some presumptions. Since everything in $\theta$ is relations, one might think that it is very presumptionistic indeed. However, one must not forget the essential fact that $\theta$ is not the modelling language that the user will see — it is only the meta-level language in which new concepts will be defined. The actual modelling language can be populated with any set of concepts.

Predefining some constructs (the hierarchical ones), it must be admitted still that we make some presumptions. But as we argued in chapter 4, the hierarchical constructs defined are very general, and the introduction of abstraction mechanisms is an unavoidable presumption if we are to manage the complexity of most IS problems. Furthermore it must be noticed that no one forces the user to apply the predefined hierarchical constructs. The user could define her own abstraction mechanisms instead, or even write completely flat models for that sake. However, if this were to be the outcome, the work done in this thesis would be rather pointless. The predefinition of the hierarchical constructs is definitely based on the assumption that such constructs will prove useful.
Chapter 10

Diagrammatic Representation

10.1 Introduction

Having suggested a conceptual basis which is powerful in expression, we turn our attention to the external representation. The challenge here is to make the language as perceivable and economical as possible. At the same time, the representation should lend itself to tool support. As noted in chapter 8, diagrams have several important advantages over other styles of representation (tables, text). Consequently, we will concentrate on diagrammatic representation.

As observed in chapter 7 there are two main styles when it comes to the diagrammatic representation of hierarchies: the tree notation and the onion notation. Some languages provide only the tree notation (for instance ER, PhM), some provide only the onion notation (DFD, Higraphs) and some provide a combined notation where the onion notation is used for one construct (aggregation in ERT, ERAE, semantic networks) and the edge notation for the others. The two different notational styles both have their pros and cons. Looking at fig. 10.2 we can state the following:

- For those hierarchies where the nodes on the higher levels are bigger (or more important) than those on the lower levels, the onion notation will be quite intuitive. In particular, this will be the case for hierarchies where the lower level nodes are somehow inside the higher level nodes also in the real world. Obviously, aggregation, generalization, association, and decomposition hierarchies have this property.

- For other kinds of hierarchies, where the lower level nodes cannot be regarded as components of the higher level nodes (as for instance an "is the father of" hierarchy), the tree notation will be more intuitive (since the sons are not parts of their father, and not necessarily smaller or less important).

- The tree notation seems somewhat better than the onion notation when it comes to showing hierarchies with many levels and branches in one picture. Here the onion notation has the problem that nodes must necessarily be smaller and smaller to fit into their parent node, which means that on a
screen of limited size they soon become too small for sensible presentation. Obviously, there is a limit also to the size of hierarchies that can be shown in one picture by means of the tree notation, but this limit might be larger. Since there is no variation in node size, all nodes can be kept reasonably small. However, the argument is somewhat dubious — as can be seen from fig. 10.2 the hierarchy does not become much smaller with the tree notation than with the onion notation, even if the node size here is as small as the smallest of the onion. Especially, the tree notation suffers from the problem that the bottom level is bound to be very wide for big hierarchies.

- On the other hand, Harel [48] points out a major drawback of the tree notation: "In deciding upon a graphical representation for capturing depth and hierarchy, there is a real disadvantage in drawing trees or other line-graphs. These media make no use whatsoever of the area of the diagram: lines and points are of no width, and no advantage is taken of location."

One of the advantages of the onion notation following from this is the possibility to improve expressive economy. In fig. 10.1(a) the fact that "man" and "woman" is a subset of "person" is expressed in the standard semantic network edge notation, and in (b) it is expressed in the Higraph notation. In the former we have to draw five things: the three nodes, plus two subtype relation arcs. In the latter we get away with drawing only the nodes — the relation between them follows from the fact that one is located inside the other. Thus, the onion notation simply saves us one symbol for every subtype relationship in this case.

It must be noted, though, that in cases where there is overlapping, i.e. where some nodes participate in several higher level nodes, we might be forced to repeat nodes with the onion notation (i.e. draw the same node in several places), whereas this can be avoided with the edge notation. Then the onion notation does not imply any gain in expressive power. As will be indicated later, the edge notation is often the best (or even the only possible) notation for complex overlapping situations where we need to express the overlapping explicitly.

However, the onion notation has another major advantage following from the fact that it does not have to use lines for depicting hierarchical relations, namely that lines are being freed for other purposes. In fig. 10.2 there is no obvious advantage
with the onion notation, but here the diagrams contain only hierarchical relations. In fig. 10.3 we have included non-hierarchical relations as well. With the onion notation it is still possible to get an impression of the system structure with a quick glimpse, and though there are some crossing lines and the picture is rather complex, it is possible to figure things out. The edge style notation, however, is a complete disaster, even though we have distinguished the non-hierarchical relations from the hierarchical ones by using different kinds of lines. Since it is very hard to depict non-hierarchical relations without using edges of some kind, the edge style notation soon runs into problems when things become complex:

- Because there are more lines, there will also be more crossing lines. Moreover, for a hierarchy to be easily perceived as such with the edge style notation, the nodes have to be organized in a tree pattern (such as the one of fig. 10.2). However, this means that it will be difficult to place nodes in such a manner that those which relate non-hierarchically get close to each other (because each level in the tree is more or less one-dimensional, whereas it is two-dimensional with the onion notation).

- Lines being used for several purposes, it will be difficult to perceive the system structure even if nodes are arranged in a tree pattern. In the onion notation, a node is always inside its mother node — thus, it will always be easy to see where it belongs. In the edge notation, this connection is only a line, which means that it is not sufficiently emphasized in the diagram compared to non-hierarchical relations.

From the above arguments, our preference goes in the direction of the onion notation.

However, our discussion has suggested that the tree notation is more intuitive for some types of hierarchies, and that it might have some technical advantages when it comes to providing overview pictures of large hierarchies (as long as non-hierarchical relations are kept out) and situations where the hierarchies are not strict (i.e. where there is complex overlapping). Since the two styles of notation complement each other, and since our framework is supposed to be very flexible anyway, it feels sensible to provide both, as alternative choices and even for possible combination.

However, the hierarchical constructs predefined in our language are mostly of the kind particularly suitable for the onion notation, and this is also the notation that we find most interesting. Therefore, we will concentrate on the onion notation in this work and only discuss the edge style rather briefly.

10.2 Onion Notation

First of all we introduce a neat little abbreviation: Hiconion, an amalgamation of Hicon and onion, is taken to mean an onion style Hicon diagram.
Figure 10.2: Two notations for hierarchies
Figure 10.3: Adding on non-hierarchical relations
10.2.1 Sources of Inspiration

As already mentioned, Harel uses the onion notation for his HiGraphs [48] [49]. Thus, the reader might be tempted to ask: Why do we want to invent a new notation — why can’t we simply use HiGraphs? There are two reasons:

- HiGraphs do not have what we would call a strict onion notation.
- HiGraphs only deal with generalization and aggregation.

For a strict onion notation we require that

- every node which is hierarchically subordinate to some other node, should be depictable completely inside this latter node, and
- every node should own some area in the diagram which is not owned by any of its subnodes, so that any point in the diagram which is within an onion of nodes can be taken to refer to one unique node.

First of all, HiGraphs allow overlapping nodes in the diagram, in which case there are areas which cannot belong to one specific node. However, this point can be disregarded for the moment — after all no one is forced to use overlapping nodes, and besides, we have not yet presented any strategy for treating overlapping diagrammatically ourselves. The main violation of onion strictness is found for aggregation. In fig. 10.4(a), featuring HiGraph generalization, all areas can be uniquely coupled to some node (the one within B to B, the one within C to C, and the one within A and not within B or C to A. However, in (b), featuring HiGraph aggregation, B and C together occupy the whole area of the mother blob. Thus, there is no area within the blob which definitely belongs to A rather than B or C. This is also a problem when naming is concerned — to be able to name aggregate nodes in HiGraphs, it is necessary to attach a little “flag” to the outside of the aggregate blob.

The fact that HiGraphs only support aggregation and generalization is definitely a more important point than the violations of the strict onion principle stated
above. Obviously, aggregation and generalization are the most common abstraction mechanisms in modelling, and to the purpose of Higraphs they might be sufficient. However, from our pan-presumptionistic point of view they are certainly not, and we have to be able to represent all the hierarchical constructs of our conceptual basis also externally.

Given the choice between extending the Higraph notation to capture the remaining hierarchical constructs and inventing our own notation, we go for the latter. The motivation for this is that it is difficult to extend the Higraph notation to cover classification, association, and vague composition without losing the advantages that the current Higraph notation has, as will be illustrated later. Thus, our goal in the following section is to create a diagrammatic representation which takes care of all our hierarchical constructs in a strict onion style.

10.2.2 Nodes and Connectives

The main problem facing us as a consequence of the decision of the previous section is:

- How can we distinguish diagrammatically between the different hierarchical constructs and still adhere to a strict onion style?

Assuming for the moment the Higraph blob symbol, various ideas that quickly come to mind can be illustrated. For instance, we could annotate blobs with letters or other symbols, we could use different types of lines to make the distinction, or we could give blobs different shades, as shown in fig. 10.5. However, there are some problems connected to the approach taken here:

- The annotation approach gives up the onion notation's advantage of expressive economy as compared to the tree notation (cf. fig. 10.1, since we have to write "agg" or "gen" or whatever.

- The annotation approach distinguishes hierarchical constructs by means of details which are rather small compared to the node size itself. As mentioned in the first section of this chapter the basic problem of the onion notation is that nodes necessarily become small if we have to depict many levels at once. Then annotations will be even smaller, reducing the perceptibility of the diagrams.

- Using different kinds of lines is also a rather subtle kind of distinction. Since we might have to use five different kinds of lines to distinguish between all the different hierarchical constructs, it will be hard to find line types which are different enough to give a satisfactory symbol discrimination.

- Shading seems to be all right when symbol discrimination is concerned, but the problem is that diagrams become very noisy due to all the darkness, and thus it will be difficult to provide a sensible use of emphasis.
Figure 10.5: Onion notation connective distinction

What we need is a way of distinction which provides easy symbol discrimination without being noisy, and which does not give away the potential expressive economy advantages of the onion notation. Our choice is to use *shape difference*, i.e. different hierarchical constructs will have different shapes. This is a major difference from the representation of Higraphs, where every node has the same shape. With our approach nodes can have different shapes, in fact one and the same node will have different shapes depending on whether you are looking at its parts, its subtypes, its members, or its instances.

Using shapes for distinction, these should be easily discriminated from each other. For this reason, as well as for diagram simplicity, the icons chosen should be rather clearcut symbols. They should also be as intuitive as possible. However, it can debated to what extent visual intuitivity is possible for abstract notions such as our hierarchical constructs. We have to find symbols for the following:

- nodes without any decomposition
- CACA
- vague composition

All in all this amounts to 6 symbols. However, there are several arguments in favour of using the same symbol for plain nodes and vague composition:

- It is still straightforward to make the distinction, because the latter will have subnodes, whereas the former will not.
They are conceptually similar, in that not at all specifying subnodes is the ultimately vague composition. After all, when to stop decomposing is generally a pragmatic choice. The lowest level nodes in some specification will also have subnodes, only that we do not bother to specify them (for instance because we have already reached a level where operations are provided as primitive in the computer environment for which the development is being done).

Moreover, a node is often non-decomposed only because we have not decomposed it yet (for instance because we must analyse the target organization in more detail to be able to do so). Working with some drawing tool, it would be natural to put subparts inside the node as soon as they are found. If plain nodes and vague composition were to have different icons, one would need an icon switch as soon as a subnode is introduced. Using the same icon, this is avoided, resulting in a smoother tool support.

For the sake of intuitivity, we suggest the amoeba icon of fig. 10.6(a) for plain nodes and vague composition — this is supposed to reflect the vagueness of these constructs. Very intuitive indeed, the shape might however be a threat to diagram simplicity. As pointed out by Zusne in [124] complex figures with many bends (such as the amoeba icon) contains more visual information than simple regular figures like squares or triangles, and thus people also need more time to process complex figures. However, our situation is somewhat different from that of Zusne who discusses figure recognition and recollection in general. In our language it is not necessary to fully remember any amoeba shape — it is enough simply to recognize it as an irregular icon (since all irregular icons, although they might differ individually, have the same meaning), because all the other symbols in the language are regular. Still, the shape might be dissatisfying. Talking to colleagues, we have found that some enjoy this icon whereas other do not.

Since we are advocating flexibility, we point out that anyone disapproving of this icon shall be allowed to replace it with a more regular shape. A good suggestion might be to use a pentagon, as illustrated in fig. 10.6(b). Ourselves preferring the amoeba, we will continue to use this icon in the remaining chapters. However, we point out that in some cases, this icon might not even be intuitive (especially if decomposition has reached the lowest level, where the nodes might for instance be subroutines from some library — these might not be perceived as vague). Then it would be wise to use another kind of icon for the lowest level, to distinguish the plain node from vague decomposition.

All the other constructs must have separate symbols. We give classification a triangle (with a reference to Ogden's meaning triangle [80]), aggregation a square (with no intuitive alibi whatsoever — it simply happened to be the symbol left), generalization a diamond1 (with a reference to the symbol for choices in Flowcharts), and association a circle (with a reference to its dynamic interpretation as repetition,

1It might be discussed whether square/diamond is a shape distinction, since the forms are actually equal, one being a 45 degree rotation of the other. However, we will call it a shape difference for the sake of simplicity. Of course, rotation of symbols will be prohibited, so that no confusion between the two constructs can arise.
i.e. loop), all illustrated in fig. 10.7.

For these icons we adapt the composition convention used for generalization in Higraphs, namely that nodes are simply put inside each other. Thus, *the node and the hierarchical connective is one* in our diagrammatic representation: the triangle, square, diamond, circle, and the big amoeba of fig. 10.8 all have two functions:

1. delineating the node N

2. stating that N is a classification, aggregation, generalization, association, vague composition of the subnodes, respectively
10.2.3 Some Diagrammatic Tricks

There are some problems connected with the diagrammatic constructs defined above. In the following sections we will do some tricks to remove or reduce these difficulties.

Reshaping

The onion notation amounts to putting shapes inside each other, and as observed earlier in this chapter, this causes a problem when hierarchies of many levels are to be depicted — nodes are forced to be smaller and smaller to fit into each other, and since screen and paper sizes are limited, this makes it difficult to show large hierarchies in one picture. With our notation, the problem is most notable for the aggregation and generalization icons, and this is not very encouraging, since these are the two most common abstraction mechanisms (as observed in chapter 4 and 7). The shapes of these two constructs make economical packing difficult, for the following reasons:

- Due to the difference of the shapes, there is a significant loss of space when putting nodes of different kinds inside each other, whereas results are better with similar nodes, as can be seen comparing fig. 10.9(a) and (b).

- Even putting similar nodes inside each other, the packing tends to be inefficient if the number of subnodes is not a square number. This is illustrated by fig. 10.10 — with only two subnodes, approximately half the area of the subnode is simply wasted, and the subnodes must be made just as small as if there had been four of them.

The last problem can be solved rather easily by allowing stretching of the icons (as is also done by Harel in [49]). If any rectangle (with a horizontal base line, or with a diagonal base line, respectively) can be used for aggregation and generalization, and any oval for association, then any number of subnodes can be packed efficiently. Notice that the gen icon must be stretched diagonally. Stretching it horizontally or vertically will not improve the situation, because corners are only made long
and narrow, making it difficult to fit something inside. Fig. 10.11 shows how 2, 3, and 5 subnodes can be packed efficiently for aggregation, generalization, and association, respectively.

However, stretching does not solve the first problem, at least not when it comes to aggregation and generalization, which fit badly because one has horizontal and vertical lines, whereas the other has diagonal ones. To make packing really efficient, icons must be reshaped more drastically, for instance by letting all shapes resemble rectangles with horizontal base lines, as indicated in fig. 10.12. The reason why we will not go for this approach here is that the icons end up being so similar that the symbol discrimination might be severely reduced. Moreover, it can be questioned to what extent efficient packing is a goal, since packing too many nodes into one picture will also be a threat to perceptibility. Moreover, there is no point in packing a node completely with subnodes, since we will always need some space also for the node itself (for instance for writing its name). Thus, in the rest of this work we will make use of stretching but no other kind of reshaping. However, reshaping can be noted as an interesting idea for further elaboration.

Semantic Relativism

The node and the connective being one is very fortunate when it comes to expressive economy. However, this comes at a price. Normally, many nodes in large models will be hierarchically constructed in several different ways at the same
time, for instance "bikes" might be generalized from "male bike", "female bike" or from "childrens bike", "adult bike", or from "Norwegian bike", "imported bike", and it might also be an aggregate in several different ways. The ability to show many different such connections at the same time can be called semantic relativism. Conceptually, semantic relativism is no problem to our framework, but diagrammatically it is. Since the node and the connective is one, the user might be restricted to see only one decomposition of a node at a time, but this will surely be felt as very limiting in many cases. Alternatively one could have several diagrammatic nodes for the same conceptual node, but this is not satisfactory either — it looks confusing, and since we have not stated (and might not want to state) any unique name assumption, it might even create ambiguities.

However, it turns out that the problem can be solved rather easily just by adapting the very useful diagrammatic convention of concatenating nodes. This will be done as indicated in fig. 10.13: removing a little piece of the border of two icons they can be put together to form one node. This node can again be concatenated to others, and hence we obtain as much semantic relativism as we might want. No conceptual problems occur from this, since the node can be treated like any other node. As long as we take care not to place any subnode in the opening between two connectives, no ambiguity will result, and thus, we do not need dividers like dotted lines to delineate the areas of the different connectives of a concatenated node.

Non-Total Compositions

Our defined constructs agg, gen, ass, and vag all assume total compositions, and thus, it might be believed that the treatment of non-total compositions would be a problem. However, as stated in the previous chapter, a non-total composition can be expressed simply by adding the something node. This approach can be also be taken for the diagrammatic representation. Fig. 10.14(a) gives an example where we know that "animal" has the subtypes "lion" and "tiger", but that there are also animals which are not lions or tigers. Then "lion" and "tiger" constitute a non-total partition of "animals", which can be depicted simply by adding the something node. Remember that this node can instantiate to anything, i.e. to a set of nodes just as well as a single node. Thus, the diagram does not mean that
there is only one more kind of animals (of which we do not yet know the name) — it means, simply, that the type "animals" covers something more than lions and tigers.

Some people might disapprove of being forced to include an empty node whenever a non-total decomposition is to be modelled. However, we do not want our diagrams to be ambiguous, so the distinction has to be made one way or another. We find it more elegant to insert a blank node than for instance

- to use different shapes for total and non-total decompositions, which would make the number of shapes uncomfortably big (since we have to provide non-total shapes for four different constructs), and which would be counter-intuitive, since the distinction between totality and non-totality is definitely on a lower level of detail than the distinction between the major hierarchical constructs,

- to annotate nodes with "total" or "non-total" (or some symbols with the same meaning). First of all, the expressive economy of this is worse than for the scheme chosen: whereas our approach requires the addition of a symbol
only in the case where the decomposition is non-total, the annotation scheme requires an extra symbol in both cases. Besides annotation is not very good for perceptibility, as stated earlier in this chapter: when nodes become small, annotation becomes even smaller.

Moreover, we think that the introduction of the blank node is a very neat and natural way to depict non-totality, since

- The blank node is a good reminder to the analyst that there might be some incompleteness in the specification, i.e. that some part has not yet been investigated.

- Furthermore, unnamed nodes can be decomposed just like other nodes. Thus, we have a natural place to put things which do not fit into the general scheme already identified, i.e. exceptions.

The latter point is illustrated in fig. 10.14(b). We have some animals, of which we have classified "Leo" and "Lola" as lions, and "Tina" and "Tim" as tigers. "Al", however, does not fit into the identified classes of "lion" and "tiger", but he is definitely an animal. Putting "Al" inside the unnamed node, we automatically make three statements, namely that "Al" is an animal, he is not a lion, and he is not a tiger. Depicting non-totality in some other way, "Al" might be introduced as a direct subtype of "animal" when it was found that he was neither a tiger nor a lion, and this would clearly be misleading. Thus, we can conclude that the chosen style of representation is advantageous in that it becomes easier to put everything on the right level, even when the understanding of the domain is limited. In a way, blank nodes make it possible for the analyst to structure not only his knowledge, but also her lack of knowledge.

Overlapping

Our defined hierarchical constructs $agg$ and $gen$ assume that the subnodes are disjoint. An interesting question is how overlapping nodes are to be depicted within this framework. As stated in chapter 9, we do not need any specific conceptual construct for overlapping, because the fact that some nodes are overlapping can easily be derived from a sufficient set of disjoint decompositions. The simplest thing would be to take the same approach diagrammatically. Merely translating each conceptual construct into one diagrammatic construct, the situation of fig. 9.1(a) would be depicted as shown in fig. 10.15. However, this kind of notation might not be very satisfactory. Nodes which take part in several alternative aggregations must be duplicated (and if there is no unique naming assumption, one must also make it clear that the icons all represent the same node).

The generalization construct in Higraphs also assumes disjointness, and here non-disjoint subsets are indicated by overlapping blobs. However, blob overlaps do not signify anything as such — to state that there really is an overlap, a subblob has to be placed inside the overlapping area. Similarly, a subblob must be placed inside
the non-overlapping area of each blob if we want to make it clear that the set has a part which do not intersect with the other. We can adapt this technique. Then the situation of fig. 10.15 can be drawn as in fig. 10.16.

As Harel [49] states, the potential of this technique when it comes to depicting overlapping is limited — it works well for two overlapping nodes, but anything more than that is likely to be intolerably chaotic. Thus, if we need to model more complicated situations, we might have to resort either to a technique like the one suggested in fig. 10.15 or maybe to the tree style notation, which can have some advantages for overlapping situations, as will be indicated later.

The Higraph approach deviates from our strict onion principle, in that there will be areas which do not belong to any specific node (like for instance the area which is intersected by B and C but which is not within E in fig. 10.16). This is because Harel applies the what he calls the unique contour convention [49]: "..., the only real, identifiable sets are the atomic sets, that is, those represented by blobs residing on the bottom levels of the diagram, containing no wholly enclosed blobs within. Any other blob merely denotes the compound set consisting of the union of all sets represented by blobs that are totally enclosed within it." There are two motivations for this choice: to allow for unique labelling, and to ensure that each set has a unique contour to which relation arcs can be connected. Clearly, the convention fulfils this need. However, it can be criticized on the following points:
10.2. **ONION NOTATION**

- it is counter-intuitive that diagrammatic intersections do not mean anything until there is a subnode within the intersection. Thus, with the Higraph approach, the diagrams of fig. 10.17(a) and (b) refer to the same conceptual situation.

- Expressive economy is reduced by the requirement of providing a blob within every area that must be referred to.

- Accordingly, there is also a waste of area in the picture.

Thus, it might be interesting to investigate other possible conventions for overlapping. What if we adopt a convention opposite of Harel's, namely what we could call the *unique area convention*? With this convention every name will refer to the smallest area around it (instead of the full contour). Now, the naming of intersections and differences becomes trivial — there is no need to introduce subnodes. However, we are faced with a new problem: How do we name the nodes denoted by the full contours? Clearly, this challenge must be taken — rejecting it would simply mean to provide only the non-overlapping diagrammatic technique of fig. 10.15. Thus, we introduce the *combined area/contour convention*:

- A name *within* a Hicon refers to the maximum area which can be reached from the location of the name without crossing any Hicon curves, plus all areas totally encircled by this area.

- A name *on* the curve of a Hicon refers to the area encircled by this curve.

- The end-points of relationship edges follow the same rules as names, i.e. ending within the area we want, or at the curve if we want the area of the whole Hicon.

By this convention our example can be represented as shown in fig. 10.18. The economy of expression and the simplicity of the diagram is improved as compared to the Higraph inspired notation of fig. 10.16 — we have been able to omit two icons (and could also have labelled the now unnamed node containing F and G without introducing any new node in the diagram). In more complex situations, the reduction of the number of icons can be quite essential to the perceptibility of a diagram. Moreover, our approach means that every identifiable area corresponds to a node — thus, diagrammatic intersections always correspond to conceptual intersections, which we think is more intuitive than the Higraph convention.

However, these gains come at a cost, namely that of a more complicated naming convention. The naming convention stated above is probably understandable enough, though — in a way it intuitively appealing that names placed on curves follow curves, whereas names placed in empty areas follow these. However, the problem is that for more complex situations, the combined naming convention stated above does not work! Introducing a new node which is partly on the area of D and partly on the area of E in fig. 10.18, there is nowhere we can put the names of D and E anymore (since the areas now mean the differences between D or E and the newly introduced node, and the curve is already used for naming B). We could always attempt some even more detailed tricks with naming conventions.
to cope with this, but it seems that whatever we do, it is possible to construct a more complex example where unique naming is possible with the less economical Higraph approach, but not by our combined approach.

Being unable to name uniquely every conceptual node reflected in the diagram is probably intolerable. Thus, we are faced with two alternatives:

- abandon the combined convention and reverting to the one inspired by Higraphs, or
- stick to the combined convention, requesting the insertion of extra diagrammatic nodes only in cases where uniqueness is otherwise impossible.

It might be felt that the latter results in a too complicated approach to naming. However, it is straightforward to provide automatic tool support for detecting cases where unique naming is impossible, which means that the user can be notified whenever such a situation has occurred, or if requested, the necessary nodes can even be introduced automatically. In the light of this, we feel that the gain of avoiding the counter-intuitivity reflected in fig. 10.17 and at the same time reducing the number of diagrammatic nodes (i.e. improving expressive economy and diagram simplicity) is worth the cost of a more complex naming scheme for overlapping nodes. After all, the diagrammatic potential for overlapping is limited, and overlapping increases the diagrammatic complexity, so it is likely to be extensively used only by experienced analysts who should be able to handle the naming scheme outlined above. With our scheme, unnamed overlaps and differences correspond to unnamed nodes. Hence, the diagram of fig. 10.19 reflects the conceptual situation of fig. 9.1(b), which we consider the most intuitive conceptual
10.2. ONION NOTATION

Figure 10.19: Unnamed overlaps

interpretation of this picture.

Negation and Other Logical Operators

The intention behind the Hicon framework is not to provide a diagrammatic representation of first order logic, but to visualize hierarchies. However, also in the specification of hierarchies it might be interesting to state what should not be the case in addition to what should be the case. Thus, it might be interesting to look at the potential of our diagrammatic constructs in this respect. As pointed out in [49] the aggregation and generalization constructs correspond to the logical connectives AND and XOR respectively. If we also provide negation, we will have an adequate set of logical connectives.

As observed in chapter 7 semantic networks and Conceptual graphs provide facilities for including negation in a diagram (cf. figs. 7.17, 7.18). The traditional semantic network style does this by introducing negation as an ordinary node (named "Negation") to which other nodes can be linked (edge style). Conceptual Graphs [104] provides negation by putting a ¬-symbol just to the left of the node to be negated. From our point of view, none of these solutions are quite satisfactory. The traditional semantic network depicts negation as a node which looks like any other node, only that its name is "Negation" instead of "Party Committee" or "Sex-Group" or whatever. Negation, being a domain independent concept with a very special function, should clearly be distinguished from ordinary nodes in a diagram by something stronger than a difference of name. In Conceptual Graphs, negation is clearly represented in a way which is very different from that of an ordinary node. However, the problem here is that the diagrammatic perceptibility is not too impressive, negation marks are very small and anonymous and require a very close look to be noticed in large diagrams. We feel that negation (since it changes the meaning of a statement to the opposite) should somehow be emphasized in a diagram.

Since the node is the only construct we have got, our approach will be to depict negation as such. This corresponds to the traditional semantic network approach. However, with the onion notation, statements to be negated will be placed inside a negation node, not linked to it with some edge. To distinguish negation from ordinary nodes we will emphasize it by giving it a grey colour instead of white. Just like semantics networks use the element link to state that some proposition is
Figure 10.20: Negation and inconsistency

a member of the set of false propositions, we can depict negation as a grey circle, whose subnodes, being its members, are stated to be false propositions.

What good is such a construct for making statements about hierarchies? Fig. 10.20 gives some examples: (a) says that A is not the aggregation of B and C. This statement is not necessarily very useful, at least not if the closed world assumption is applied, in which case we could achieve the same effect simply by not stating anything. However, combining negation and empty nodes (something) we obtain the possibility of making quite powerful statements, as illustrated by (b), which says that A is not an aggregation of B and something. Since the something node can instantiate to anything, this statement has the effect of prohibiting the use of B as a component of A at any level. Thus, if the statement (c) is added to our model at a later stage, consistency checking could discover this as an error because the empty node of (b) can instantiate to the set of nodes D, E, F, J, together with which B will make up A, according to (c). For instance, one make statements like “Engines of the type X should not contain any untested components”, and if afterwards someone makes an error and includes an untested kind of component in the detailed specification of the engine (which can easily happen) this can be detected. Thus, we conclude that negation can be quite useful.

In many cases we might also need to express the concept nothing. This can be done by negating a something node, as shown in fig. 10.21(a). For notational simplicity we introduce the convention that this can be written as shown in (b). The use of this construct is exemplified in (c). Here we express that there are three alternative ways of obtaining A: either by putting together B, C, D, E, and F, by putting together B, C, D, E, and G, or by putting together only B, C, D, and E (e.g. a car could have a metal roof, a canvas roof, or no roof at all). It would be possible to express the situation of (c) without negation, but this would have required mentioning each of B, C, D, E three times instead of only one, making the total number of diagram nodes 18 instead of 9. Thus, it can be concluded that negation is also essential to expressive economy and diagram simplicity.

By combining negation and the AND/XOR operators of aggregation and generalization, we can obtain any other logical connective as well as universal and existential quantification, in a way similar to that of Conceptual Graphs. For no-
tational simplicity we will adopt the same conventions for negating `agg` and `gen` nodes as the one shown in fig. 10.21 for the plain/vague node, i.e. instead of putting `agg/gen` nodes inside a grey `ass` node, we will simply colour the nodes themselves grey.

It might also be interesting to include other facilities, such as different kinds of modal logic operators. However, since our main interest is the depiction of hierarchies and not a diagrammatic representation of modal logic, we choose not to elaborate this issue any further in this report.

### 10.2.4 User-Defined Constructs

It is hard to say exactly how to represent the different constructs that the user is going to define. Moreover, the lay-out of user-defined constructs should as far as possible be the user's own decision. However, the choices made concerning the representation of hierarchical constructs leads to some important restrictions on the diagrammatic representation of other constructs. Thus, we should at least provide some general ideas for how the diagrammatic framework can be extended with user-defined constructs.

#### Depicting User-Defined Nodes

Whether connecting the Hicon framework to an already existing language or using it as a basis for a CASE-shell, the user is likely to have several kinds of nodes that must be distinguished (although some of the languages used in IS modelling has only one kind of node, for instance State-Transition Diagrams), for instance processes, stores, agents, events, entity classes, relationship classes. There are several ways of distinguishing diagrammatically between different kinds of nodes:

- line types
  - dotted vs. plain lines for LOTs and NOLOTs in NIAM,
  - dotted vs. plain lines for derived and base classes in ERT
• annotation
  – entity classes vs. value classes in ERT (small black triangle on the latter)
  – entity classes vs. relationship classes in PhM (an extra line inside the latter)
  – keys vs. ordinary attributes in SHM+ (underlining the former).

However, the most common way of distinguishing between different kinds of nodes is undoubtedly by means of icon shape. In our framework we use icon shape for distinguishing between hierarchical constructs, and this means that the possibility of using shape distinctions for other purposes will be limited. Basically, there are two possibilities:

• Refraining completely from the use of shape distinction for other purposes (i.e. using annotation, line types, colour etc.) to distinguish between different kinds of nodes.

• Providing a dual diagrammatic representation where shape distinctions can be used as long as the decomposition of a node is not shown (and possibly even when only a vague decomposition is shown), substituting the user-defined constructs with Hiconions when more precise hierarchies are requested.

The first approach is the cleanest, but the latter definitely have its advantages, especially when the Hicon formalism is supposed to be connected with an already existing language, in which case the lay-out of the language might have to be changed substantially if the first alternative is chosen. The difference between the two alternatives will be illustrated in the next chapter by some examples.

When the first alternative is concerned, the reader might remember that we rejected line types and annotation as means of distinguishing between hierarchical connectives earlier in this chapter. Since the distinction between different kinds of nodes might be equally important, it would be rather strange if we were to advocate this now. Thus, we should make it clear that neither of these are our first choice — in fact we would prefer distinguishing different kinds of nodes by means of colour, which we think would result in a better perceptibility than the other two alternatives. However, colour has other problems, as will be pointed out in chapter 15.

Depicting User-Defined Relations

Just as with nodes, the user can define any kind of relation. If the user defines a new hierarchical relation, it might be appropriate to define a new Hicon for this, i.e. represent it merely as a node. However, for other kinds of relations it is hard to avoid the use of some kind of edge. A binary relationship can be modelled simply by an edge (but if the user wants to, she can of course choose to model it with a node and two edges), and a n-ary relationship will require a node plus n edges.
10.3. **TREE NOTATION**

If several kinds of non-node relationships are defined, we need to distinguish between different kinds of edges. Basically, we have the same possibilities as for nodes, namely line types, annotation, and colour. Moreover, undirected, and directed relationships can be distinguished according to the use of arrows, and for edges with arrows, we can also use different kinds of arrows to distinguish. Whereas line types and annotation were rejected for nodes, we feel that they are quite acceptable for edges:

- Whereas nodes become very small when we depict large hierarchies, thus making annotation or line type distinctions hard to see, it is more likely that edges can be kept sufficiently long for these techniques. This is because it is unusual to look at relations at many different levels at one time. To look at non-hierarchical relations deep down inside some node, it will be usual to zoom in on this node first, thus making line types or annotations perceivable.

- Lines being lines, line type distinctions are particularly appropriate.

Anyway, lines cannot be distinguished by means of shape, and it pictures might be confusing if both edges and nodes were to be distinguished by means of colour. Thus, we will recommend the use of different line types or annotation for distinguishing between edges if necessary.

### 10.3 Tree Notation

#### 10.3.1 Nodes and Composition

With the tree notation it makes less sense to distinguish hierarchical constructs by means of nodes. Since we have to use edges anyway, we might just as well locate the distinction to these, freeing node shapes distinctions to other purposes. However, to retain some visual similarity to the onion notation, we can use our Hicon shapes for the annotation of edges.

Using a tree notation it becomes essential to reduce the number of edges as much as possible. Thus, it might not be wise to base a tree notation on hierarchical constructs that require totality, as we did for the onion notation. Alternatively we could provide both total and non-total constructs. For instance, these might be distinguished by filling the total one and letting the non-total one be empty, just like generalization in ERT (cf. fig. 7.6 — however, we would rather fill with grey than black, to reduce the emphasis somewhat). As will be shown the provision of non-total constructs will also make it easier for the tree notation to deal with overlapping.

Our tree notation constructs are shown in fig. 10.22. Classification is assumed never to be total (as reflected also in our conceptual basis discussions) — thus, only the non-total construct is provided here. The Hicon shapes being used for annotation, it is our intention that they be rather small compared to the nodes of the diagram (which will be present at labels A, A1,...). As can be seen from
Figure 10.22: Tree notation

the figure the annotation shapes also provide a nice junction for forking several lines (which will reduce the total length of lines in the diagram, thus improving perceptibility a little).

The use of the tree notation is illustrated in fig. 10.23 which shows some generalization hierarchies (where we have assumed some user-defined node having the shape of a rectangle). (a) shows a hierarchy with no overlapping. However, the partition of B is non-total. This could alternatively have been shown by adding an empty node. However, the advantage of the approach taken here becomes evident when looking at (b). Here we have shown that the four subsets B, C, D, E of A are overlapping in that they all contain K. As stated earlier (and as stated in [49]), the overlapping of so many nodes would have been very problematic in the onion notation. It would also have been difficult if we did not have non-total constructs in our tree notation — then we would have had to specify which other subsets B, C, D, and E contains of, describing the whole situation in terms of 5 alternative total partitions of A. This would have resulted in a total chaos of edges.

The example indicates that complex overlapping situations are more easily conveyed by means of the tree notation. Generally preferring the onion notation, this is a good motivation for the support of both in our general framework — then one can complement onions with trees when the need for depicting some complex overlapping occurs.
10.4 Diagrams and Pan-Presumptionism

One might think that presumptions belong to the conceptual dimension (i.e. what is supported and what is not), and that the diagrams only reflect the presumptions made by the conceptual basis, nothing more, nothing less. However, this is not true. As mentioned in chapter 8, the diagrammatic representation of some language will almost inevitably emphasize some constructs, at the cost of others. Thus, making some statements look more important than others, and making some statements more easily expressed diagrammatically than others, diagrams can also contain presumptions that are not present in the conceptual basis.

Consequently, it should be our goal that the diagrammatic representation should be as flexible as the conceptual basis itself. Obviously, the user needs to be able to define how the concepts she defines are to be linked diagrammatically, so for user-defined constructs such flexibility is necessary anyway. Also, the user might want merely to change the shapes used, and there is no reason why this flexibility should not be allowed. As mentioned earlier, we expect the amoeba icon to attract some bad feelings, so for this we even suggested an alternative icon ourselves. It might also be interesting to change other icons — especially when the framework is connected to an already existing language where some of the Hicon symbols might have a completely different meaning. In such cases, rigidly insisting on the symbols suggested here could create lots of misunderstandings.

A more interesting question is whether to allow the user to change also the lay-out of the predefined constructs, if this should be requested. From a pan-presumptionistic point of view, this flexibility should definitely be present, for obviously, our choice of diagrammatic representation implies some presumptions. Choosing to use shapes for hierarchical constructs, we implicitly assume that these constructs are particularly important. The user might feel that they are not, wanting for instance to distinguish different hierarchical constructs by means of line types, so that shape
can be used for other purposes.

On the other hand, the assumption that abstraction mechanisms should receive a prominent treatment is the essential philosophical argument of this work. Thus, it might be said that it is not interesting within the Hicon framework to support changes in the diagrammatic representation which imply a reduced emphasis on hierarchical aspects. However, it should be noticed that all the ideas presented here on the onion notation as such will be applicable also if something else than shapes would be used for distinguishing the different Hicons (for instance line type, annotation, colour). Thus, allowing for flexibility does not mean to let users completely by-pass the work done here.
Part IV

Evaluation
Part Introduction

This part tries to evaluate the applicability of the Hicon framework, and the quality of its diagrammatic representation (in particular, the onion notation).

Chapter 11 discusses the two general uses of the framework: as a basis for a case-shell, or as a set of constructs which can be added to an existing language. Some of the languages discussed in chapter 7 are revisited from this point of view.

Chapter 12 suggests how the framework could be coupled to the modelling language PPM to yield larger uniformity between the static and dynamic parts. Moreover, we indicate how we believe Hicons can contribute to solving the process description problem of PrM. Examples from an existing solution to the IFIP Conference Case are used for comparing the traditional and Hicon-based representations.

Chapter 13 suggests similarly how the framework could be coupled to the external modelling language of TEMPORA. Special attention is given to the hierarchical treatment of business policies. In addition to discussing the coupling between rules and the static and dynamic hierarchies, we present some interesting hierarchical relations between individual rules.
Chapter 11

General Applicability

As stated earlier, there are basically two things we can do with the Hicon framework:

- add the hierarchical constructs to existing languages, or
- let the framework be the basis for a CASE-shell where new languages can be defined.

Of course, it is also possible to combine these two alternatives, i.e. making a CASE-shell featuring both Hicon extensions of existing languages and possibilities for defining new languages. In two subsequent sections we will try to estimate the framework's potential for the two alternatives above by looking at some small examples.

11.1 Adding Hicons to Existing Languages

The addition of Hicons to an existing language will of course change this language to some extent. Thus, some reeducation is required from people who are going to use the language (and who were familiar with the previous non-Hicon version). It might also require extensions of already existing tools. All in all, extending a language with Hicons can be a significant effort. Consequently, there must also be gains resulting from the extension to justify such an effort. Considering whether a Hicon extension could be worthwhile for a particular language, there are two questions that must be asked:

- Does the Hicon extension increase the expressive power of the language?
- Does the Hicon extension improve the external representation of the language?

Obviously, if the answer is "No" to both these questions, a Hicon extension for that particular language does not appear to be interesting. In the following we
will consider the potential for adding Hicons to some of the languages looked at in chapter 7.

11.1.1 SHM+

Expressive Power

The constructs of aggregation, generalization, and association are the same as those provided by SHM+. Hicons also has constructs for explicit classification (whereas this is only implicit in SHM+) and vague composition (which is not provided in SHM+). Moreover, SHM+ allows the definition of dynamic hierarchies only to the extent that they are strictly coupled to the static hierarchies, and dynamic aggregation requires an assumption of sequence. Hicons, on the other hand, do not require such a strict coupling between statics and dynamics, and it is possible to deal with parallel operations and operations for which the sequence is unknown.

All in all, it can be concluded that a Hicon extension would lead to some extension of the expressive power of SHM+, especially in terms of increased generality and flexibility.

External Representation

SHM+ uses a typical edge style notation for its hierarchies. Since the Hicon framework support both edge style and onion style, one possible benefit of an Hicon extension could be a switch to an onion notation for SHM+. However, would such a change make the diagrammatic representation better? To be able to give any answer to this, we will compare the two notations on some small examples, particularly concentrating on the coupling between statics and dynamics, using the following diagrammatic conventions:

- Object nodes will have full lines, operation nodes dotted lines.
- Keys will be distinguished by having underlined names.

With these conventions, the SHM+ diagram of fig. 7.7, which is repeated in fig. 11.1(a) will look like 11.1(b).

SHM+ does not depict operations as separate nodes but puts them together with the static ones. With Hiconions we could have separate nodes for operations, relating these to the static nodes by edges, in which case the traditional representation of fig. 11.2(a) turns into the one in (b). However, if one is not interested in separate nodes for operations, one could of course allow for writing these inside the static nodes, obtaining the diagram of (c), which is more compact.

The issue of diagrammatic representation boils down to the following:

- Is it advantageous to be able to write SHM+ onions instead of or in addition to trees?
Figure 11.1: Traditional and Hiconion SHM+ aggregation

- Is it advantageous to be able to give operations separate nodes?

As stated early in chapter 10, the onion notation is mainly advantageous when other relations must be depicted in addition to the hierarchical ones. However, the only such relation dealt with in SHM+ is the coupling between operations and objects, and due to the limitations on dynamic modelling, no edge is needed for this. Due to the same limitations, there is no need for separate operation nodes, which tend to make the notation less compact. Notice however, that when several operations are to be defined for each component (which might often be the case), the comparison of compactness gradually changes in our favour. SHM+ has to redraw the static structure for each operation (thus repeating the names of object nodes), whereas the separate node notation lets us get away with drawing it only once and connecting different operations to it. This is illustrated in fig. 11.3 where we perform 4 different operations on the same structure. Although in a way more economical, this way of doing it is likely to require crossing lines — thus, people might prefer to redraw the static hierarchy 4 times instead (i.e. use the standard SHM+ notation).

Conclusion

A Hicon extension of SHM+ might be interesting if one wants to increase the expressive power of the language. If one does not want an increase in the expressive power, a Hicon extension does not seem very interesting, since the limitations of SHM+ are of a kind which by and large eliminate the advantages of the onion notation.

11.1.2 DFD

Expressive Power

As observed in chapter 7, the only hierarchical construct of the DFD language is the vague composition. Thus, a Hicon extension would represent a significant
Figure 11.2: Objects and operations: alternative representations
11.1. ADDING HICONS TO EXISTING LANGUAGES

Figure 11.3: Several operations on one structure

increase in expressive power, particularly in allowing for a higher level of precision in describing process decomposition.

External Representation

The information encoded in the more precise hierarchical construct not provided by the DFD language, tends to be of a kind which has to be represented sooner or later anyway. Due to the poor expressive power of the diagrams themselves, DFDs are usually accompanied by other language facilities (for instance structure charts, flowcharts, and/or decision tables for describing process logic). With an Hicon extension, some of this information could be conveyed within the DFD itself. This has the obvious advantage of reducing the need for jumping from window to window in a tool, shifting the attention from one diagram to another.

DFD already uses an onion notation, and the tree notation does not seem to be of much interest to this language (maybe except for showing large overview process hierarchies in pictures where flows are not included). For the sake of familiarity we might depict non-decomposed and vaguely decomposed processes by the standard Gane/Sarson rounded rectangle (with a double top line) symbol instead of the Hicon amoeba.

Conclusion

Hiconions have much to offer DFD style languages, particularly in providing a choice of more precise decomposition constructs. This will be more or less directly illustrated in chapter 12 and 13, where we consider a Hicon extension of PrM (which is itself an extension of DFD).
11.1.3 Petri Nets

Expressive Power

If at all providing decomposition, Petri net dialects are likely to support only one (vague) composition construct, just like DFD. However, Petri nets having much stricter semantics than DFD (and a rather limited flexibility), it may be possible to compute automatically whether the subtransitions of a transition have a XOR or AND relationship, and whether any of them are repeated. Thus, the expressive power might not necessarily be extended very much by the addition of Hicons (except that the classification construct does not exist in Petri nets, but it is hard to see any obvious use for it in this formalism). But a Hicon extension would make modelling with Petri nets far more flexible. Whereas the problem with DFD is that it is impossible to be precise, Petri nets make it difficult to be vague. One has to write down all details at once instead of adding knowledge more gradually, which might be a more comfortable way of working. One example of this is that one has to know whether transitions are sequential or parallel (and if sequential: what the sequence is) to be able to draw a correct Petri net. With a Hicon extension, this is no longer a problem. Fig. 11.4(a) shows the high level transition t from the place P to the place Q. (b) shows that the transition t is an aggregation of four lower level transitions, t_1, ..., t_4, but it does not say anything about the ordering of these. Finally, (c) gives the full Petri net picture of the situation, revealing that t_2 and t_3 are parallel, whereas t_1 goes before these and t_4 after.

External Representation

Whereas it is possible to figure out whether transitions are AND'ed or XOR'ed, and whether they are repeated or not, it is certainly not easy — one has to follow links around, and if there is some logic (for instance pre- and postconditions as in BNM) describing the transitions, one also has to interpret these to make sense of what goes on. Consequently, Hicons have much to offer when it comes to the diagrammatic presentation of Petri nets. One example is already shown in fig. 11.4.

Fig. 11.5 gives an example of how we can use the generalization construct. The three transitions t_1, ..., t_3 in (a) have the same input place and are thus three mutually exclusive alternatives for handling the token in the place. Often such transitions will be different specializations of the same task. By means of the GEN-operator this can be shown as (b). If we generalize also the three output places into one higher level place, the whole situation can be shown as in (c), where we have abstracted away the details of the three alternatives completely.

Due to the semantic strictness of most Petri net dialects, the irregular amoeba shape might feel a little inappropriate in this language. An alternative might be to use the more standard rectangle for transitions\(^1\) and circle for places. Then, the icons for aggregation and association would also have to be changed to avoid

\(^1\)The even more standard solid bar icon has the disadvantage that it is rather unsuitable for supporting decomposition diagrammatically.
Figure 11.4: Aggregation in Petri nets
confusion. If really suggesting a Hicon extension to some Petri net user, we might use a triangle for aggregation (given that classification is not of interest), the ordinary diamond for generalization, and a pentagon for association, thus allowing the user to keep the rather familiar symbols of rectangle and circle for transition and place (plain and vaguely composed).

Conclusion

A Hicon extension for Petri nets looks interesting. Although there is not necessarily any substantial increase in expressive power (at least not for Petri net dialects already providing some decomposition), modelling is made much more flexible in that different constructs are made more independent.

It seems that a Hicon extension can improve the external representation of Petri nets significantly. The introduction of aggregation, generalization, and association icons will make it much easier to see where the choices are, where the sequences and parallels are, and where the repetitions are. Moreover, information hiding is facilitated.

11.1.4 ER

Expressive Power

As noticed in chapter 7, the traditional ER language is not very powerful when it comes to abstraction mechanisms, supporting only aggregation (which is, moreover, restricted to two levels only and typically oriented towards data modelling). The poor abstractive power of ER is also pointed out in [49] where it is illustrated how the Higraph constructs for aggregation and generalization can be added to
the ER language to extend its abstractive power.

Diagrammatic Representation

ER uses a typical edge style representation. In [49] it is illustrated how an onion notation for ER can be provided by means of Higraphs. The same example can also be done in Hiconions. For the purpose of a later comparison with Higraphs, we also show the Higraph version.

The ERD of this example is shown in fig. 11.6, and the Higraph extension in fig. 11.7. Both these are adapted from [49] which points out that the ordinary ER diagram of fig. 11.6 makes no difference between hierarchical relations (like “is-a”) and non-hierarchical relations (like “can fly”). Adding Higraph constructs to the language, this can easily be achieved. Fig. 11.7 shows the class “employees” partitioned into secretaries, pilots, and others. Dates are aggregates of months and years. The Hiconion for the same diagram is given in fig. 11.8. Here, entity classes have been drawn with solid lines and relationship classes with dotted lines. Again, this might not be the most convenient representation to suggest to experienced ERD users. Aggregation (at least if stretched horizontally, as here) is the same as the standard entity class symbol, and generalization is the same as the standard relationship symbol. Thus, some people might prefer a representation where these symbols retain their original use (i.e. plain nodes, possibly also vague composition), forcing us to use for instance a triangle for generalization and a pentagon for aggregation.
Figure 11.7: Higraph extension for the ERD

Figure 11.8: Hiconion equivalent to the Higraph extension
Conclusion

Hicons definitely has much to offer the ER language (and hence also many of its spin-offs), both when it comes to expressive power and diagrammatic representation.

11.1.5 STD

Expressive Power

As pointed out in [48] [49], traditional state transition diagrams are very poor when it comes to abstraction mechanisms, and Higraphs have been used to remedy this weakness. Of course, Hiconions can be used similarly.

External Representation

In [48] [49] Harel has shown how the addition of abstraction mechanisms also makes the diagrammatic representation of STDs much nicer, especially by reducing the number of edges. A similar effect would result from a Hicon extension of STDs. Again we illustrate both approaches for the purpose of a later comparison. Fig. 11.9 shows a traditional STD. Fig. 11.10 shows the equivalent Higraph extension (i.e. Statechart). Both these figures are adapted from [49]. Fig. 11.11 shows the equivalent Hiconion — since there is only one kind of node (state) and one kind of edge (transition) we do not need any line type distinctions.
Figure 11.10: Higraph extension of the ST diagram

Figure 11.11: Hiconion for the Higraph extension
11.1. ADDING HICONS TO EXISTING LANGUAGES

Conclusion

Hiconions definitely have much to offer State-Transition Diagrams, both when it comes to expressive power (in terms of abstraction mechanisms) and diagrammatic representation.

11.1.6 Semantic Networks

Expressive Power

As stated in [17] many semantic network dialects are rather unclear semantically, for instance in that no distinctions are made between general hierarchical relations like part-of, member-of, subtype-of and more domain dependent relations. Thus, expressive power and clarity might be enhanced by the introduction of Hicons. However, there are semantic network dialects which already provide all the hierarchical constructs of our framework. For these Hicons would be interesting only if they could improve the diagrammatic representation.

External Representation

Most semantic network dialects having a typically edge based notation, or a combined notation (conjunction spaces have an onion notation, whereas generalization and association is represented by means of edges). Often they suffer from the same weaknesses as other edge based languages (as pointed out in [49]). Clearly, Hiconions can make a contribution to solving this problem. Fig. 11.12(a) shows the Hiconion version of the semantic network of fig. 7.17. Compared to the standard notation we have had to draw four extra nodes (the blank ones), but at the same time we have been able to eliminate five edges. In an example this small it is hard to determine whether this is a benefit or not, but with larger examples, crossing lines would tend to be a problem for the traditional representation, making our way the preferable. Moreover, we would argue that our notation lends itself more easily to natural tool support by zooming in and out to see more or less detail. This is indicated by 11.12(b), which expresses what should be the case, namely that the paper is written and reviewed by disjoint subsets of persons. Both (a) and (b) can be obtained in a natural way from the more abstract (c), which only says that papers are written by persons and reviewed by persons.

Conclusion

Although semantic network formalisms including our hierarchical constructs already exist, a Hiconion version might be interesting for the improved diagrammatic clarity resulting from the switch from edges to nodes in the representation of hierarchical relations.
11.1.7 ERAE

Expressive Power

After the inclusion of the context construct, it seems Hicons would not extend the expressive power of ERAE significantly. The only things Hicons have which cannot be found in ERAE is the classification and vague composition constructs. Explicit classification would be interesting in ERAE if meta-level reasoning capabilities were requested, otherwise not. The vague composition construct could be helpful in modelling vague knowledge, but this would have to be further investigated.

External Representation

ERAE uses a combined notation for hierarchies: onions for aggregation and edges for generalization and association. A Hiconion extension would make a pure onion style possible, and this might have advantages in cases where lots of crossing lines would otherwise reduce diagram perceptibility. Moreover, ERAE currently suffers from the same weakness as many semantic network dialects: generalization and association are depicted just like ordinary domain dependent relations. However, again it is a problem that Hicons clash with the predefined symbols of the language: Rectangles are used for entity classes, and ovals for event classes. Thus, an adaption of Hicons to ERAE might force us to change the shapes for aggregation and association.

Conclusion

Hicons would not contribute significantly to increasing the expressive power of ERAE. The possibility of changing the external representation might be interesting to improve diagrammatic clarity.
11.1.8 Summing up

From the above discussions we have seen that Hicons have a rather general applicability — we have looked at many rather different kinds of languages, and for all of them, a Hicon extension has been possible. For some of the languages Hicons had much to offer, for others not so much (these were basically the languages that already were strong on abstraction mechanisms).

For some of the languages we noted that Hicon symbols clashed with the current use of icons in the language (in that squares, diamonds, or circles already were in use for other purposes). If making a Hicon adaption of one specific language, we might of course adapt to the current symbol use by finding new icons for the hierarchical constructs.

However, a more ambitious goal would be to build a CASE-shell supporting something like all of the languages considered above, using these together for one integrated approach. In this case we should be more reluctant to adapt Hicon symbols to each language, because one of the major advantages of using Hicons as the basis for such an integrated approach would be that hierarchical constructs will look the same for all languages in the tool set. Such consistency in the use of symbols should obviously be strived for, since it would be much easier for the user to learn one uniform hierarchical representation than to remember that a symbol used for aggregation in one language is an entity class in another and an external agent in the third. Thus, we note that

- Hicons have potential as a basis for a CASE-shell featuring existing languages as well as the possibility of defining new ones.
- One of the main advantages of Hicons in this respect is its ability to enforce a uniform representation of hierarchies upon the languages supported, so that the diagrammatic representation of hierarchical construct will be consistent throughout the toolset.

In the light of this, Hicon extensions will even be interesting for languages whose expressive power on hierarchies is already sufficient.

11.2 A Comparison with Higraphs

From the conclusion above it seems that Hicons have a substantial potential as a CASE-shell basis. Still, we cannot claim to have justified its existence, since there might be other languages which do the same thing in a more elegant way. As already indicated, Higraphs provide a generic hierarchical language framework similar to Hicons. Thus, to justify that Hicons are useful, we must either

- show that Hicons provide something more than Higraphs, or
- show that Hicons the same things as Higraphs more elegantly.

These two points will be discussed in the following two sections.
11.2.1 Something More than Higraphs

Hicons are definitely stronger than Higraphs in that

- Higraphs provide only aggregation and generalization, whereas Hicons also include association, classification, and vague composition.

- The Higraph notation assumes that only one kind of node is going to be decomposed. Compare for instance the examples of fig. 11.7 and 11.8. The Higraph notation works well here as long as we only want to decompose entity classes, but what if we also want to decompose relationship classes? It is straightforward to do this in Hicons, but Higraphs would suffer from a lack of symbols.

However, we must admit that this problem could very easily be solved in the Higraph framework, either by using different kinds of lines, just like the Hicon example, or by defining generalization and aggregation for diamond icons in a way similar to that of blob icons, as suggested in fig. 11.13.

Since it is so easily solved, we will not count the last problem of Higraphs as a point on our score. But the existence of Hicons is definitely justified by the provision of classification, association, and vague composition.

11.2.2 Elegance

For a comparison of elegance, there are some points that go in the favour of Hiconions:

- Higraphs allow diagrammatic overlapping only for generalization, whereas Hiconions can overlap any construct. This follows from the fact that a strict onion notation is provided only for generalization in Higraphs, whereas Hicons provide it for all the constructs. Thus, the examples on overlapping aggregation shown in figs. 10.16, 10.18, 10.19 would require duplication of nodes to be shown in Higraphs.
• We conjecture that it is easier to discriminate visually between a square and a diamond than between a blob containing a dashed line and a blob not containing a dashed line (cf. "dates" in ffig. 11.7 and 11.8).

• By allowing for a tree style representation where this is wanted, Hicons can make it easier to depict models for which trees are advantageous.

However, other arguments go in the favour of Higraphs:

• the Higraph representation is not shape-consuming in the same way as Hicons, and thus, Higraphs are more easily adaptable to the standard notation of many languages (at least if extensions along the line of fig. 11.13 are supported), as indicated by ffig. 11.7, 11.8, 11.10, 11.11, where the Higraph versions are obviously most similar to the standard notations for ER and ST diagrams.

• The Higraph notation is more compact. This is illustrated by ffig. 11.10, 11.11, where the Higraph version does not need separate nodes for the states A and D due to the dotted line symbol. For Statecharts there does not seem to be any documented need for the additional hierarchical constructs of the Hicon framework (association, vague composition, classification), neither does there seem to be any need for abstracting transitions into higher level transitions. (Having nodes for the transitions, which would be possible in the Hicon language, would also have increased the complexity of the diagrams significantly, since there would be approximately twice as many links as there are in a Higraph STD).

It appears that for the kind of languages where the limitations of Higraphs are appropriate, Higraphs might be a better choice than Hicons. This is particularly evident for STDs, for which Higraphs were initially developed.

11.2.3 Conclusion

Having compared Hicons with the similar existing Higraph framework, we conclude that the existence of Hicons is justified because of a larger expressive power, in providing association, classification, and vague composition. However, sometimes this increased expressiveness is not necessary, and in such cases Higraphs might be the better choice. Thus, Hicons do not make Higraphs superfluous. Neither has this, at any time, been our intention.
Chapter 12

Hicons and PPM

The modelling language PPM (Process& Phenomenon Model) consists of two sub-languages: PhM (The Phenomenon Model) for statics and PrM (The Process Model) for dynamics. The purpose of this chapter is to show how PPM could be extended with Hicons, and why we believe that it is interesting to do so.

The main motivation for a Hicon extension of PPM is to increase the uniformity of the language (both conceptually and diagrammatically) when it comes to the use of abstraction mechanisms. In the following two sections we will discuss PhM and PrM separately. Our ideas will be illustrated by means of extracts from a solution to the IFIP Conference Case given for the traditional PPM. Finally we will draw some conclusions concerning the positive and negative effects of introducing Hicons in PPM.

12.1 Statics: PhM

First we will describe PhM as it is today, and then we will discuss how Hicons can be introduced. The traditional and the new representation will be compared by means of the IFIP Conference Example.

12.1.1 PhM as of Today

Basic Philosophy

The Phenomenon Model [107] [60] (henceforth abbreviated PhM) is a formalized extension of the ER language. As opposed to ER and many of its data model spin-offs, PhM is a reality model. The argument for this is that it distinguishes between real world phenomena and information about real world phenomena. The three basic constructs of the language are the following:

- phenomenon classes
- datatypes
• information objects

The formation of phenomenon classes is based on the recognition of similarities between individual phenomena. Similarities are reflected in that the members of a class have some common properties. Datatypes are classes of symbols which can represent individual properties. It must be noticed that a clear distinction is made between the phenomena as such and the properties of the phenomena. Thus, one would not say that an individual of the phenomenon class “PERSON” consists of its “WEIGHT”, “HAIR COLOUR” etc. (which might be done by ordinary data models). An information object, on the other hand, can be said to consist of the datatype elements which are connected to it — thus, an information object is similar to an ordinary entity in a data model. Information objects can be collected in information sets, which can be related to phenomenon classes. Information sets reflect the information which is going to be handled by the information system, whereas the phenomenon classes simply reflect the relevant part of the real world, as perceived by the analysts.

Relations

There are three kinds of relations between phenomenon classes and datatypes, namely

• identifier, which expresses that the individuals of the phenomenon class have unique values for the property,

• attribute, which expresses that the individuals of the phenomenon class have (not necessarily unique) values for the property, and

• quality, which expresses that the class itself, not the individuals, has a value for the property.

Two different kinds of phenomenon classes can be distinguished:

• entity classes.

• connection classes.

The former are the parallel to ER entity sets, and the latter the parallel to relationship sets. The general relation between the two is the binary relation coordinate from connection class to entity class. A connection class will be the Cartesian product of all the entity classes that are its coordinates. Thus, this is mathematically equal to the standard definition of aggregation. There are four specializations of the coordinate relation:

• the from relation (meaning that some members of the entity class must appear as first coordinates of the connection),

• the domain relation (meaning that all members of the entity class must appear as first coordinates of the connection),
• the to relation, and (meaning that some members of the entity class must appear as second coordinates of the connection), and

• the range relation. (meaning that all members of the entity class must appear as second coordinates of the connection).

In addition, abstraction mechanisms corresponding to generalization and association are provided by the relations

• subset (e.g. the phenomenon classes MEN and WOMEN are subsets of the class PERSONS), and

• member (e.g. the phenomenon classes MEN and WOMEN are members of PERSONELGROUPS).

Generally, the convention is taken that subclasses inherit the identifiers and attributes of the superclasses. To make exceptions from this scheme, one can state that an attribute is non-applicable to a certain class. It is also possible to state that attributes are mandatory (i.e. that every individual of the class must have a value for the property) or optional (i.e. that individuals may or may not have a value for the property).

The IFIP Conference Example

Phenomenon classes are represented as rectangles. Entity classes have plain rectangles, whereas connection classes have non-filled triangles in the bottom corner. Datatypes are represented as circles, and information sets as parallelograms. The different relations between phenomenon classes, information objects, and datatypes, and between connection classes and entity classes, are distinguished by means of annotation along the arcs.

The diagrammatic representation\(^1\) is illustrated in fig. 12.1, (phenomenon classes and their interrelationships) and fig. 12.2 (a more detailed picture with datatypes included). Both figures are adapted from the solution to the IFIP Conference Case given in [10].

12.1.2 PhM with Hicons

Conceptual Issues

Hicons would not introduce fundamental conceptual changes in the PhM language. As indicated in chapter 7, the basic abstraction mechanisms of aggregation, generalization, and association are already supported. However, it can be noticed that aggregation is provided by the connection class construct, which does not

---

\(^1\)except information sets, for an illustration of this, see fig. 7.2
Figure 12.1: A PhM Diagram for the IFIP example

Figure 12.2: The properties of Response
distinguish between *part-of* relationships and other kinds of relationships. Coupling Hicons to the PhM language, it becomes straightforward to provide this distinction: using the `agg` construct only for cases where there is a *part-of* relationship between each of the subnodes and the supernode (e.g. that FRAME, SEAT, HANDLEBARS are parts of BICYCLE), and using connection classes for other relations (e.g. that BICYCLES are owned by PERSONS).

Two new constructs will be introduced though: classification and vague composition. Classification is interesting if meta-level reasoning capabilities are needed in the system to be specified, and vague composition can be helpful when the knowledge of the domain is uncertain.

Since PhM maintains a clear distinction between phenomena as such and their properties, datatypes should not be modelled as parts of the corresponding phenomenon classes (which is often done in data modelling). Information sets, on the other hand, can be treated as aggregations of the participating datatypes, since an information object is nothing but a tuple of values.

**Diagrammatic Issues**

PhM uses a typical edge style notation for the abstraction mechanisms it provides. The Hicon edge style constructs described in chapter 10 could be introduced without any essential changes of notation. However, it is more interesting to take a look at the possibilities for using the onion style, both because of the advantages of this style stated in chapter 10 and because using the onion style for PhM would allow for more uniformity with *PrM* (which uses the onion style for decomposition).

For the onion style we need to distinguish between four different kinds of nodes: entity classes, connection classes, datatypes, and information sets. As stated in chapter 10, the fact that we use shape to distinguish between different Hicons means that we have to use other means for distinguishing between different semantic categories of nodes. Here we use the following conventions:

- phenomenon classes will have solid lines, and entity classes and connection classes will be distinguished by the convention that the latter is given an extra vertical line close to the bottom of the icon.
- information objects will have dotted lines.
- datatypes will have solid lines and an extra horizontal line at each side.

The notational conventions stated above should be considered as a basis for illustration and experimentation, rather than a concrete suggestion for a new way of writing PhM diagrams.

**The IFIP Conference Example**

A Hiconion version equal to the diagram of fig. 12.1 is shown in fig. 12.3. People familiar with the traditional notation and not with Hicons would probably find
the former the most appealing, and it might be hard to see any obvious advantage of the latter notation. However, a simple count of symbols in the two diagrams reveal a substantial achievement on the side of Hiconions:

- All in all (counting nodes, edges, and text strings), the traditional diagram uses 100 symbols, whereas the Hiconion version uses only 78 in expressing the same knowledge, i.e. a reduction of 22%.

- Most important is the reduction of the number of edges: the traditional version has 23, whereas the Hiconion version has only 12, i.e. a reduction of 47%. This exemplifies the argument made in chapter 10 on the advantages of an onion notation.

Furthermore, the example is fairly small, and there are no crossing lines even in the traditional version. We believe that the advantages of the Hiconion version would be more apparent (even to those fond of the traditional notation) with more complicated models where crossing lines would seriously decrease the perceptibility of edge style diagrams.

Still, it must be admitted that the perceptibility of a diagram is not only dependent on the number of symbols. Thus, we cannot say anything definite about perceptibility improvements on the basis of these examples — to be certain, we would have to perform thorough psychological experiments (which is beyond the scope of this work). And certainly, a reduction in the number of symbols is generally promising when it comes to improving perceptibility.

Moreover, the Hiconion notation is intuitively appealing in that subsets have smaller icons than their supersets, and subsets are located within the supersets (whereas none of these points apply to the traditional representation). A side-effect of this is that the diagrams directly facilitate zooming in and out to see more or less detail — zooming away from the diagram of fig. 12.3 one could for instance obtain the diagram of fig. 12.4, where the distinction between different classes of Papers is no longer visible.

The Hiconion parallel to the diagram of fig. 12.2 is given in fig. 12.5. Here the number of edges has been reduced from 14 to 6 (57% reduction), and the total number of diagram symbols from 58 to 42 (27% reduction). Moreover, it can be noticed that composition of complex value types is only provided textually in the traditional diagram, as indicated for the datatype “Front Page” (the star symbols denote repetition). With Hiconions we would be able to show also this decomposition diagrammatically, as indicated in fig. 12.6. Introducing this notation for composite datatypes does not reduce the number of symbols, but it may result in a clearer presentation.

12.2 Dynamics: PrM

In the following two subsections we will describe the PrM language as it is today, and how we envision the PrM language extended with Hicons, respectively. The
Figure 12.3: Hiconion for the previous PhM
Figure 12.4: Slightly zoomed-out Hiconion for the same system
Figure 12.5: The properties of Response

Figure 12.6: Datatype decomposition
representation styles will be compared by extracts from the IFIP Conference Case solution specified in [10].

12.2.1 PrM as of Today

Basic Constructs

The PrM (Process Modelling) language was defined in [9] and modified in [83]. The basic concepts of the current PrM are

- processes,
- flows,
- stores,
- external agents, and
- timers.

The meaning of the four first are approximately the same as in the standard DFDs, but it should be noticed that flows and stores can contain material as well as information, i.e. PrM is intended to be used for reality modelling rather than only the modelling of data processing. Timers can send signals at specific time points or provide delays of a specified duration.

Items and Events

PrM contains two important underlying concepts which are not directly visualized in the diagrams, namely items and events. The item is the link between the static and the dynamic world — items are what is lying in the stores, and what is moving on the flows, and they correspond to some real world phenomena (which could be defined in the phenomenon model). An item has a material aspect and an information aspect; one or both of these may be zero (in the latter case, the item is merely a signal).

Events are instantaneous, and denote that items arrive (to some process, store, or external agent) or depart (from some process, store, or external agent) on some flow. The behaviour of a process, as seen from the outside, can be fully described in terms of the input and output events of the process and the relations between these.

Ports, Triggering and Termination

What is usually called a process is actually a class of what we could call process incarnations. Similarly, each connection point between a flow and a process signals the existence of an event class. Each process incarnation can be described as a set of
12.2. \textit{DYNAMICS: PRM}

![Diagram of PrM port symbols]

Figure 12.7: The PrM port symbols

individual events belonging to the different event classes of the process. Processes can be provided with input ports describing the logical relations between inputs, and output ports describing the logical relations between outputs. These ports define the set of combinations of input events and output events which will make up an acceptable process incarnation. The following ports are provided: \textsc{AND}, \textsc{OR}, \textsc{XOR}, \textsc{COND} (conditional), and \textsc{REP} (repeating), and their symbols are shown in fig. 12.7.

Ports can be put inside each other to form composite ports. The \textit{immediate constituents} of a port P are defined to be the event classes and ports which are within P, but not within any other port N such that N is within P. Following from this, the port semantics can be defined as follows:

- For any incarnation $I$ of a process $P$
  - an event class is said to be satisfied if an event from this class participates in $I$
  - an AND port is satisfied if all its immediate constituents are satisfied,
  - an XOR port is satisfied if one and only one of its immediate constituents is satisfied,
  - an OR port is satisfied if at least one of its immediate constituents is satisfied,
  - a REP port is satisfied if all its immediate constituents are satisfied N times (the exact number N of repetitions can be specified if known), and
  - a COND port is satisfied if all its immediate constituents are satisfied, or if none of its immediate constituents are satisfied.

- Every incarnation of a process must behave such that its outermost input port and its outermost output port are satisfied. (The outermost port is the one which is not the constituent of any other port.)

For example, the process of fig. 12.8 has the 6 event classes "arrivals on A", "arrivals on B", "arrivals on C", "departures on D", "departures on E", and "departures on F". There are two possible input combinations: $(A,B)$ (i.e. 1 A event, one B event), $(A,C)$, and six possible output combinations: $(E)$, $(F)$, $(E,F)$, $(D,E)$,
Figure 12.8: A small composite port example

(D,F), (D,E,F). This yields a total of 12 acceptable event class combinations for the process as a whole.

Input event classes that are triggering for a process are marked with T in the input port. This means that events of this class will occur at the very start of the process execution (i.e. the event triggers the process). Output event classes that are terminating are marked with T by the output port. This means that events of this class will occur at the very end of the process execution (i.e. the events will signal the termination of the process).

The Process Description Problem

It is a well known fact that one cannot go on decomposing forever. This also applies to PrM. Sooner or later the behaviour of a process must be expressed completely in terms of constructs which do not require the introduction of any subprocesses. A problem with the current PrM language is that a satisfactory formalism for this has not yet been found. The following have been tried, without any substantial success:

- using BNM ([108] [60], for a brief description of BNM, see chapter 6) at the lowest level was tried out in the case study of [97], but it was discovered that BNM diagrams tended to be uncomfortably complex.

- using predicate logic pre- and postconditions directly on processes. This idea was tried out in [63] but was not found satisfactory — in addition to a rather poor user-friendliness, there was actually no use for preconditions (since a flow goes to one specific process which has to receive what it receives, whereas several BNM transitions can have the same place as input, the preconditions being used to choose between them).

- using decision tables. A major problem of this was the fact that decision tables, presenting a column for each alternative way of executing the process, necessarily made the ports redundant. Neither was the user-friendliness impressive, and several things (such as sequencing and timing constraints) were difficult or impossible to express.
Thus, the quest for a satisfactory process description formalism in PrM continues.

The IFIP Conference Example

An overview diagram of the activities of the Program Committee is shown in fig. 12.9. Ffиг. 12.10 and 12.11 show the decompositions of P1 and P2, respectively. Fig. 12.12 shows the application of BNM at the lowest level, within the process P1.3. All these four figures are adapted from [10].

12.2.2 PrM with Hicons

In PrM we might be interested in decomposing stores and flows as well as processes. However, since the constructs to be used will be exactly the same, and since nothing but process decomposition is shown the case study of [10] we will concentrate on this. When it comes to process decomposition in PrM, it is our belief that the use of Hiconion will contribute to

- increasing the expressive power and diagrammatic clarity on hierarchical relations between processes, and
- reducing the process description problem

In fact these two uses are not as different as they might seem. As stated earlier in this chapter, a process class can be considered as constructed from a set of event classes. Relating some subprocesses P1, ..., Pn to a superprocess P then amounts to stating some relationship between the event classes of P1, ..., Pn and the event classes of P. On the lowest level we just want to relate event classes themselves to the process which is constructed from them. Since there is no fundamental difference between a set of several event classes and a single event class (only that it feels a little awkward to call the latter a process), describing a process is almost identical to decomposing it (from a mathematical point of view), only that the former has to be somewhat more detailed, so that no further decomposition is necessary.

Using Hicons for process class composition, \texttt{agg(P,P1,\ldots,Pn)} means that the set of possible event class combinations for P is a subset of the Cartesian product of possible event class combinations for P1,\ldots,Pn, \texttt{gen(P,P1,\ldots,Pn)} means that the possible event class combinations for P is the union of the possible event class combinations for P1,\ldots,Pn, and \texttt{ass(P,P1)} means that the set of possible event class combinations for P is a subset of the power set of possible event classes for P1 (we shall only use association with one child in PrM). Thus, the definition are the ordinary set-based ones given in chapter 9, applied on processes by considering them as sets of events.

\footnote{Some details of the original diagrams which were not relevant to our discussion have been omitted for the sake of simplicity.}
Figure 12.9: PrM for the IFIP Conference Case
Figure 12.10: The decomposition of P1

Figure 12.11: The decomposition of P2
Figure 12.12: BNM at the lowest level
12.2. DYNAMICS: PRM

The Hicons above can be used also for process description, then relating single event classes (rather than subprocesses) to their owner process. However, a full process description definitely requires more. One would have to deal with things like the following:

- event occurrence (e.g. "for every instance of the process, if there is an arrival on a, then there must also be a departure on d")
- event sequencing (e.g. "for every instance of the process, departures on d must always be after departures on c")
- event timing (e.g. "for every instance of the process, departures on d must always be within 5 minutes of arrivals on a")
- flow contents (e.g. "the weight of the item departing on the flow d will be equal to the sum of the weights of the items arriving on a and b minus 0.2 kgr")

Of course, one must be able to combine these different aspects in complex relations. Obviously, Hicons alone cannot provide us with the expressive power necessary for dealing with all this — neither should we hope to be able to capture all this knowledge diagrammatically. Hicons on their own can only capture event occurrence relations, but in doing this they still represent a step in the right direction compared to the traditional ports, since Hicons cover all event occurrence relations, whereas the traditional ports were not able to deal with those between input and output.

Moreover, the Hicon framework turns out to provide a very good basis for attaching additional knowledge to the diagrams. Thus, in the following we will first illustrate the use of Hicons for describing event occurrence relations, and afterwards we will indicate how the other kinds of relations can be dealt with.

Event Occurrence

As noted in [49] aggregation and generalization corresponds to AND and XOR, respectively, and just like in SHM+ association will correspond to repetition. Thus, the AGA constructs easily replace the AND, XOR, and REP port constructs of the traditional PrM. The OR port can be dealt with by combining AND and XOR, and the COND port by combining XOR and negation. As mentioned above, Hicons can relate all the events of a process, whereas the traditional ports can only relate input events to input events and output events to output events.

Arrival event classes are defined by flows pointing at the Hicon and departure event classes by flows pointing away from the Hicon. A Hicon relating one or more event classes and or process classes can be called a process class. With the convention that process class nodes have ordinary lines, whereas event class nodes have dotted lines the exact meaning of flows pointing to/from process class nodes can be explained. The notation of fig. 12.13(a) is then just a simplification of the
notation of 12.13(b), i.e. the presence of events is implicitly understood by the connection between a flow and a process node.

For an illustration of the increased expressive power of the Hicons as compared to the traditional ports, consider the example of fig. 12.14. The traditional notation in (a) is only able express that $a$ and $b$ are mutually exclusive, and that $c$ and $d$ are mutually exclusive. However, it might be the case that we always send $c$ when we have received $a$, and always $d$ when we have received $b$. This is impossible to express diagrammatically with the traditional ports. With Hicons, the situation can be clarified, as shown in (b).

This does not mean that it is impossible to relate inputs to inputs and outputs to outputs without also relating inputs to outputs. The diagram of fig. 12.15(a) contains the same information as the one of 12.14(a). The outermost agg node
is introduced because there is an implicit AND in the PrM diagram due to the requirement that a process must have both input and output to be complete. This might also be a threat to flexibility — one might want to model processes which are able to take input without giving output (or vice versa) during one execution. Thus, in (b) we have shown the example where we know that the two inputs are mutually exclusive, and we know that the two outputs are mutually exclusive, but *nothing* about the relation between input and output. By means of the vague composition construct we can be fuzzy whenever we want to. Also here, the flexibility is increased compared to the current port formalism. Assume for instance that our only i/o relation knowledge concerning \( P \) at some point was that the receipt of \( a \) and the sending of \( c \) were mutually exclusive. With Hicons we are able to state this, as shown in (c), without bothering about the relations between \( a \) and \( b \), \( b \) and \( d \), and \( c \) and \( d \).

**Other Relations**

As stated earlier in this chapter, relations apart from those only involving event occurrence cannot be dealt with by the Hicon formalism as such. But this is no surprise or disappointment — the ports of the current PrM give us even less. The idea in PrM is that other information must be given by means of tables or text, in separate windows. Also with our approach we have to resort to text for the most detailed information, but our experiment is to use as few separate windows as possible — the basic idea being that

- *every node in a Hiconion is its own window,*

which is applicable to Hiconions just because of the property that a node always has its subnodes inside it. In the following we will indicate how we envision the presentation of sequencing, timing, and contents relations within the diagrammatic representation of occurrence relations already provided by Hiconions.

**Sequencing and Timing** Sequencing in the current PrM is partly covered by the introduction of triggering and termination marks. However, these only tell you
what the first and last event of a process will be. Thus, there is much uncertainty left when it comes to sequencing. Here we suggest a quite intuitive scheme which can be combined with the Hiconions. The strategy is simply to number the nodes within a node to state the sequence of subnodes if the supernode occurs. If the sequencing is uncertain, one might use lists of alternatives or intervals to suggest this. Fig. 12.16(a) gives an example. Here numbers have been connected to implicitly understood event class nodes. For more complex situations we could also number subprocess nodes within a superprocess. The example says that the first event of the process will always be an “arrival on a” and the second a “departure on d”. Between the third and fourth there is some uncertainty, as “arrival on b” may happen before or after (or maybe at the same time as) “arrival on c”. The last thing which happens is the simultaneous “departure on e” and “departure on f” (these are definitely simultaneous, since they have the same number).

Timing can be dealt with similarly by attaching time points or intervals to various nodes. Fig. 12.16(b) shows an example where the first thing that happens for every execution of the process is an “arrival on a” (i.e. at time 0 of each execution). Subsequently, an “arrival on b” should occur within 5 seconds afterwards, a “departure on c” between 3 and 6 seconds afterwards (i.e. this may happen before the arrival on b) and a “departure on d” exactly 19 seconds after the start of the execution.

We are the first to admit that it is possible to construct sequencing and timing relations so complex that they cannot be dealt with by the scheme presented here. Thus, more sophisticated means for presenting such requirements might also be needed. However, it is believed that also these could be used within the appropriate node. Moreover, if sequencing or timing requirements are so complicated that they cannot be dealt with by a simple scheme like the above, this might be a sign that one should decompose further before trying to state such requirements. Finally, the scheme suggested above is definitely both more flexible and more expressive than the triggering and termination mark facilities of the traditional PrM.
Choice between Alternatives  Hicons will model mutually exclusive actions in PrM by several nodes contained in the same GEN node. Just like sequencing and timing, conditions can be located in the corresponding nodes. Fig. 12.17 shows an example where there are three alternative actions, depending on the relation between the weight of the item arriving on a and the weight of the item arriving on b. If we introduce standard predicates for event occurrence times, we can also deal with situations where the choice depends on sequencing or timing.

Operations  Having discussed conditions for choices, the only thing which remains is to provide something for the what the low level node does if it is chosen for performance. Here we need to describe

- the output in terms of the input (both for information and material), and
- the criteria for selection when reading and updating stores.

It is beyond the scope of this thesis to elaborate in detail a language for making these expressions. However, we will make some suggestion as to how to address the main problem experienced with the decision tables used in [101], namely that they did not work well on high levels, and they did not work well for material. For these purposes we suggest four general primitive operations, which can be used both for material and information:

- JOIN. This operation will have more than one input, but only one output. The meaning is that the input items are put together to form the output item (for instance joining a wooden plate and four wooden legs to create a table, or joining some letters to form a word).

- SPLIT. This is the opposite of JOIN. There will be only one input, and more than one output.
- **MODIFY.** This operation will have only one input and one output and the meaning is that the item is modified (for instance changing a piece of dough to a bread by heating it in an oven).

- **PASS.** This operation will have only one input and one output, and the meaning is that the item is passed on unchanged. An interesting observation is that PASS could be considered a special case of any of the other three (a JOIN with only one input, a SPLIT with only one output, or a MODIFY which is a null-operation). For clarity it seems better to provide PASS as a separate operation — after all it is quite often needed in modelling, for instance in cases where a process takes something from a store and just sends it on to another process.

All these operations will be attached to AGG nodes. Of course, since we advocate flexibility, the user should also be able to define other primitives than the ones stated here. To see the difference between choice conditions, operations, and store selectors, we prefix them with C, O, and S, respectively.

### 12.2.3 The IFIP Conference Example

The Hiconion equivalent of the diagram of fig. 12.9 is shown in fig. 12.18. Again those familiar with the traditional notation and not with Hiconions might feel that the Hiconion diagram does not represent any improvement. However, resorting to our practice of counting symbols, we find that:

- There are 21 process node symbols in the Hiconion diagram. These correspond to processes and ports in the traditional diagram, which add up to a total of 28 symbols (i.e. a reduction of 33%).

- Hiconions are more flexible to flow attachment than the traditional ports. In the traditional notation, input flows have to be attached to the input port and output flows to the output port. With Hiconions, a flow can be connected to a process at any side. The positive effects of this are the following:
  - A reduction of the number of flow arcs due to the possibility of using bidirectional arrows. In our example diagram, this reduction is small, though — from 43 to 41 (because we are able to use bidirectional arrows only in two cases), i.e. 4% .
  - A reduction of the average flow length (because flows never have to go around processes to get in at the right side). For example, look at the three flows connecting “Standard Forms” to P1, P2, and P5. In the traditional diagram these have a total length of 248mm, whereas they only add up to 119mm in the Hiconion version, a reduction of 52% . We must admit that such reductions only occur when some store, document, or external agent has to be connected to several processes, but since this happens quite often, the impact should not be underestimated.

---

3 The lengths are measured in an A4 format version of the report.
12.2. **DYNAMICS: PRM**

- A reduction of the number of flow bends (for the same reasons as the previous points) — again consider the three flows from “Standard Forms”. In the traditional notation, each has two bends, in the Hiconion notation they have only one each.

- A reduction of the number of line crossings (due to the all of the above points). There are 6 in the traditional version, and 0 in the Hiconion version.

  - Since Hicons are not attached to process icons but play the role of process icons themselves, they can be made bigger (and thus more easily readable) than the traditional port icons given the same total diagram size.

All in all, we believe that these factors can improve the perceptibility of complex models significantly.

Another interesting point about the Hicon version of PrM is that we gain methodological flexibility. As observed in chapter 7, the ports reveal something about the hierarchical structure of a process. However, ports are used to relate input and output flows, i.e. they have no meaning when flows have not been introduced in the diagram. Hicons, on the other hand, can express hierarchical relations between processes without flows being present. Hence, Hicons are both more expressive and more flexible than the traditional ports. It has also been suggested that such flexibility is needed:

  - Experience ([65], [68], as well as personal experience not accounted for in any report) has shown that it is often difficult to establish the ports in a strict top-down manner — especially at higher levels, where the ports will be so complex that they can only be found after decomposition and introduction of ports at lower levels.

  - Bubenko [21] reports problems with the use of flows at high levels of abstraction. In the light of this, it is particularly unfortunate that hierarchical knowledge is made dependent on the declaration of flows in the traditional PrM. With Hicons we are able to be make precise distinctions between aggregation, choice, and repetition of subprocesses within a superprocess without introducing flows.

To illustrate the increased independence of flows in the Hicon version of PrM, we give some more examples on the IFIP case. Just like for the traditional PrM we would normally start with identifying the highest level activities, as illustrated in fig. 12.19. From here, both the traditional and the Hicon version can continue in two directions:

  - identify flows between the processes, or
  - decompose processes.

However, the traditional version is able to do the second only in a very unprecise way because only vague composition is provided, whereas the Hicon version also
Figure 12.18: The overview diagram in Hiconions
12.2. DYNAMICS: PRM

Figure 12.19: Identification of processes

Figure 12.20: Decompositions of P2

has AGA. Fig. 12.20 shows what could result if we went on decomposing P2 (cf. the traditional decomposition of fig. 12.10). (a) shows the vague decomposition, and (b) shows the more precise decomposition where we have stated that P2.1 is always performed when P2 is performed, and that there is a choice between P2.2 and P2.3 for each execution of P2. The fact that P2.2 and P2.3 are mutually exclusive within P2 could not be expressed in the traditional version without introducing flows in the diagram.

Thus, whereas the traditional PrM analyst would be more or less bound to perform the following sequence of steps in doing hierarchical decomposition: (1) identify subprocesses (i.e. vague decomposition), (2) identify flows between the subprocesses, (3) introduce ports etc., the Hicon version allows for both (1) identify subprocesses, (2) identify flows, (3) replace vague decomposition with more precise Hicons (which would correspond to the traditional sequence) and (1) identify subprocesses, (2) replace vague decomposition with more precise Hicons, (3) identify flows. The latter sequence is illustrated for the decomposition of P1 (cf. the
traditional diagram of fig. 12.10) in fig. 12.21(a), (b), (c), and in (d) we finally describe the event occurrence relations of the lowest level processes also by means of replacing vague Hicons with precise ones. Fig. 12.21 also illustrates very clearly why we find the amoeba shape for vague decomposition so appealing: the diagrams (a)–(c) give a very intuitive picture of how uncertainty is pushed inwards in the diagram (i.e. downwards in the hierarchy), being replaced with precision as the knowledge of the system gets more detailed. Thus, the level of precision in a diagram is more or less directly reflected by the fraction of regular shapes.

At this level, [10] stops decomposing and describes the behaviour of the processes by means of BNM diagrams. The most complex of these processes is P1.3, for which 5 BNM transitions are used, as shown in fig. 12.12 (taken from [10], pre- and postconditions for the transitions included). Expressing this information by means of Hicons we simply elaborate a little more on the nodes already established for P1.3 in fig. 12.21(d). The result is shown in fig. 12.22, and can be explained as follows: To connect the relevant inputs and outputs, we introduce two AGG nodes inside the GEN node — now, the details about choices and operations can be located inside these lowest level nodes. Recalling the primitive operations suggested some sections ago, we state that both of these lowest level AGG nodes denote JOIN-operations — the name and address is added to the standard contents, resulting in a letter with both name, address, and contents. We also state the conditions for choosing either of the operations. The updating of the Distribution List is obviously a MODIFY operation. Describing it amounts to expressing
what element of the list is selected, and how this element is changed.

Since the standard JOIN operations implicitly assumes that nothing is changed, we get away with taking for granted much of the information that had to be expressed in the postconditions of fig. 12.12. Mostly, the postconditions express that the value of output attributes are equal to the values of the corresponding input attributes, and such expressions are very common when using BNM. If the operations just suggested can relieve the analyst from the effort of writing down such standard statements, this would certainly be a benefit.

The total number of nodes (process node, ports, places, transitions, arcs) and arcs in fig. 12.12 is 36, whereas it is only 12 in fig. 12.22, a reduction of 66% ! Moreover, the conditions become simpler, and instead of being forced to introduce place tokens \((y, z_1, z_2, \text{ etc.})\) in fig. 12.12) we can use the flow names directly. Finally, the operation names JOIN and MODIFY make it easier to get a quick understanding of what goes on in the process than the mere relating of variables provided by the BNM formalism.

All in all, it seems that Hicons are very promising in PrM since the examples have indicated that they function well both for modelling vague knowledge on high levels of abstraction and for modelling details at the lower levels.
12.3 Effects of Introducing Hicons in PPM

12.3.1 Gains

The most obvious features to be gained from introducing Hicons in PPM are the following:

- *increased conceptual uniformity* between the static and dynamic sublanguages when it comes to hierarchical aspects. As of now, PhM supports AGA (but not vague composition) whereas PrM supports vague composition (but not AGA). With Hicons, all four constructs would be available both for statics and dynamics.

- *increased diagrammatic uniformity* between the static and dynamic sublanguages when it comes to hierarchical aspects. As of now, PhM uses an edge style notation for hierarchical relations, whereas PrM uses the onion style. Adopting Hicons it would be possible to use the onion style both for statics and dynamics.

- *simpler diagrams* (assuming an overall use of the onion notation), due to several factors indicated by extracts from the IFIP Conference case study of [10].

- *increased expressive power* by the introduction of classification and vague composition in PhM, and particularly by the introduction of AGA in PrM.

- *increased methodological flexibility* in PrM by making the statement of precise hierarchical knowledge independent of the declaration of flows, and finally

- some contribution to solving the process description problem of PrM.

The increased uniformity will make PPM look much more like one language, instead of a rather loose coupling of two different languages. The fact that hierarchical constructs and notation are common between statics and dynamics would probably make PPM easier too learn.

Although the point about process description definitely requires further research, our examples have suggested that Hiconions have a significant potential in this direction. At least

- it has been made easier to see the connection between input and output,

- Hiconions may enable us to connect sequencing and timing requirements directly to the diagrams, whereas earlier process description formalisms did not offer anything at all for such requirements,

- Hiconions makes explicit all the mutually exclusive choices of a process and has a node for each choice, so that it becomes very easy to associate conditions or probabilities with choices in the diagram,
12.3. EFFECTS OF INTRODUCING HICONS IN PPM

- similarly, the expression of input/output transformations and store selection may be attached to the appropriate nodes, and

- the problems with decision tables and ports (redundancy), logical conditions (no use for the precondition) have been avoided, and the diagrams are much simpler than the equivalent BNM diagrams.

All in all, we feel that the strategy of using Hiconions for PrM process description is very interesting for further elaboration.

12.3.2 Threats

The following problems might result from an introduction of Hicons in PPM:

- Since icon difference is used to distinguish between the basic hierarchical relations, different semantic categories of nodes (like entity classes, relationship classes, datatypes, processes, and stores) have to be distinguished by other means. The best suggestion from a perceptibility point of view is probably colour (but a drawback with this is that it heightens the equipment threshold for using PPM). More subtle means like different line types and annotation, might reduce the perceptibility of these aspects. Moreover, the change of notation resulting from an introduction of Hicons is pretty radical (and might thus be irritating to people already familiar with the current notation).

- The possibility to put much information and different kinds of detail into the diagrams may be misused, leading to chaotic specifications. Undoubtedly, the increased expressiveness also requires an increased tool support for sensible information filtering. Thus, a Hicon version of PPM would probably be harder to commercialise than a version based on more traditional approaches.

12.3.3 Conclusion

Considering the possible gains stated above, it must be said that a Hicon version of PPM looks very interesting. The examples have suggested that Hiconions are applicable on all levels, from the very vague (cf. fig. 12.19) to the detailed lowest level (cf. fig. 12.22, being both more expressive and more flexible than the traditional ports.

However, the threats pointed out (as well as the cost of changing already existing code) makes it interesting more as an issue for further research than as an idea for short term implementation within the PPM2001 project, which is building a PPM case-tool based on the current language definition.
Chapter 13

Hicons and TEMPORA

13.1 Introduction

For a brief description of the TEMPORA project as a whole, see appendix B. In this chapter we will discuss whether Hicons could contribute anything to the external specification languages of TEMPORA. These languages are

- the binary relationship language ERT for static modelling,
- the external process language (EPL) for dynamic modelling, and
- the external rule language for the explicit modelling of business policies.
  Rules may be both of a static and a dynamic nature.

The latter two are not yet completely agreed upon. ERL will be a syntactic sugaring of the internal temporal Prolog language TERQUAL. EPL will provide two kinds of diagrams: Process Hierarchy Diagrams (PHDs) which are completely trivial process hierarchies which do not show any interaction between different activities, and Process Interaction Diagrams (PIDs). It seems that the latter will be rather similar to PrM (cf. ch. 12). For more on the three languages, see appendix B.

In the following we will first discuss the addition of Hicons to ERT and EPL (which can easily be compared to the Hicon extension of PPM in the previous chapter). Then we will address the treatment of the rule language in connection with this framework. Finally, we will draw some conclusions as to the applicability of Hicons within the conceptual world of TEMPORA.

13.2 Hicons for ERT and EPL

We assume that EPL will be very similar to PrM conceptually (and that PIDs will be rather similar to PrM diagrams). Consequently, there is no point in investigating here the effects of introducing Hicons in EPL — these effects will be more or less the ones stated for introducing Hicons for PrM in the previous chapter.
The effects of adding Hicons to ERT will also be rather similar to the effects of adding Hicons to PhM, as can be illustrated by revisiting the IFIP Conference example. The ERT equivalents of the PhM diagrams of figs. 12.1 and 12.2 are shown in figs. 13.1 and 13.2. For the latter of these, the distinction between identifier and attribute datatypes can be fixed in ERT by defining the possible cardinalities of the mapping between the entity class and value class. However, something corresponding to the quality relation is not supported, so the datatype “Terminated” of fig. 12.1 has simply been omitted. The complex value classes “Sender, date” and “Front page” could (when zooming in) be presented as aggregations of “Sender”, “Date” and “Title”, “Authors”, “Abstract”, “Keywords”, respectively.

Adopting the convention that value classes have dotted lines whereas entity classes have full lines (like in NIAM), the Hiconion versions of these diagrams are presented in figs. 13.3 and 13.4. Notice that the first of these is rather different from the Hiconion version of the corresponding PhM diagram (cf. fig. 12.3). This is because PhM supports n-ary relationships, which means that relationships have to be represented as nodes in the diagram, whereas relationships in ERT can never be more than binary, and thus, they can always be represented by simple edges. For the second, we could be able to reintroduce the datatype “Terminated” in the Hicon approach, without defining the quality relation: Recognizing that this relation actually signals a shift of meta-level, applying to the class as a whole instead of its instances, we could cover it by treating the class itself as an instance of a higher level class by means of the classification construct, as indicated in fig. 13.5.
13.2. HICONS FOR ERT AND EPL

Figure 13.2: The properties of Response in ERT

Just like before, the introduction of Hiconions leads to a significant reduction of the number of edges in the diagrams. To give some numbers, we first have to decide how to count symbols in ERT diagrams. The following questions must be answered:

- Do we count the ordinary binary relationships as one symbol (edge with black square) or three symbols (edge, black square, edge)?

- Do we count subset relationships as one symbol (fork) or as several symbols (circle plus N arrows)?

Toolwise, they are treated as several symbols (i.e. first the square or circle is drawn, then arrows are connected to it). Moreover, for the subset relationship, it is sufficient for syntactic correctness that there is one arrow from the circle and one to it — apart from this arrows to the circle can be added or removed arbitrarily. For the binary relationship, on the other hand, both the edges are required in addition to the square for the construct to have any meaning. Thus, a nice compromise could be to

- count binary relationships as one symbol, but

- subset relationships as many symbols (i.e. each arrow is an edge).

With these conventions, there are 23 edges in the diagram of fig. 13.1 and 16 edges in the diagram of fig. 13.2. The corresponding numbers for the Hiconion diagrams of figs. 13.3 and 13.4 are 6 and 5, which constitute reductions of 73 and 68% ,
Figure 13.3: Hiconion for the overview ERT diagram
Figure 13.4: Hiconion for the values of Response

Figure 13.5: Simulating the quality relation by introducing meta-levels
respectively. The number of nodes and text strings are the same in both diagrams. Thus, ERT compares even worse with Hiconions when it comes to the number of edges than what PhM did (the corresponding reductions for PhM were 47 and 57%). This does not necessarily mean that PhM diagrams are more compact than ERT diagrams — the reason is mainly that ERT supports only binary relationships, which means that we do not need nodes for these in our Hiconion diagrams, so that we get away with using only one edge per relationship instead of two. Still, the numbers definitely suggest that ERT could gain much in switching to an onion notation for subset relations.

Given that the external process language will be something similar to PrM, a Hicon addition is obviously of interest also here, as concluded in chapter 12. All in all, the effects of introducing Hicons in ERT and EPL would be very similar to those of introducing them in PPM:

- increased uniformity, and
- simpler diagrams (especially in reducing the number of edges and providing something better than the PrM ports)

However, there is one aspect that we have not yet treated, namely that of rules. How can we make these fit into our framework? This will be addressed in the following section.

### 13.3 The Hierarchical Treatment of Rules

[102] discusses some ideas for structuring rules hierarchically. There are two main alternatives:

- structure rules in separate hierarchies just like processes or entity classes. Then a model as a whole would end up with three hierarchic structures, as indicated in fig. 13.6(a). The following questions would then have to be answered:
  - Are the standard hierarchical constructs CAGA applicable to rules?
  - Are there other constructs which are more interesting?

- structure rules by attaching them to the hierarchies of the static and dynamic submodels. For instance, any static rule can be attached to some part of the ERT (at some level of abstraction), and any dynamic rule can be attached to some process in the PID (at some level of abstraction). This gives us the structure shown in fig. 13.6(b).

It seems that one should always connect rules to processes or the static structure (since we would usually prefer one model of which three different parts are represented in three different languages, rather than three loosely coupled or not at all
coupled models of the system). Thus, the essential question is whether to try the first approach or not — the second has to be taken anyway.

Thus, we will first discuss the coupling of rules to static and dynamic models. Then we will discuss separate rule hierarchies. To give a definite answer as to whether separate hierarchies of rules are worthwhile is not within the scope of this report, but we will give some examples of hierarchical relations between rules and how these go together with our hierarchical framework.

13.3.1 Coupling Rules to Other Models

Connecting the three sublanguages of TEMPORA, the idea is that static rules (i.e. rules which require or prohibit some state) belong to some class in the ERT, and dynamic rules (i.e. rules which require or prohibit some action) to some process in the PID. A more thorough discussion of this coupling can be found in [53] which suggests a design discipline for the TEMPORA modelling language, and the coupling is also discussed in [68] which evaluates the TEMPORA modelling language (to the extent that it has been defined this far) in a library case study.

To achieve any hierarchical structuring of the rules from coupling them to the other hierarchies, it is obvious that each rule should be connected at the lowest
Figure 13.7: Rules connected to ERT Hiconions

level possible (e.g. a rule which only applies to women should not be connected to “Persons” but to “Women”). In this way TEMPORA will have much in common with ERAE:

- Both languages use temporal logic rules in addition to diagrams (only that ERAE has both statics and dynamics in the same diagram, whereas TEMPORA has one diagram for statics and another one for dynamics).
- If TEMPORA defines a visibility relation similar to that of ERAE, complex objects and processes in TEMPORA will be very similar to ERAE contexts (as described in [37]).

The conceptual side of this connection is not changed in any way by the introduction of Hicons. However, it can be noticed that Hicons lend themselves very easily to a diagrammatic presentation of rules embedded in contexts. Since every node has its definite area in the diagram, rules connected with this node can be shown inside this area (which might of course require zooming to make the area big enough for the rule to be readable). Rules embedded in Hicons have been illustrated in fig. 13.7 and 13.8 for statics and dynamics respectively. Since the external rule language is not fully defined and the exact rule syntax is anyway irrelevant to this illustration, we have simply written the rules here in natural language.

The diagram of fig. 13.7 shows the statics for customers and orders. Some example rules and their location:

- “There should never be more than 100 privileged customers.” This rule could be presented in the position A in the diagram, because it only refers to the class privileged customers.
- “The number of privileged customers should never exceed 20% of the total number of customers.” Due to visibility, this rule has to see not only the privileged customers, but also all other customers. Thus, its position for presentation would be B.
13.3. THE HIERARCHICAL TREATMENT OF RULES

Figure 13.8: Rules connected to EPL Hiconions

- "All order headers are initiated by a serial number" only refer to order headers and can thus be located at C.

- "Every order is defined uniquely by its order header." refers to a relationship between order headers and orders which cannot be seen merely by looking at the set of the former. Thus, this rule has to be placed at D. Finally,

- "To be a privileged customer you have to place at least one order every month" can neither be placed within "Customers" nor within "Orders", and thus it must be located in the outer context (E).

The diagram of fig. 13.8 shows the part of the dynamics for equipment repair. Some example rules and their location:

- "Repairing a bad part should never take more than 5 man-hours." This rule only refers to the Repair Part activity, and thus, it can be located at A.

- "Choosing between repairing a part and replacing it, the cheapest alternative should always be chosen, unless the part has been repaired more than 3 times before, in which case it should be replaced." This rule refers to both the Repair Part and Replace Part activity, and thus, it has to be placed within the diamond which contains both (and which denotes the choice between them), i.e. at B.

- "It is not required that defect finding, defect repair, and testing is done by the same person for one particular bug" refers to all the subparts of the Find One Bug process, and thus, it must be placed within this, i.e. at C. Finally,
"If more than 10 iterations of the Find One Bug process have been performed for one particular piece of equipment, and it is still not working, repair should not continue." refers to the working of the outer process, i.e. the one repeating the Find One Bug activity, and thus, it is located at $D$.

As can be seen, rules that only apply to very specific parts of the (dynamic or static) specification, will be placed deep down in the hierarchy, whereas rules that have a wider application will be located higher up.

### 13.3.2 Rule Hierarchies

It is possible to structure rules in classes, such as "employment rules", "accounting rules", "Dept. A rules", "fire rules", "earthquake rules" etc., and this could also make it possible to form generalization hierarchies of rule classes — for instance, fire rules and earthquake rules could both be specializations of catastrophe rules. Such classification would make a nice index for an otherwise flat rule base, and the modelling of such hierarchies of rule classes could easily be supported by Hicons, since rules would merely function like any other entities in this scheme.

However, from a hierarchical point of view, rules are more interesting when it comes to how they can relate to each other as individuals, the different levels reflecting different levels of detail. In work within the TEMPORA project [102], three main kinds of hierarchies have been encountered:

- **Hierarchies of purpose** (which are often called goal hierarchies in organization theory): These will explain why the organization has a certain rule in terms of a rule on a higher level. For instance, a requirement that employees travel as cheaply as possible on business assignments might be motivated by the higher level requirement that the business must be profitable.

- **Hierarchies of definition**: Many of the rules in an organization depend on or contain definitions of what is meant by various concepts. If the business does some special favours to *good customers*, and there are rules for this, then there should also be some rule defining what a good customer is. This definition might refer to other definitions (for instance, it might be required that good customers are not bad payers, which would again require a definition of what a bad payer is), and thus, a hierarchy of definitions would be formed.

- **Hierarchies of exception**: A common saying goes "No rule without exceptions". If the exception is known in advance, the exception will of course be a rule in itself. However, it is inherent in the nature of exceptions that there is also a need for handling unexpected cases, but for the unexpected it is likely that no rules will be given. Thus, hierarchies can be given only for known exceptions, whereas the unknown must be handled by providing the sufficient flexibility in the run-time system. Two rules can be related hierarchically if one *overrides* the other — for instance, there might be a general rule that to be a good customer you must be a good payer. Then, there might be exceptions, for instance: If a customer is a member of the
Royal Family, then he shall be considered a good customer even if he is a bad payer. An interesting point about this kind of hierarchies, is that whereas hierarchies of purpose or definition have the most general rules on top, hierarchies of exception tend to have the most detailed and special cases on top (since these are usually the ones to overrule all other rules).

These hierarchical relations are what we have called instance level relations, and as stated in chapter 9, we do not provide any predefined instance level relations within our framework. However, they can easily be defined in the meta-level language (just like any other concept we might need). Hence, we could also define relations like “boss of”, “father of” etc. In the following we will give some small examples illustrating rule hierarchies. Since the onion notation is not very intuitive for such hierarchies anyway, we do not bother to invent any new icons. Instead we simply use the tree notation.

**Refers-to** In the examples rules are given in a more or less natural language rather than in the formal rule language of TEMPORA because the formal syntax is not interesting here — we only want to show how rules relate hierarchically. Assume that we have got the following rules:

1. a customer is good IF he is not a bad payer AND he is not a cheater AND he is not excessively complaining AND he has had a long term relationship with us

2. a customer is a bad payer if he has been more than a month late with at least two payments the last three years

3. a customer is a cheater IF ...

4. a customer is excessively complaining IF he has made at least five complaints the last year AND at least three complaints have been sour

5. a complaint is sour IF what the customer complained about was his own fault

6. a complaint is sour IF dealing with it demanded more than 1 hour effective secretarial time AND the customer’s economical loss due to the error complained about is estimated to be smaller than 100 dollars.

7. a customer has had a long term relationship with us IF he has been registered as a customer for more than five years AND his annual purchase has never dropped below 100,000 dollars.

8. it is obligatory to grant a credit to an applying customer IF that customer is good

9. it is prohibited to send Xmas gifts to a customer who is an excessive complainer
Figure 13.9: The \textit{refers-to} relation

These rules have been structured in a hierarchy in fig. 13.9a, where the rules are identified by their numbers in the list above. Of course it is not intended that numbers should be used in the actual diagrammatic presentation in an actual tool — here it one would have to present either the whole rule, or a meaningful abbreviation for it.

In the figure the relation does not result in a strict hierarchy — this is supposed to be a rather normal situation, since often several rules will refer to the same definition at the lower level. However, cycles \textit{should} not occur.

\textbf{Motivates} \quad Here we have called the relation for forming hierarchies of purpose \textit{motivates}. With the rules

1. it is required that the organization survives
2. it is required that the income exceeds the expenses
3. it is objectionable to sell a car if the price obtained is less than 20\% higher than the purchase price
4. it is required to sell a car IF the estimated costs of keeping it exceed a possible loss suffered from selling
5. it is prohibited to purchase trucks weighing more than 3 tons

a small goal hierarchy is shown in fig. 13.10. The motivation for the last rule could be that the storage facilities cannot accommodate larger vehicles, so that this would require costly extensions of the organization's buildings.

Just like the case for definitions, goal hierarchies need not be strictly hierarchical — one rule could be motivated by several higher level rules. Again, cycles should certainly not occur.

\textbf{Overrules} \quad It is beyond the scope of this work to discuss exception handling as such. Here we will only give a simple example on how rules could relate hierarchically as a result of some rules being exceptions to others.
1. it is required to reduce the price of a product IF a competitor launches a similar product AND that product is cheaper than ours

2. it is required to increase the price of a product IF the demand for this product shows a dramatic increase AND the production cannot be increased accordingly

3. it is prohibited to change prices during opening hours

4. it is permitted to change prices during opening hours IF the daily inflation rate exceeds 5%

Assuming that 3 overrules 1, 2 and 4 overrules 3, we can put up the hierarchy of fig. 13.11.

Just like the two previous kinds of hierarchies, an overrule hierarchy need not be strictly hierarchical, but cycles should not occur. A more formal analysis of how a hierarchical ordering of rules can serve to overcome imperfections in the code itself is given in [2], which among other things discusses the possibility for overruling.

Of course it might also be possible to show all kinds of inter-rule relationships in one picture, but this would easily become messy. Obviously, the resulting picture would not be a neat hierarchy, but a general network of rules. As always when there is much information to be shown, filtering is important — one cannot expect to show or see everything at once.
13.4 Conclusion

A Hicon extension of the TEMPORA external language is about as interesting as a similar extension of PPM, in that the gains are the same. A more uniform language is achieved, and there is a potential for simpler diagrams, especially in that the number of edges may be significantly reduced. Moreover, it seems that the diagrammatic presentation of rules within processes and entity classes is enhanced by the use of Hiconions, since the onion notation is also used by ERAE contexts (which is a facility for structuring rules by connecting them to static and dynamic declarations).

Hierarchical relations between individual rules have not been predefined, but any interesting relation can easily be captured by the meta-level language. Anyway, flexibility is important here, since the question of rule hierarchies has not been thoroughly researched.

However, just like for PPM there are threats. Switching to Hicons would require much extra work, and the time-limits are narrow, it will be impossible to experiment with Hicons and the onion notation within the TEMPORA project. Thus, a Hicon extension of the TEMPORA external language is rather an option for those who might elaborate upon ideas of the TEMPORA language some time in the post-project future.
Part V

Visions and Achievements
Part Introduction

This part concludes the thesis. Our visions for the future are presented, and we sum up what has been achieved and what has not been achieved.

Chapter 14 indicates how we envision the method and tool support for a general framework like Hicons.

Chapter 15 concludes the thesis, stating what has been achieved, and suggests some courses which could be taken to develop the Hicon framework further in the future.
Chapter 14

Method and Tool Issues

14.1 Methodological Flexibility

We believe that Hicons can contribute much to the methodology of IS engineering. This thesis has been one about language, not methods, but the ultimate goal is methodological rather than linguistic. The key word, again, is flexibility. Except for emphasising hierarchical abstraction mechanisms, which we have argued is a necessity in the design of complex systems, our framework does not make any methodological presumptions. This can be utilized in several ways:

- Hicons can be connected to already existing languages. In this case, the methods already provided for these languages could be used, possibly with some modifications if Hicons extend the expressive power of the language. For instance, a method for systems development by means of the traditional PPM would certainly be applicable also to a Hicon extension of PPM without any fundamental changes.

- Hicons being sort of method independent, a Hicon based CASE-shell could be an excellent facility for experimenting with different kinds of methods in different problem-domains. The uniformity promoted by the use of Hicons makes it easier to compare the different cases, with the ultimate goal of identifying which steps are common to several areas and which are more specific.

For instance, it follows from the flexibility of Hicons that they can be used both for working top-down, bottom-up or in some combination of these two.

14.2 The Hicon CASE-Shell

As mentioned earlier, the ideal tool support for the Hicon framework is that of a flexible CASE-shell. The basis of this CASE-shell would be the interpreter of a meta-level language. In this meta-level language, one should be able to define both
• the modelling constructs to be used (syntax and semantics, as well as the external representation),

• methods to be used (such as the steps to be taken, routines for checking consistency and completeness), and

• the dialogue between the tool and its user (warnings, error messages, guidelines, questions etc.).

Obviously, this CASE-shell is in itself a very ambitious idea (which explains why we have not made any attempt at designing it in this thesis), and maybe not the most appealing to short term commercial interests. However, we believe that the Hicon CASE-shell can be obtained gradually, in a way which will give interesting results all the time, as will be explained in the next section.

### 14.3 The Future: One Path, Not Two

As stated earlier, we see two basic uses for the Hicon framework:

• extending other languages to strengthen their support for hierarchical abstraction mechanisms and/or increase their uniformity of notation.

• as the basis of a CASE-shell with a pan-presumptionistic idea.

The first one is the most easily achievable and the most directly commercially applicable, the second the most interesting (at least from our point of view). However, we do not see the two above uses of the Hicon framework as two different branches winding into the future — rather our vision is the following:

• Since the Hicon CASE-shell (which is the ultimate goal) is such a huge effort with no direct pay-off, it cannot be realised in one jump. Therefore, the following sequence of action seems sensible:

  1. Make Hicon extensions of some existing languages and experiment with these.

  2. Provide an integrated tool support for these languages.

  3. Gradually extend the flexibility of this CASE-shell so that the user is less and less restricted to specific languages and methods, being able also to define her own constructs if this is found necessary.

Iterations of the third step would then give us the wanted tool. In this way, the ultimate scientific goal of Hicons can be realized gradually from by-products which could even be commercially useful.

Thus, we envision the Hicon CASE-shell as a very realistic idea.
Chapter 15

Conclusions

15.1 Claimed Contributions

This thesis has contributed to

- a general understanding of hierarchies as such and the importance of hierarchical abstraction mechanisms in IS engineering languages, as well as an identification of the major abstraction mechanisms provided in current modelling languages within the field.

- an identification of the main techniques for diagrammatic presentation of hierarchies within current approaches for conceptual modelling. In particular, the advantages of the onion style have been illuminted.

- the promotion of a uniform approach to the conceptual and diagrammatic treatment of hierarchies by the definition of the Hicon framework, whose generic character makes it a candidate for

  - extending (or modifying) current modelling approaches and/or
  - forming the basis for a powerful case-shell for hierarchical modelling.

15.2 Main Conclusions

The main conclusions drawn from this work are the following:

- The support for abstraction mechanisms in current modelling approaches is varying. Some languages provide almost nothing, whereas others are more advanced. The hierarchical relations of classification, aggregation, generalization, association, and vague composition encountered in the analysis of current approaches were found to be very general, in that they are applicable both to static and dynamic modelling.
• There are two major styles for the diagrammatic representation of hierarchies, the \textit{onion style} and the \textit{tree style}. Both have their pros and cons, but our discussion suggests that the onion style is the most interesting, at least for the general hierarchical constructs mentioned above.

• The Hicon framework seems to have a very general applicability. We have investigated the possibilities for extending the modelling languages of PPM and TEMPORA with Hicons, and these look very promising, particularly if the onion notation is used. We have also briefly indicated the use of Hicons with several other languages, and for many of these the abstractive power and/or external representation of hierarchies would certainly be improved by such a modification.

• There exists a framework similar to Hicons, namely \textit{Higraphs}. From the comparison made between the two, it can be concluded that:

  – the existence of the Hicon framework is justified by the fact that it is more expressive than Higraphs (which only supports aggregation and generalization).

  – On the other hand, some examples suggest that the notation of Higraphs is more compact, so that Higraphs might be the best framework for problems where aggregation and generalization are the only constructs necessary (as seems to be the case for state-transition diagrams). Thus, Hicons do not make the Higraph framework superfluous either.

  – Hiconions are more radical than Higraphs in taking the onion notation to its ultimate consequence. This work suggests the use of \textit{shapes} in distinguishing between the different hierarchical constructs, which seems to emphasize these aspects more strongly than other notations surveyed. The effect of this is that other means than shape (e.g. colour) has to be used for distinguishing between different semantic categories of a language, and this makes Hiconion extensions rather different from the current notations of most modelling languages. This might be a disadvantage, at least in a short term.

15.3 Directions for Further Work

* The main directions for putting the Hicon framework to practical use were lined out already in the previous chapter:

• attaching Hicons to existing languages. For instance, it could be interesting to elaborate on

  – the ideas presented for using Hicons for describing PrM processes, and
  – the pros and cons of using Hiconions to achieve larger uniformity in the PPM language.

• develop the Hicon case-shell. This will amount to
15.3. DIRECTIONS FOR FURTHER WORK

- making a definite choice for a meta-level language,
- developing an interpreter for this language,
- predefining the essential constructs, and
- developing the necessary drawing facilities.

In addition to, or rather in correspondence with this work it would be interesting to assess the quality of the Hicon framework and its suggested external representation through case studies:

- Psychological studies to
  - verify or falsify the claimed advantages of the onion notation when it comes to perceptibility, or more specifically, to investigate the claimed advantages of the Hiconion extensions of PPM or TEMPORA as compared to the current notations.
  - investigate whether the choice of using shapes for distinguishing between the hierarchical constructs is good, specifically if it is satisfactory to use colour for distinctions which are often made by means of icon shape in other languages,
  - investigate whether the suggested choice of icons is good, or whether it can be improved.

- Modelling large examples by means of Hiconion based languages. In this work we have only used small examples. The quality of the language can more easily be established if larger modelling efforts are undertaken. However, it is difficult to model large examples without tool support.

- Practical use in realistic IS development projects.

The last point is one which cannot be achieved quickly. However, we think that in due time, Hicons will reach a maturity sufficient for its practical commercial use. The discussions earlier in this report have definitely indicated that the Hicon framework can contribute very positively indeed to the discipline of IS engineering, with its combination of generality and a focus on the very important aspect of hierarchical abstraction mechanisms. So if anyone were to ask teasingly what our invention can be used for, we can answer just like Edison used to do: "What good is a baby?", with the implicit attitude that a baby can be very good indeed.
Part VI

Appendices
Appendix A

A Description of $\theta$

The description of $\theta$ provided here is based on [11], chapter 7. First we will explain the use of relations in $\theta$. Then we will discuss the $\theta$ interpreter. Finally, we will indicate how $\theta$ can be extended with a basis of predefined constructs.

A.1 The Use of Relations in $\theta$

The language $\theta$ is very clean in that the only concept provided is the relation. To enable relations to refer to each other, each of them is given a unique label (this labelling is also a relation, although a notational shorthand has been given). A relation has three aspects:

- the description aspect,
- the application aspect, and
- the reference aspect.

To distinguish between the three different aspects, each relation name is subscribed by either D, A, or R. As an example

\[
<_{A}^{#1} \text{likes}_{A} \text{john}_{R} \text{mary}_{R} >_{A}
\]

could be the $\theta$ expression for “John likes Mary”. Here the brackets are the meta-construct which collects relations into a composite relation (i.e. they denote the relation relation). Of the three relations collected, “likes” is applied, whereas “john” and “mary” are referred to (i.e. in a predicate logic terminology this would mean that “likes” is the predicate and the other two its terms. The difference between reference and application can be exemplified further by the statement

\[
<_{A}^{#2} \text{knows}_{A} \text{john}_{R} \text{knows}_{R} >_{A}
\]
meaning that “John knows the relation knows” — a typical meta-knowledge difference.

The label # 1 illustrates the naming relation, which provides every composite relation with a unique identifier, so that it can be referred to by other relations, like

\[
<^4_A \text{knows}_A \text{jack}_R \#1 >_A
\]

which says that “Jack knows that John likes Mary”.

Every relation must be defined to be applied or referred to. A definition of a relation will apply the predefined relations denotation (which states the name(s) of the relation to be defined), intension (which defines the semantics of the relation in terms of other already defined relations, if this is possible), and extension (which describes the relation in terms of its typical instances, if any instances are given). As an example we give a description of the relation “john”:

\[
<^4_A \text{john}_D \\
<^5_A \text{instance-of}_A \text{john}_R \text{person}_R >_A \\
<^6_A \text{has-property}_A \text{john}_R \text{tall}_R >_A \\
>_A
\]

which also illustrates the use of nested composite relations.

### A.2 The θ Interpreter

Even in a meta-level language like θ there has to be some primitives, i.e. some knowledge which is not defined in terms of other knowledge, and which cannot be explicitly represented in θ itself. Such knowledge will have to be encoded in the θ interpreter (which interprets statements made in θ). Basically, the interpreter will have to know the following:

- the meaning of the relation concept.
- the meaning of the three different aspects of a relation.
- the description of primitive relations.
- the unique numbering scheme.
- the deep meaning of each relation represented.

The last point, of course, can only be dealt with by human interpretation, since we can never expect to automate the deep meaning of any concept. However, the first four points could be implemented in a running program.
A.3 Defining Useful Relations in $\theta$

As indicated in [11] it will be rather tedious to define new concepts in $\theta$ if only the basic constructs discussed in the previous section are provided. Thus, it might be comfortable to define some more relations as primitive in the interpreter than what is strictly needed. The creators of $\theta$ call such a version of $\theta$ the natural expansion of $\theta$, $\theta_{NE}$. In this version they provide the following primitive relations: denotation, intension, extension, syntax, set-of, member-of, has-property, and the logical connectives. Other relations, such as instance-of, specialization-of, and subset-of can be defined in terms of these. In the example below, we define instance-of (which would be used for classification in the Hicon framework):

\[
<_{A^{14}} \text{intensionally-equal}_{A} \\
<_{A^{15}} \text{instance-of}_{R} \text{rel}_{1R} \text{rel}_{2R} > \\
<_{A^{16}} \land_{A} \\
<_{A^{17}} \text{member-of}_{A} \text{rel}_{1A} \text{rel}_{3A} > \\
<_{A^{18}} \text{extension}_{A} \text{rel}_{2A} \text{rel}_{3A} > \\
<
\]

$\theta$ provides two ways of defining new concepts:

- classical definition (i.e. by means of intensional equality with some expression in terms of concepts already defined). The definition of the instance-of relation above is an example of this.

- family resemblances (i.e. by means of schemata and prototypes). In some cases, a classical definition is impossible, because there are no already defined concepts which can describe the semantics of the one we want to define. Prototypes can be listed in the extension part of a relation description.

For the sake of illustration we give classical definitions for the concepts node, aggregation and generalization in concepts could be defined in $\theta$. Since the bracket constructs are always implied, we omit the $A$ subscripts from these in the below examples. Moreover, since we do not define any relations which need to refer to other composite relations, we omit the labels. These omissions are only done for the matter of convenience and should not be regarded as an alternative way of writing $\theta$ statements in general.

The node relation can be defined as follows:

\[
< \text{node}_{D} \\
< \text{denotation}_{A} \text{node}_{R} > \\
< \text{intension}_{A} \text{node}_{R} \\
< \text{instance-of}_{A} \text{node}_{R} \text{relation}_{R} > \\
< \forall_{A} \\
< \text{has-property}_{A} \text{node}_{R} \text{composable}_{R} >
\]
\(<\text{has-property}_A \text{node}_R \text{decomposable}_R >\>
>
>

where we assume that the relations \emph{composable} and \emph{decomposable} are already defined, stating that a node must at least either be composable or decomposable (most nodes will be both).

The \textbf{aggregation} relation can be defined as follows:

\(<\text{agg}_D
<\text{denotation}_A \text{agg}_R >
<\text{intension}_A \text{agg}_R
<\text{syntax}_A \text{agg}_R \text{node}_R <\text{set-of}_A \text{node}_R >
<\text{instance-of}_A \text{agg}_R \text{relation}_R >
<\text{implies}_A
<\text{agg}_A \text{m}_R \text{d}_R >
<\wedge
<\text{subset-of}_A \text{m}_R \text{n}_R >
<\text{cart-prod-of}_A \text{n}_R \text{d}_R >
>
>
>
>
>

where we assume that the relation \textit{cart-prod-of} which says that the first argument is the Cartesian product of the members of the second argument (which should thus be a set) is already defined.

The \textbf{generalization} relation can be defined as follows:

\(<\text{agg}_D
<\text{denotation}_A \text{agg}_R >
<\text{intension}_A \text{agg}_R
<\text{syntax}_A \text{agg}_R \text{node}_R \text{daughter-nodes}_R >
<\text{instance-of}_A \text{agg}_R \text{relation}_R >
<\text{implies}_R
<\text{gen}_A \text{m}_R \text{d}_R >
<\wedge_A
<\text{intensionally-equal}_A \text{m}_R \text{n}_R >
<\text{union}_A \text{n}_R \text{d}_R >
<
>
>
>
where we assume that the relation \textit{union} which says that the first argument is the union of the members of the second (which should be a set) is already defined.

If we should use \( \emptyset \) as a basis language for the Hicon framework, it might be rather inefficient to rely on definitions such as these (which would then have to be interpreted each time an aggregation or generalization construct was to be analyzed. Indeed, it could be wiser to make an expansion of \( \emptyset \) where all the hierarchical constructs (as well as some other constructs that might be frequently needed) were predefined in the interpreter. We could call such an expansion \( \emptyset_{HE} \) (Hicon expansion).

Finally, the user will be able to define required constructs in addition to the framework provided by us. The example below shows the definition of a data flow (taken from [11]):

\[
<^{1}\text{data-flow}_{D} \\
<^{2}\text{denotation}_{A} \text{data-flow}_{R} > \\
<^{3}\text{intension}_{A} \text{data-flow}_{R} > \\
<^{4}\text{syntax}_{A} \text{data-flow}_{R} \text{source}_{R} \text{sink}_{R} > \\
<^{5}\text{instance-of}_{A} \text{data-flow}_{R} \text{relation}_{R} > \\
<^{6}\text{implies}_{A} > \\
<^{7}\wedge_{A} > \\
<^{8}\text{instance-of}_{A} f_{1D} \text{data-flow}_{R} > \\
<^{9}\text{instance-of}_{A} f_{2D} \text{data-flow}_{R} > \\
<^{10}\text{equal}_{A} > \\
<^{11}\text{denotation}_{A} f_{1A} > \\
<^{12}\text{denotation}_{A} f_{2A} > \\
> \\
<^{13}\text{intensionally-equal}_{A} f_{1A} f_{2A} > \\
> \\
<^{14}\text{union}_{A} \text{terminals}_{D} #25#34#41 > \\
<^{15}\text{member-of}_{A} \text{source}_{D} \text{terminals}_{A} > \\
<^{16}\text{member-of}_{A} \text{sink}_{D} \text{terminals}_{A} > \\
> \\
<^{17}\text{extension}_{A} \text{data-flow}_{R} > \\
<^{18}\text{set-of}_{A} \text{application}_{R} \text{rejection}_{R} \text{acceptance}_{R} \text{employee-data}_{R} > \\
> 
\]

This description of the data-flow relation contains both general syntax and semantics (the \textit{intension} part) and particular data flows (the \textit{extension} part). The labels \# 25, \# 34, and \# 41 refer to relations in the definitions of processes, data stores, and external agents which we have omitted here for the sake of brevity.
A.4 How We Would Use $\theta$

$\theta$ can be used both for representing schemata and meta-schemata (as suggested by the data flow definition above). However, it must be emphasized that it is intended to be only an internal representation structure — the user of a case-shell should not be required to write statements in $\theta$ when she wants to define a new concept. For this purpose, more user-friendly facilities should be provided (so that the user can define his concepts in an easier way, which can then be represented internally as $\theta$ statements). Also, it must be noticed that the most frequently used constructs would probably have to be predefined in the interpreter, rather than defined in terms of other primitives, if efficient tool support is to be achieved.
Appendix B

The TEMPORA Project

In this appendix we describe the TEMPORA project. In section 1, its scientific background and goals are summarized, and the different partners are listed. In section 2 we describe the external modelling languages provided by TEMPORA.

B.1 Introduction to the TEMPORA Project

B.1.1 Partners and Background

The ESPRIT2 project TEMPORA is to a large extent based on the results of the ESPRIT1 project RUBRIC, which was finished in 1989. Two of the partners of RUBRIC are central in the TEMPORA project, namely BIM (Belgium) and UMIST (UK). The other partners of TEMPORA are Univ. of Liege (Belgium), Sybase Ltd. (UK), Imperial College (UK), Hitec (Greece), LPA (UK), SISU (Sweden), and SINTEF (Norway). As with any ESPRIT project this is a mixture of industry and research institutions.

The RUBRIC project was inspired by the fact that in spite of an increased productivity in software development through the introduction of CASE-tools, there are still major problems that have not been dealt with. Most importantly, information systems often do not satisfy the requirements of the customers, either because one was not able to formalise them properly, or because they were misunderstood. To deal with this, RUBRIC advocates a rule-based paradigm. Traditionally, the rules of the business organization will be implicitly represented in programming code within the information system. This makes the system difficult to maintain if the rules were to change (as they often do). Consequently, RUBRIC goes for an explicit representation of the rules, and for this a formal language has been developed.
B.1.2 The Objectives of TEMPORA

The results of the RUBRIC project are still only on a research level, and the following shortcomings can be identified:

- a tool support for the RUBRIC paradigm has to be developed
- the power of expression in the RUBRIC rule language is still limited, particularly when it comes to the temporal aspects

The objectives of the TEMPORA project is to deal with both these two. The idea is to generate Prolog code from the specifications. BIM-Prolog (a relatively fast Prolog developed by BIM) has been coupled to SYBASE (a relational database system developed by Sybase Ltd.) The hope is that this might make systems based on Prolog code commercially acceptable. At the same time, several graphical modelling tools are to be developed.

To deal with the lack of expressive power, the RUBRIC language has been extended with new features. Temporal logic reasoning will be done by transforming the external specifications to the temporal Prolog language TERQUAL. Three external modelling languages are to be provided:

- ERT (Entity-Relationship Time), a binary relationship language for static modelling, also providing facilities for time-stamping (to restrict the time-granularity of information of various classes).
- EPL (External Process Language) — although not yet fully established, it seems that this will be something similar to PrM (cf. chapter 12).
- ERL (External Rule Language), also not yet fully defined — a user-friendly layer on top of TERQUAL.

All in all, the languages involved in specifying some application with the TEMPORA paradigm can roughly be sketched as shown in fig. B.1. Models in the three external languages are transformed to a DB-schema and a set of rules in TERQUAL. In TERQUAL automatic reasoning (such as consistency checking) can be performed. Finally, TERQUAL rules can be transformed to BIM-Prolog.

B.2 The External Languages

B.2.1 ERT

ERT is a binary relationship language (just like NIAM, cf. ch. 7). The conceptual basis of the language can be described as follows:

- There are three kinds of classes:
Figure B.1: The TEMPORA application modelling languages

- entity classes,
- value classes,
- relationship classes.

- Aggregation is provided for entity classes and value classes by means of a complex object construct.

- Generalization is supported through two constructs, one for total subclass partition and one for non-total subclass partition.

- There are constructs for stating the cardinalities of relationships.

- There is a construct for distinguishing between stored and derived information.

- There is a construct for time-stamping entity and relationship classes in order to restrict their temporal granularity (i.e. how often tuples of the class can change).

The symbols of ERT are illustrated in fig. B.2. Examples of ERT diagrams can be found in chapter 7, 8, and 13.

B.2.2 External Process Language

The external process language of TEMPORA provides two kinds of diagrams: Process Hierarchy Diagrams (PHD) and Process Interaction Diagrams (PID). The former only present a trivial hierarchy of processes, as indicated in the example of fig. B.3, i.e. nothing is said about how activities interact. This is supposed to be shown in the PID's. The PID part of the language has not been fully agreed on yet, but we believe that the PID language will be rather similar to PrM, which has been described in chapter 12.
Figure B.2: ERT symbols

Figure B.3: A Process Hierarchy Diagram
B.2.3 The External Rule Language

The external rule language (ERL) should be more easily understandable to human beings and have a better expressive economy than what is offered by the temporal Prolog language TERQUAL. The external rule language might be textual or a combination of text and diagrams. Ideas for a semi-natural rule language are elaborated in [59]. The draft version of ERL provided by TEMPORA [72] goes for a more direct syntactic sugaring of TERQUAL. This version is summarized below.

All ERL rules have the same basic structure, namely

\[
\text{ERL. rule::=}[\text{WHEN } \langle \text{trigger. exp} \rangle][\text{IF } \langle \text{cond. exp} \rangle]\text{THEN}<\text{exp}>
\]

where the expressions within the square brackets are optional. The difference between IF and WHEN is that the former signals conditions which are true, whereas the latter signals conditions becoming true — in terms of temporal logic WHEN T is a shorthand notation for IF \(\sim T \wedge \sigma T\), i.e. T holds in the current state but did not hold in the previous state. The above scheme gives us four types of rules:

- \(\langle \text{exp} \rangle\)
  \(\text{exp}\) must always hold.

- IF \(\langle \text{cond. exp} \rangle\) THEN \(\langle \text{exp} \rangle\)
  \(\text{exp}\) must hold whenever \(\text{cond. exp}\) holds.

- WHEN \(\langle \text{trigger. exp} \rangle\) THEN \(\langle \text{exp} \rangle\)
  \(\text{exp}\) must hold when \(\text{trigger. exp}\) has just begun to hold.

- WHEN \(\langle \text{trigger. exp} \rangle\) IF \(\langle \text{cond. exp} \rangle\) THEN \(\langle \text{exp} \rangle\)
  \(\text{exp}\) must hold if \(\text{cond. exp}\) holds and \(\text{trigger. exp}\) has just begun to hold.

Whereas the executable rules of a system will refer to the db-schema, ERL rules naturally refer to the corresponding ERT model. This is done by the following general structure (where the parts within the bold brackets are optional):

\[
\text{ERL. data. access::=}<\text{entity/value name}>[(<\text{variable}>)]
   \quad[<\text{relationship}>\text{ERL. data. access}>,<\text{relationship}>\text{ERL. data. access}>]
\]

Naming an entity or value causes the access expression to hold for each instance of the class, and if a variable is provided, this can be bound to all instances found. Including information about relationships with other entities or values, we can restrict our selection of instances.

In addition to the above, ERL provides some set operator expressions to extend the user-friendliness of temporal predicate logic. To conclude the presentation of ERL we show a rather complex rule in natural language and ERL:

- If it is seven days after an instalment was due on a clearly arranged, and the sum of all payments made during the period of the arrangement is within £25 of the instalment amount, close the arrangement the normal way.
• IF 7*DAYS AGO
   (instalment{value amount(install),belongs_to arrangement(a)})
   AND clearly(a)
   AND install-25<SUM{pay:IN. PAST amount(pay)}of account, movement
   [type payment, charged_to_account{governed_by arrangement(a)}]

   THEN close_arrangement(a)

Although some improvements have been made as compared to ordinary predicate logic, this language is still rather compact and cannot be expected to be very readable to non-logicians. Hence example rules in chapter 13 have rather been stated in natural language, since anyway, the exact language chosen for writing the rules is not essential to the discussion of rule hierarchies and the connection of rules to the ERT and EPL.
Bibliography


[22] M. Bunge: The Metaphysics, Epistemology, and Methodology of Levels, in [119]


BIBLIOGRAPHY


[99] H. A. Simon: *The Organization of Complex Systems*, in [86].


