The Open University

Open Research Online

The Open University's repository of research publications
and other research outputs

Reusable components for knowledge modelling

Thesis

How to cite:

Motta, Enrico (1998). Reusable components for knowledge modelling. PhD thesis The Open University.

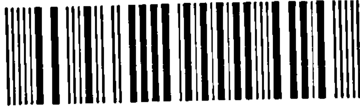
For guidance on citations see FAQs.

© 1998 The Author

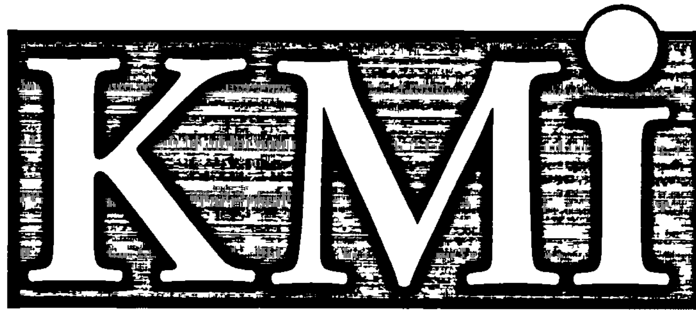
Version: Version of Record

Copyright and Moral Rights for the articles on this site are retained by the individual authors and/or other copyright owners. For more information on Open Research Online's data policy on reuse of materials please consult the policies page.

oro.open.ac.uk



UNRESTRICTED



Knowledge Media Institute

Reusable Components for Knowledge Modelling

Enrico Motta

KMI-TR-54

November 26th, 1997

Thesis submitted in partial fulfillment of the requirements for a PhD in Artificial Intelligence

Author no. M7045185
Date of submission 28th November 1997

The Open University

Abstract

In this work I illustrate an approach to the development of a library of problem solving components for knowledge modelling. This approach is based on an epistemological modelling framework, the Task/Method/Domain/Application (TMDA) model, and on a principled methodology, which provide an integrated view of both library construction and application development by reuse.

The starting point of the proposed approach is given by a *task ontology*. This formalizes a conceptual viewpoint over a class of problems, thus providing a task-specific framework, which can be used to drive the construction of a *task model* through a process of model-based knowledge acquisition. The definitions in the task ontology provide the initial elements of a task-specific library of problem solving components.

In order to move from problem specification to problem solving, a generic, i.e. task-independent, model of *problem solving as search* is introduced, and instantiated in terms of the concepts in the relevant task ontology, say T. The result is a task-specific, but method-independent, *problem solving model*. This generic problem solving model provides the foundation from which alternative problem solving methods for a class of tasks can be defined. Specifically, the generic problem solving model provides i) a highly generic *method ontology*, say M; ii) a set of generic building blocks (*generic tasks*), which can be used to construct task-specific problem solving methods; and iii) an initial problem solving method, which can be characterized as the most generic problem solving method, which subscribes to M and is applicable to T. More specific problem solving methods can then be (re-)constructed from the generic problem solving model through a process of *method/ontology specialization* and *method-to-task application*.

The resulting library of reusable components enjoys a clear theoretical basis and provides robust support for reuse. In the thesis I illustrate the approach in the area of parametric design.

Acknowledgements

In general, it is not much fun to do things on your own. This is even more true in the case of knowledge work: on the one hand, it is only possible to talk about knowledge, as long as this is communicated and shared; on the other hand, knowledge work is normally a collaborative process. As it turns out, I have been very lucky in being a member of a very stimulating work environment, first in HCRL and now in KMI. For all this I have to thank Marc Eisenstadt, from whom, over the years, I have learnt most of what I know about technology and R&D, and who has always been a source of stimulus and inspiration. Having joined Marc's group straight after university, I have held only one job in life (actually zero, if one subscribes to my parents' view); therefore I don't have direct work experience in any other academic (or non-academic) environment. Nevertheless, I doubt that it will be possible to find anywhere else the same degree of excitement and intellectual freedom which characterizes KMI (with the obvious exceptions, of course, of the research groups run by the external examiners).

Most of the contents of this thesis (especially the parametric design technology) was the result of a four-year collaboration with Zdenek Zdrahal, whose scientific rigour and encyclopaedic knowledge of the Godfather trilogy have always been a constant source of inspiration and reassurance, especially for those tricky moments in which one cannot remember the name of the Sicilian wife of Michael Corleone, or is unsure about the relation between the min-conflict heuristic and Propose&Revise. In addition, I would also like to thank Zdenek for reading and commenting on large chunks of the thesis.

A number of other people have provided comments on various parts of the thesis, including Dieter Fensel, Arthur Stutt, and John Domingue.

I would also like to thank the friends and colleagues with whom over the years I have collaborated on topics related to knowledge modelling and/or design. These include Dieter Fensel, Arthur Stutt, John Domingue, Nigel Shadbolt, Kieron O'Hara, Mauro Gaspari, Stuart Watt, Stefan Decker, Frank van Harmelen, Richard Benjamins, Bob Wielinga, Rudi Studer and Annie Brooking. I also benefited from discussions with the participants in the Sisyphus initiatives, especially Bill Birmingham, Karsten Poeck, Guus Schreiber, Jay Runkel, and Gregg Yost. I'd also like to thank Simon Buckingham-Shum, Marco Ramoni, Sabina Falasconi, Mario Stefanelli and Silvana Quaglini for useful discussions on issues related to knowledge modelling. Special thanks to Tamara Sumner for being a constant source of useful material on design and for helping with the tennis development programme.

Finally, I would like to thank Stefania, for being patient while I took so long to complete this work, and Emanuela, just for being there.

Table of Contents

1. Knowledge, Models and Reuse 1

1.1. Introduction 1

1.1.1. The nature of knowledge modelling 1

1.1.2. Context of the work 3

1.1.2.1. Contributions to research in design problem solving 3

1.1.2.2. Contributions to research in knowledge modelling, sharing and reuse. 4

1.1.2.3. Knowledge, models, and reuse 6

1.2. A characterization of knowledge-based systems 6

1.2.1. The ambiguous notion of knowledge-based system 6

1.2.2. The existential predicament of knowledge-based systems 7

1.2.2.1. Knowledge as representation 8

1.2.2.2. Knowledge-based systems as agents 8

1.2.2.3. The experiential nature of problem-solving knowledge 11

1.2.3. The role of knowledge-based systems 11

1.3. Evolving perspectives on knowledge acquisition 12

1.3.1. Knowledge acquisition as mining 13

1.3.2. Cognitive and technical problems with the mining view 14

1.3.3. Multiple levels of description 15

1.3.4. Knowledge acquisition as modelling 17

1.3.5. A minor caveat 19

1.4. Reuse in knowledge based systems 20

1.4.1. Economic and scientific motivations for reusable knowledge bases 20

1.4.2. Reuse and knowledge modelling 21

1.5. From battleground to background 22

2. Approaches to Knowledge Modelling 23

2.1. Introduction 23

2.2. An overview of knowledge modelling frameworks 23
2.2.1. KADS/Common KADS 24
2.2.1.1. Types of components 25
2.2.1.2. Relations between components 26
2.2.1.3. Model building process 26
2.2.1.4. Evaluation of the approach 27
2.2.2. Components of Expertise 28
2.2.2.1. Types of components 29
2.2.2.2. Relations between components 29
2.2.2.3. Model building process 29
2.2.2.4. Evaluation of the approach 29
2.2.2.5. Conclusions 30
2.2.3. Generic Tasks 30
2.2.3.1. Types of components 30
2.2.3.2. Relations between components 31
2.2.3.3. Model building process 31
2.2.3.4. Evaluation of the approach 31
2.2.4. Role-limiting Methods 31
2.2.4.1. Types of components 32
2.2.4.2. Relations between components 32
2.2.4.3. Model building process 32
2.2.4.4. Evaluation of the approach 32
2.2.5. Protégé 32
2.2.5.1. Types of components 33
2.2.5.2. Relations between components 33
2.2.5.3. Model building process 33
2.2.5.4. Evaluation of the approach 33
2.2.6. DIDS 34
2.2.6.1. Types of components 34

2.2.6.2. Relations between components 34

2.2.6.3. Model building process 35

2.2.6.4. Evaluation of the approach 35

2.2.7. Spark/Burn/Firefighter 35

2.2.7.1. Overview 35

2.2.7.2. Evaluation 36

2.2.8. Summing-up 37

2.3. Approaches to library organization 37

2.3.1. The Common KADS library: a general-purpose library for knowledge modelling 37

2.3.2. Libraries of ontologies 38

2.4. Problem solving methods: Organization and development 39

2.4.1. Characterizing and developing problem solving methods 39

2.4.2. Approaches to the organization of libraries of problem solving methods 40

2.4.2.1. Pragmatic and fundamental limitations of local method selection knowledge 41

2.4.2.2. Lack of a clear theoretical basis 42

2.5. Legacy of the review: what needs to be done 42

2.5.1. Modelling framework 42

2.5.2. Development and organization of reusable method components. 43

2.5.2.1. Method characterization and development 43

2.5.2.2. The organization of a library of reusable problem solving components 44

3. An Approach to the Organization of a Library of Problem Solving Methods which Integrates the Search Paradigm with Task and Method Ontologies 45

3.1. From world-view to theory 45

3.2. Characterizing generic tasks 47

3.2.1. Types of generic tasks 47

3.2.2. Generic tasks: viewpoints over applications 50

3.3. Generic task specification as task ontology 50

3.4. From generic tasks to generic problem solving methods 53

3.4.1. Search as an epistemological device to integrate tasks and methods 53

3.4.2. A search-based model for parametric design problem solving 54

3.4.2.1. Parametric design as search 55

3.4.2.2. Identifying generic problem solving actions for parametric design 56

3.5. Characterizing problem solving methods 57

3.5.1. Definition of problem solving method 57

3.5.2. Modelling problem solving methods 58

3.6. Reusable domain models 59

3.6.1. The knowledge interaction problem 60

3.6.2. Integrating domain ontologies and mono-functional models into the framework 64

3.6.2.1. Mono-functional models as customised domain views 64

3.6.2.2. The role of domain ontologies 65

3.7. Conclusions 67

4. Knowledge Modelling in OCML 68

4.1. Introduction 68

4.2. Language tenets 68

4.2.1. Knowledge-level modelling support 69

4.2.2. Support for the TMDA modelling framework 69

4.2.3. Compatibility with emerging standards 69

4.2.4. Integration of formal/informal/operational modelling 70

4.2.5. Support for quick prototyping of knowledge models 70

4.3. Types of constructs in OCML 72

4.3.1. Functional terms 72

4.3.1.1. Control terms 72

4.3.1.2. Logical expressions 73

4.4. Basic domain modelling in OCML 73

4.4.1. OCML relations 73

4.4.1.1. Relation specification options 74

4.4.1.2. Operationally-relevant relation options 74

4.4.1.3. A meta-option for non-operational specifications 76

4.4.1.4. OCML relations: summing up 77

4.4.2. OCML functions 77

4.4.3. OCML classes 79

4.4.4. OCML instances 80

4.4.5. Object-oriented and relation-oriented approaches to modelling 80

4.4.6. The generic Tell-Ask interface 81

4.4.6.1. Tell: a generic assertion-making primitive 81

4.4.6.2. Ask: a generic query-posing primitive 82

4.4.7. OCML procedures 83

4.4.8. Rule-based reasoning in OCML 83

4.4.8.1. Backward rules 83

4.4.8.2. Forward rules 84

4.5. Functional view of OCML 85

4.6. Mapping 85

4.6.1. Instance mapping 86

4.6.2. Relation mapping 88

4.7. Ontologies 89

4.8. Comparison with other languages 91

4.9. Conclusions 94

5. An Ontology for Task-Method Structures 95

5.1. Basic task types 95

5.1.1. Modelling tasks in OCML 95

5.1.2. Executable vs. goal-specification tasks 97

5.2. Modelling problem solving methods 98

5.2.1. Representing methods in OCML 98
5.2.2. Modelling support for library organization 100
5.2.3. Types of methods 101
5.3. Modelling goal expressions 102
5.4. Roles and role values 103
5.4.1. Roles as meta-level concepts 103
5.4.2. Modelling roles in OCML 104
5.4.3. Roles as variables: issues of scope 106
5.5. Carrying out tasks 107
5.6. Application modelling 110
5.7. Conclusions 110
6. An Ontology for Parametric Design Tasks 112
6.1. The nature of parametric design applications 112
6.1.1. Creative design 112
6.1.2. Configuration design 114
6.1.3. Parametric design 114
6.2. Parametric design problem specifications 116
6.2.1. Parameters and design models 117
6.2.1.1. Types of design models 117
6.2.1.2. Legal values 118
6.2.2. Constraints and requirements 118
6.2.3. Key design parameters. 119
6.2.4. 'Better' and 'worse' solutions 120
6.2.4.1. Preferences 120
6.2.4.2. Global and preference-specific cost functions 121
6.2.5. Summing-up 122
6.3. A graphical overview of the parametric design task ontology 123
6.4. An OCML ontology for parametric design tasks 125
6.4.1. Modelling the notion of parametric design task 125

6.4.2. Representing design models ... 127

6.4.3. Representing constraints and requirements ... 128

6.4.4. Modelling preferences ... 130

6.4.5. Modelling costs and cost functions ... 131

6.5. Comparison with other approaches ... 133

6.5.1. Comparison with configuration design ontology by Gruber, Olsen, and Runkel. ... 133

6.5.2. Comparison with DIDS approach ... 134

6.5.3. Comparison with work by Wielinga, Akkermans, and Schreiber ... 134

6.6. Conclusions ... 136

7. A Generic Model of Parametric Design Problem Solving ... 137

7.1 Introduction ... 137

7.2 A search-based model of parametric design problem solving ... 138

7.2.1 Design as search ... 138

7.2.2 State transitions and design operators ... 139

7.2.2.1. The role of design operators ... 139

7.2.2.2. Representing design operators in OCML ... 139

7.2.3 Parameter dependencies ... 141

7.3 Methodological aspects of parametric design problem solving ... 142

7.3.1 Parameters. ... 143

7.3.2 Constraints ... 143

7.3.3 Design Operator ... 144

7.3.4 Cost Function ... 145

7.4 A generic model of parametric design problem solving ... 145

7.4.1 Generic tasks in parametric design problem solving ... 146

7.4.2 Constructing the generic model ... 147

7.4.3 Subtasks of Gen-design-control ... 150

7.4.4 Design state evaluation ... 150

7.4.5 Design state selection ... 152

7.4.6 State-based design process 155

7.4.7 State generation and backtracking 156

7.4.8 Context-centred design 158

7.4.9 Design focus selection 160

7.4.9.1. Variable ordering heuristic and focus selection in design extension context 161

7.4.9.2. Default parameter selection strategy for design extension context 162

7.4.10 Collecting and prioritizing operators 164

7.4.10.1. Task sort-design-operators 164

7.4.10.2. Design extension operators 164

7.4.11 Focus-centred design 166

7.4.12 Design operator selection 167

7.4.13 Applying a design operator 168

7.4.14 Main aspects of the generic model for parametric design problem solving 170

7.4.14.1. Methodological framework. 170

7.4.14.2. Knowledge roles 170

7.4.14.3. Generic tasks 171

7.5 Comparison with other approaches 173

7.5.1 Comparison with DIDS toolkit 173

7.5.2 Comparison with Chandrasekaran 175

7.5.3 Comparison with constraint satisfaction approaches 175

8. Problem Solving Methods for Parametric Design 177

8.1 Introduction 177

8.2 Characterizing problem solving methods 178

8.3 Propose&Backtrack 180

8.4 Hill-Climbing 183

8.5 A*-based design 185

8.6 Beyond uniform approaches to parametric design 187

8.7 Design modification operators 188
8.8 Propose&Improve 190
8.8.1 Modelling Propose&Improve 190
8.8.2 Task-method structure of Propose&Improve 191
8.8.2.1 Focus collection in :improve context 192
8.8.2.2 Focus selection in :improve context 193
8.8.2.3 Operator collection and selection in :improve context 193
8.8.3 Analysis of Propose&Improve 194
8.9 Propose&Revise 196
8.9.1 Introduction 196
8.9.2 Differentiating Propose&Revise 197
8.9.3 Introducing fixes. 198
8.9.4 Task-method structure of Propose&Revise 199
8.9.4.1 EMR vs. CMR architectures 199
8.9.4.2 Modelling Propose&Revise control regimes 200
8.9.5 Methods for design revision 202
8.9.5.1 One-step revision 202
8.9.5.2 Focus-centred revision 203
8.9.5.3 Fix-monotonically 204
8.9.5.4 Focus collection in :revise context 205
8.9.5.5 Focus selection in :revise context 206
8.9.5.6 Operator collection and selection in :revise context 206
8.9.6 Characterizing the P&R-class of problem solving methods 207
8.9.7 P&R-Marcus 209
8.10 Conclusions 210
8.10.1 Classifying problem solving methods 211
8.10.2 Uniform view of problem solving methods 211
8.10.3 Modularity (Plug and Play) 212
8.10.4 Task-independent approaches 213

9. Application Development by Reuse........216

9.1. Introduction........216

9.2. A shell for parametric design problem solving........217

9.2.1. Integrating knowledge-level and symbol-level constructs........217

9.2.1.1. Integration through procedural attachments........218

9.2.1.2. Integration through classes........218

9.2.1.3. Integration through functional interface........219

9.2.2. Symbol-level support for parametric design........219

9.2.2.1. Replacing OCML tasks and methods with CLOS methods........219

9.2.2.2. Optimizing knowledge-level models for symbol-level efficiency........220

9.3. The Sisyphus-I office allocation problem........221

9.3.1. Description of the Sisyphus-I problem........221

9.3.2. Constructing a task model for the Sisyphus-I problem........224

9.3.2.1. Parameters........224

9.3.2.2. Value ranges........225

9.3.2.3. Constraints and Requirements........226

9.3.2.4. Preferences........226

9.3.2.5. Cost function........228

9.3.3. Domain modelling........228

9.3.4. From task to problem solving: specifying design operators........230

9.3.4.1. Multiple design extension operators........230

9.3.4.2. Head of group........231

9.3.4.3. Secretaries........232

9.3.4.4. Manager........233

9.3.4.5. Head of project........234

9.3.4.6. Researchers........235

9.3.5. Modelling constraints and requirements........236

9.3.6. Mapping Knowledge........238

9.3.7. Solving the Sisyphus-I office allocation problem 239
9.3.7.1. Solving by Gen-design-psm 239
9.3.7.2. Solving by HC-design 242
9.3.7.3. Solving by A*-design 242
9.3.7.4. Summing up 243
9.3.7.5. Comparison with other solutions to the Sisyphus-I problem 243
9.4. The KMI office allocation problem 245
9.4.1. Domain model 245
9.4.2. Task model 248
9.4.2.1. Parameters and value ranges 248
9.4.2.2. Requirements and constraints 248
9.4.2.3. Preferences and cost function 249
9.4.3. Design operators 250
9.4.4. Solving the KMI office allocation problem 251
9.4.4.1. Solving by Gen-design-psm 251
9.4.4.2. Solving the KMI office allocation problem by means of Propose&Improve 252
9.5. The VT elevator design problem 257
9.5.1. A critique of the VT domain model provided as part of the Sisyphus-II data set 258
9.5.1.1. Mapping procedures to constraints 259
9.5.1.2. Lack of knowledge about preferred or optimal solutions 260
9.5.2. Constructing a task model for the VT problem 260
9.5.2.1. Parameters 260
9.5.2.2. Requirements 261
9.5.2.3. Constraints 261
9.5.2.4. Preferences and cost function 262
9.5.3. Applying Propose&Revise to the VT domain 263
9.5.3.1. Modelling the Propose step 263

9.5.3.2. Modelling the Revise task ... 266

9.5.3.3. Experimental results ... 269

9.5.3.4. Evaluation of the VT application ... 270

9.5.3.5. Conclusions ... 272

9.5.3.6. Comparison with some contributions from the Sisyphus-II initiative. ... 272

9.6. Conclusions ... 274

10. Concluding Remarks ... 275

10.1 Legacy of the work ... 276

10.1.1 Epistemological foundations of knowledge-based systems ... 276

10.1.2 Problem solving ... 277

10.1.3 Ontologies ... 278

10.1.4 Libraries of problem solving components ... 278

10.1.5 Software Reuse ... 279

10.1.6 Knowledge Acquisition. ... 279

10.1.7 Knowledge modelling languages ... 279

10.1.8 Design Problem Solving. ... 280

10.2 Open Issues for future research. ... 280

10.2.1 'Strategic' issues ... 280

10.2.2 'Scholarly' issues ... 281

10.2.2.1. Validation issues ... 281

10.2.2.2. Application delivery issues ... 281

10.2.2.3. Foundational issues ... 282

10.3 Concluding, Visionary, Techno-political remarks ... 282

References ... 284

Appendix 1. Additional details on the OCML language ... 293

1.1. Functional term constructors ... 293

1.2. Control term constructors ... 295

1.3. Inheritance and default values ... 296

1.4. Interpreters and proof system 297

1.4.1. The OCML interpreter for functional terms 297

1.4.2. The OCML interpreter for control terms.................................... 297

1.4.3. The OCML proof system.. 298

1.4.3.1. Procedure for proving basic goal expressions in OCML.......... 298

1.4.3.2. Proof rules for non-basic goal expressions 298

Appendix 2. Full specification of the task-method ontology..............300

Appendix 3. Full specification of the parametric design ontology........308

Appendix 4. Full specification of gen-design-psm.........................314

Chapter 1.
Knowledge, Models and Reuse

In this chapter I introduce the 'world view' informing this thesis. This world view is characterised by three main themes: knowledge (the subject of investigation), modelling (the chosen approach), and reuse (the pragmatics of the exercise). In particular, the discussion i) emphasizes the non-deterministic nature of knowledge-based systems, ii) reviews the research work which has led to the emergence of the knowledge modelling paradigm, and iii) highlights the synergistic relationship between modelling and reuse.

1.1. INTRODUCTION

1.1.1. The nature of knowledge modelling

In this thesis I will present an approach to the specification, organization, configuration, and development of *reusable components for knowledge models*. The expression 'knowledge model' is a commonly used abbreviation for 'knowledge-level model', a term introduced by Allen Newell (1982) to indicate a description of a problem solving agent which abstracts from implementation considerations and focuses instead on the knowledge it embodies. The key assumption here is that 'intelligent' problem solving behaviour can indeed be explained in terms of a body of knowledge - i.e. there is a causal relation between the agent's knowledge and its behaviour. Newell calls this assumption the *principle of rationality*. Hence, knowledge-level models a-la-Newell are knowledge-centred descriptions of rational problem solving behaviour. For example a knowledge level description of an engineering design program would consist of the various types of knowledge it relies on - e.g. knowledge about the design process in general, knowledge about specific design requirements, knowledge about the various types of applicable constraints, etc. This description is independent of the specific data structures in which such knowledge can be encoded. Analogously, the same type of knowledge-centred approach can be used to describe the expertise of a human designer performing the same or a similar task.

Thus, the object of a knowledge-level analysis is a *knowledge-based system* (*KBS*), i.e. a system whose behaviour is defined by the principle of rationality. Consistently with Newell's theoretical framework and usage of the term, a knowledge-based system can be

a software system, a human being, or even an organization (Nonaka and Takeuchi, 1995). However, the term 'knowledge-based system' is normally reserved for a specific class of software systems, i.e. those built according to a *knowledge engineering* approach (Feigenbaum, 1977). Hence, to avoid confusion, in this work I will reserve the term 'knowledge-based system' to refer to knowledge-based software artefacts, and I will use the more generic term 'problem solving agent', when I wish to emphasize that the object of analysis is not necessarily a piece of software.

Since Newell's proposal, many researchers have pursued the idea of knowledge-level analysis and applied it to a number a fields, such as cognitive science, education, knowledge engineering and *knowledge management* (Wiig, 1994). For example, the Soar architecture (Laird et al., 1987), which is a direct implementation of Newell's knowledge-level framework, has now been used in cognitive science for several years, as a testbed for trying out theories of problem solving, memory and learning (Newell, 1990). In the education area there have been experiments which show the benefits gained by students engaged in model construction (Conlon and Pain, 1996). Stutt (1997) builds on these results and argues that the application of knowledge modelling to education "may provide the means for achieving some of the larger ambitions of constructivist researchers"[1]. In knowledge engineering various knowledge modelling techniques have been proposed to support activities in the system development life-cycle, such as domain and task analysis (Steels, 1990; Chandrasekaran et al., 1992; Wielinga et al., 1992a; Shadbolt et al., 1993), system specification (Jonker and Spee, 1992), knowledge acquisition (Musen, 1989), and validation and verification (Fensel and Schoenegge, 1997a). Finally, researchers in knowledge management are adapting and developing knowledge modelling solutions to support the identification and representation of *organizational knowledge assets* (van der Spek and de Hoog, 1994). These have become especially important in recent years, as Fordist economies are giving way to skill-intensive, service-oriented ones, where knowledge is not so much an asset among others, but the *key competitive resource* (Nonaka and Takeuchi, 1995).

In a nutshell, Newell's vision of a knowledge level which abstracts from implementation considerations is currently being developed and applied well beyond the original idea of providing the 'right' level of description of a problem solving agent. Today, knowledge modelling is a mature, distinct technology whose domain of investigation is the analysis

---

[1] Throughout the thesis I will use double quotes when citing from a publication. I will make use of single quotes either when I wish to emphasize that an expression is not to be interpreted literally, or when the object of discourse is an actual word or expression, rather than its denotation - e.g. to indicate that the word 'cat' is three characters long.

and construction of both 'static' (knowledge-intensive resources) and 'dynamic' (knowledge-intensive processes) models (Motta, 1997).

1.1.2. Context of the work

In this thesis I will instantiate these broad notions concerning the nature of knowledge modelling technology into a specific *modelling framework*, which will provide the basis for i) carrying out a knowledge-level analysis of *parametric design problem solving* (Wielinga et al., 1995; Motta and Zdrahal, 1996) and for ii) developing a number of reusable technologies to support the construction of parametric design applications. These technologies include *task* and *method ontologies*, a *generic model* of parametric design problem solving and a number of *problem solving methods* applicable to parametric design tasks. Thus, the work presented here should be of interest not only to researchers and practitioners working in knowledge modelling, but also to those active in the more general area of software reuse and to developers of design applications.

Below, I outline the main contributions of this thesis to parametric design problem solving and to knowledge modelling, sharing and reuse.

1.1.2.1. Contributions to research in design problem solving.

The contribution of this work to design consists of a number of reusable technologies for parametric design. These include:

- A precise specification of the class of parametric design problems - the parametric design **task ontology**. This provides a generic, conceptual framework for characterizing parametric design problems. An application-specific instantiation of this generic task ontology can be seen as the target of an application analysis (or *requirements engineering*) process.
- A **generic model**[2] **of parametric design problem solving**. This consists of a number of *generic tasks* (Chandrasekaran et al., 1992) and a *method ontology*. The generic tasks define a method-independent framework which can be used for i) characterizing parametric design problem solving, ii) structuring the development of specific problem solving methods, and iii) evaluating, comparing and contrasting alternative problem solving methods.
- A **library of problem solving methods** for parametric design. These subscribe to different paradigms: some are weak search methods; some are case-

---

2 Here, I use the term 'model', rather than 'method', to emphasize that this construction is not necessarily a complete problem solving method. For instance, while it aims to cover all generic tasks associated with parametric design problem solving, it leaves the control structure unspecified.

based; some are based on knowledge-intensive, heuristic approaches. However, they have all been re-formulated in the library as refinements of the generic model of parametric design problem solving mentioned above. This homogeneous characterization of the heterogeneous methods makes it easier to evaluate, compare, and contrast them and also provides structuring principles for organizing the library.

- Examples of **parametric design applications**, developed by reusing and configuring the library components. These examples illustrate and validate the approach to application development by reuse presented in this thesis.

1.1.2.2. Contributions to research in knowledge modelling, sharing and reuse.

The novel contributions of this work to knowledge modelling, sharing and reuse can be formulated in terms of a number of 'slogans'. These summarise the main tenets of my approach to library and application development.

- **Typologies of knowledge modules and ontologies provide the structure for organizing libraries and applications**. Both the library of parametric design components and the applications developed by reuse are structured in terms of different types of ontologies and knowledge modules. In particular, I distinguish between task, method, domain and application knowledge. This distinction is useful in that it provides a framework for structuring the epistemologically diverse types of knowledge embedded in a library or application model. The analogous distinction between task, method, domain and application ontologies is also important, as it provides the 'conceptual handles' needed for acquiring application-specific knowledge and for 'plugging-in' reusable modelling components.

- **Approaches to developing and structuring reusable problem solving components should be theory-based**. This slogan states that a library of reusable problem solving components, which is associated with a class of problems, say P, should be based on some kind of theory describing the problem solving processes which are carried out when solving an instance of P. In other words, a library should be more than a collection of methods: it should be grounded on some theory characterizing problem solving behaviour in a (more or less restricted) space of applications. The approach taken here advocates the construction of a generic model of problem solving, which is specific for a class of tasks, but independent from any particular problem solving method - see next bullet point.

- **Task-specific, method-independent problem solving model = search + task ontology**. A generic problem solving model for a *problem type* (i.e. for

a class of applications) can be developed by instantiating a *generic problem solving paradigm*[3] - e.g. search - in terms of a task ontology. This principle sloganizes the approach taken to develop the problem solving model on which the library of parametric design components is based. The advantage of this approach is that it outlines a model of how to move from the specification of a problem type to a class-specific, but method-independent model of problem solving. Thus, it is possible to build generic theories of problem solving behaviour for a class of applications, which closely integrate the task and method 'dimensions' and make it possible to characterise problem solving methods precisely - see next bullet point.

- **Problem solving method = refinement of task-specific, method-independent, generic problem solving model.** A problem solving method is defined as a particular specialization of the generic problem solving model associated with a problem type, say Gen-PSM, and its method ontology is a refinement of the method ontology associated with Gen-PSM. This slogan precisely defines the somewhat vague notion of problem solving method, which is typically defined as "a way to solve a task". The approach characterises the nature of problem solving methods, provides a basis for a method development methodology and facilitates the process of evaluating, comparing and contrasting alternative problem solving methods applicable to the same problem type. This capability is especially useful when the methods are heterogeneous, i.e. subscribe to different paradigms (e.g. case-based vs. search-based methods).

The above slogans specify the main tenets of the approach used for developing and structuring a library of model components. Of course, in order to develop a library, one needs not just a methodology and a modelling framework but a *modelling language* too.

A modelling language needs to address many, often inconsistent requirements. For instance it should support informal modelling, provide a pathway to operationalization and possibly have a formal semantics. Moreover, it has to address the trade-off between the need to support the full expressiveness required by ontologies and software specifications (Hayes and Jones, 1989) and the advantages offered by executable specifications (Fuchs, 1992). Finally, for pragmatic reasons it is important to be compatible with emerging standards. In this thesis I will use the OCML modelling

---

[3] The term 'paradigm' refers to a "philosophical and theoretical framework of a scientific school or discipline within which theories, laws, and generalizations and the experiments performed in support of them are formulated" (Merrian-Webster, 1997). Here, I use the expression 'generic problem solving paradigm' to refer to a theoretical framework providing a foundation to problem solving, be it cognitive, semantic, philosophical, or computational.

language (Motta, 1995). This supports informal, formal, and operational modelling; integrates the specification of ontologies with that of reasoning components and provides degrees of compatibility with emerging standards for ontology specification, such as Ontolingua (Gruber, 1993), and for knowledge modelling, such as Common KADS (Schreiber et al., 1994b).

1.1.2.3. Knowledge, models, and reuse

As I said at the beginning of this chapter, this thesis is about reusable components for knowledge models. Another way of characterizing it is to say that it is informed by three main 'themes': **knowledge**, **models**, and **reuse**. These define the area of interest (**knowledge-based systems**), the world view underlying the chosen approach (**knowledge modelling**) and the pragmatics of the exercise (**knowledge sharing** and **reuse**). In the rest of this introductory chapter I will discuss these themes in detail - thus clarifying the research context for the work described here - by i) characterizing the class of knowledge-based systems (**knowledge**); ii) providing a (necessarily subjective) reconstruction of the research process which has led to the formulation of the knowledge modelling paradigm (**models**); iii) illustrating the economic and scientific factors stimulating research on shareable and reusable knowledge bases (**reuse**); and iv) emphasizing the synergistic relation between modelling and reuse (**reuse as abstraction**).

1.2. A CHARACTERIZATION OF KNOWLEDGE-BASED SYSTEMS

1.2.1. The ambiguous notion of knowledge-based system

That the notion of a knowledge-based system needs to be clarified might be surprising to some, given that knowledge-based systems have been explicitly recognised as a distinct class of software systems for at least two decades - i.e. at least since Feigenbaum first used the term 'knowledge engineering'. Nevertheless, both from discussions with students and other software practitioners, and from reading the literature, it emerges that different people have different views on what is a knowledge-based system and that many find this notion unclear.

The main problem here is that 'knowledge' is a very fuzzy word. For instance a payroll system requires knowledge about salary scales, tax codes, allowances, benefits, and other specialised expertise and uses this body of knowledge in order to compute the net salary of an employee. Hence, one could (maybe should) view a payroll system as an example of a knowledge-based system. Nevertheless, many researchers in knowledge engineering would not accept that the typical payroll system found in an organization is 'really' a knowledge-based system. They would point out that such a system does not embody a 'reasoning process'. It just follows a procedure which deterministically

generates a result. This criticism suggests that an explicit encoding of task or domain specific knowledge is not sufficient to 'qualify' as a knowledge-based system: such a property has more to do with the way a solution is achieved, rather than with specific data structures. The criticism also implies that the embodied knowledge we refer to in the context of knowledge-based systems is in fact specialised in some sense - i.e. not all knowledge-embedding systems are knowledge-based. Of course one can also take the view that this criticism to the 'ordinary' payroll system is not just unclear but also unfounded. Maybe humble payroll systems are after all truly knowledge-based. Obviously, whether a system is or is not knowledge-based is not so much a fundamental characteristic of the system as a matter of complying with some operational definition. Hence, my goal here is not to characterise fundamental cognitive properties, such as 'intelligence' and 'skilled problem solving', but rather to provide a pragmatic definition which is adequate to set the context for the rest of this work.

The problem of characterizing knowledge-based systems is also compounded by the existence of other ambiguous and controversial terms, such as *expert system* and *intelligent system*, which are often used as synonyms for knowledge-based systems. For example, let's consider the term 'expert system'. Is an expert system "a computer model of expert human reasoning", which would reach "the same conclusions that a human expert would reach if faced with a comparable problem" (Weiss and Kulikowski, 1984 - page 1), or is it a system which embodies a *model of expertise* (Wielinga et al., 1992a), but makes no claim about modelling human reasoning (Schreiber, 1992). A choice in favour of the former view strongly situates expert system research in a cognitive science context and raises important questions about the nature of expertise and the relationship between human and machine intelligence. Adopting the latter view implies taking a stand which separates expert system construction from cognitive modelling and therefore emphasizes engineering rather than cognitive issues. Again, the point here is that as a prerequisite to a discussion concerning knowledge-based systems one needs to try and clarify this notion and take a stand with respect to various possible perspectives. In the next section I will attempt to do exactly this and in the process I will introduce some important themes informing the approach taken by my research, such as the fundamental role of *search techniques* in knowledge-based problem solving.

1.2.2. The existential predicament of knowledge-based systems

A knowledge-based system is typically described as a computer system which uses knowledge to solve a task (Stefik, 1995). While this definition is correct, it is not very useful on its own; it basically says that a system is knowledge-based if it makes use of knowledge. Which is a bit of a tautology. Moreover, as illustrated by the example of the payroll system, relying on intuitive notions of what it means to be knowledge-based leads to confusion and disagreement. Therefore, in order to provide a more useful

characterization of knowledge-based systems, Stefik (1995) goes one level deeper and tries to define the meaning of the word 'knowledge' in the context of knowledge-based systems. He says that knowledge refers to the *codified experience of an agent.* This definition captures three important aspects associated not just with research in knowledge-based systems but also with much research in the wider field of *artificial intelligence* (AI): *representations, agents,* and *experience.* I will examine each one in turn.

1.2.2.1. Knowledge as representation

The definition of knowledge-based system given by Stefik points out that in order to talk about knowledge in the context of knowledge-based systems, we need a representation, i.e. this knowledge must be explicitly codified in the system's data structures. Such an encoding is a necessary property of knowledge-based systems. In other words, a system can be regarded as knowledge-based only if it contains structural ingredients that "we take to represent a propositional account of the knowledge that the overall process exhibits" (Smith, 1982). This emphasis on knowledge representation has historically played a fundamental role in AI research; some researchers have argued that the requirement for an explicit encoding of knowledge is not just a necessary feature of knowledge-based systems, but a necessary and sufficient condition for intelligent behaviour. This thesis was formulated by Newell and Simon (1976), who called it the *physical symbol system hypothesis.* Needless to say, not everybody agrees with such a conjecture, which has been criticised both within (Brooks, 1991) and outside (Dreyfus, 1979) AI. However, in the context of this thesis (i.e. in the context of knowledge-based systems), this is not so much a hypothesis as a definition: there is no knowledge-based system without explicit knowledge representation.

1.2.2.2. Knowledge-based systems as agents

Stefik's definition also relates the notion of knowledge to that of agent. The word 'agent' is currently much en vogue in AI and software engineering circles but unfortunately there is little in common between the different uses of the term which can be found in the literature (Bradshaw, 1996). Stefik uses this term to characterize two aspects of the knowledge associated with a knowledge-based system. First of all, he wants to emphasize that the explicitly encoded knowledge makes sense only with respect to an agent/observer, which is able to interpret it. In other words a representation has no meaning per-se. Secondly, because an agent is by definition an acting entity, it follows that the knowledge embodied by a knowledge-based system can be characterized as *potential for generating actions* (Newell, 1982 - page 100). Hence, when trying to describe knowledge-based systems, it is not enough to say that they explicitly embody knowledge, but it is crucial to highlight the *problem solving nature* of the embodied knowledge.

In order to make this notion of problem solving knowledge more precise we need to look at the kind of tasks which are solved by knowledge-based systems. By definition these are complex tasks whose solution requires knowledge. From a problem solving point of view a task is complex if there is no *direct method* which can effectively solve it (i.e. given time and resource limitations). Here I use the term 'direct method' to refer to an algorithmic, deterministic procedure, which guarantees to find a solution to the task. Intuitively, this definition means that complex tasks are those which force a problem solver to resort to 'guessing'. More precisely, the essential feature of complex problem solving scenarios is that *decisions have to be taken under uncertainty.* As a result, there is no guarantee that any particular decision is correct. Given this scenario we can then define problem-solving knowledge as follows.

> *Problem-solving knowledge is knowledge which is brought to bear during a problem solving process, when a system is faced with uncertainty in choosing among a number of alternatives.*

This definition implies that problem solving knowledge is more than a typical conditional in a conventional software module. The latter specifies alternative computational options, the choice of which depends on the case-specific input. However, each path is assumed to be correct and to lead to a solution for a given input. In the case of knowledge-based systems there is no such assurance. Given the uncertain context of the problem solving process, it follows that there is no guarantee that any particular decision is correct. As a result the problem-solving agent has to *search.* Newell (1990) calls this property the *existential predicament of intelligent systems.*

To clarify this point and to provide a concrete instance of problem solving knowledge, I shall illustrate an example taken from the VT elevator configuration application (Marcus et al., 1988; Schreiber and Birmingham, 1996). The problem consists of computing the specification of an elevator which satisfies a number of structural (e.g. number of floors) and functional (e.g. elevator capacity) requirements. From a computational point of view the process is one of *parametric design* (Motta and Zdrahal, 1996), in which values have to be assigned to a number of parameters in a way which satisfies the problem requirements and does not violate any constraint.

Figure 1.1 shows some of the parameters in the VT application. In particular, the parameter `cwt-to-hoistway-rear` specifies the distance between the counterweight and the rear of the hoistway. In absence of relevant domain constraints, we can assume that the counterweight can be positioned anywhere in the space defined by the distance between the platform and the rear of the hoistway (this space is called "counterweight space"). However, the description of the VT application (Yost & Rothenfluh, 1996) indicates that only two options are ever considered by the VT domain experts, when

deciding on the position of the counterweight. The preferred solution locates the counterweight half way between the platform and the U-bracket. If this solution does not work (because the counterweight is too close to the U-bracket), then the alternatives are: i) to reduce the depth of the counterweight; ii) to move the counterweight closer to the platform; and iii) to increase the counterweight space (presumably by decreasing the depth of the platform). In practice all this means that a VT domain expert only considers two options for positioning the counterweight: either half way between the platform and the U-bracket, or, in case the previous approach fails, a position ensuring that its distance from the U-bracket is more than 0.75 inches.

This example provides a nice example of the pragmatics of problem solving knowledge. In search-oriented terms, problem-solving knowledge is knowledge which is used to reduce or navigate efficiently the *search space* associated with a problem. In this case, VT domain experts know that they can reduce the problem of finding a position for the counterweight down to two choices (rather than an infinite number). Moreover, they also provide an ordering of these two choices, which is based on a cost-minimization criterion. However, in a problem solving situation neither of these choices can be regarded as correct 'a priori'. Depending on the particular case specification it is possible that a design system using this knowledge will have to *backtrack*. This situation is a particular instance of a general case: the existential predicament of knowledge-based systems implies that backtracking, whether chronological (Golomb and Baumert, 1965), *dependency-directed* (Stallman and Sussman, 1977) or *knowledge-based* (Marcus et al., 1988), is a crucial feature of knowledge-based problem solving.

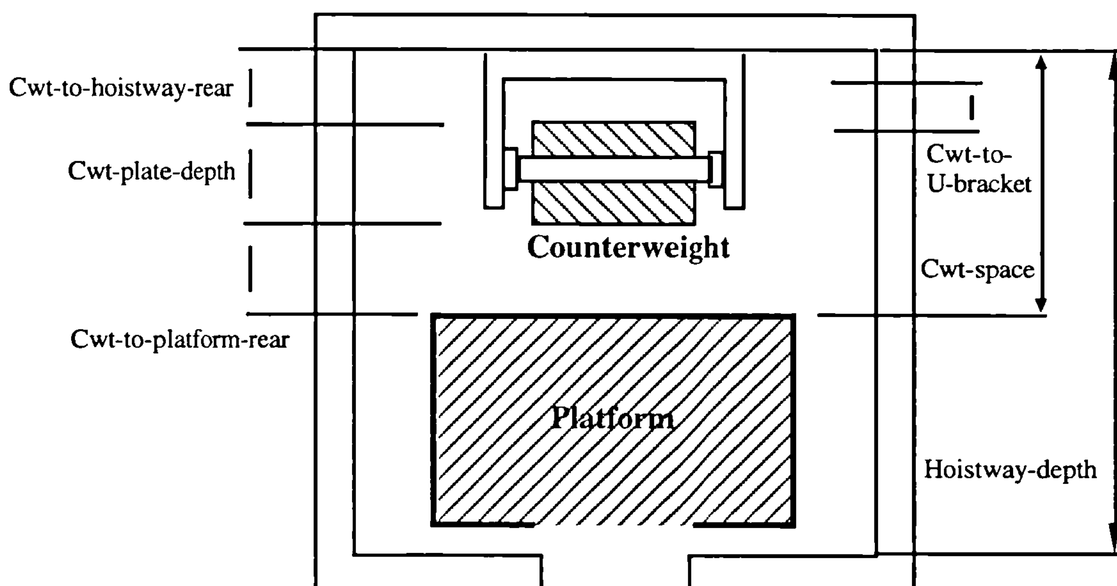


Figure 1.1. The elevator example

It should now be clear why the paradigmatic payroll system discussed above is not 'really' a knowledge-based system. While it can be said to encode domain knowledge, it cannot be viewed as a system which performs search or takes decisions under uncertain conditions. On the contrary, it employs direct methods which are guaranteed to find a solution without searching. In other words, payroll systems do not backtrack[4].

1.2.2.3. The experiential nature of problem-solving knowledge

Stefik's characterization of knowledge also emphasizes the importance of experience as the source of knowledge. This is in line with the view of knowledge systems as expert systems, i.e. systems which exhibit expertise in a particular area. By putting the accent on experience, Stefik points out that problem solving knowledge tends to be experiential in nature: it is knowledge which is learnt through problem solving, or acquired from external sources through a process of *knowledge acquisition* (*KA*). This property is nicely illustrated by the example discussed above. It is precisely domain expertise which allows domain experts to choose the 'probably right' position for the counterweight and which needs to be acquired by a software agent trying to configure the same class of elevators.

1.2.3. The role of knowledge-based systems

The above discussion clarifies - I hope - the nature of knowledge-based systems. In a nutshell,

> *Knowledge-based systems are complex decision-making systems, which use problem solving knowledge when taking decisions under uncertain conditions. Their search-centred behaviour is an inevitable consequence of their 'existential predicament'.*

Of course, a disadvantage of a search-oriented approach is that necessarily it is not as efficient as an approach based on direct, procedural methods. As a result, one often gets asked what's the point of knowledge-based systems compared to conventional systems. Wouldn't it be better to use an algorithmic approach to solve complex problems, rather than relying on expensive search?

The answer to this question is implicit in the discussion on complexity presented above. For typical knowledge-intensive tasks - e.g. design, scheduling, diagnosis - it is the case that not only i) there is no general-purpose algorithm, but ii) there cannot even be one. The reason for this state of affairs is that these tasks are normally *intractable* (Bylander,

---

4 Although the problem solving agents (human or artificial) employed by the Inland Revenue do backtrack a lot!

1991; Bylander et al., 1991; Nebel, 1996), i.e. there exists no effective direct method, which can solve them in polynomial time. Hence, an alternative approach has to be taken, which is based on the use of problem solving knowledge to tackle complexity. Of course, if a task is intractable, it remains intractable even when a KBS approach is used. However, the idea here is that by means of task and case-specific problem solving knowledge a KBS can solve many cases efficiently and reduce the complexity of the problem in the average case.

This situation leads to what I regard as the *existential paradox of knowledge-based systems*. A knowledge-based system is - by definition - a system which performs search. However, search is expensive and should be minimized. Therefore, it follows that:

> *Developing knowledge-based applications requires weakening task specifications, acquiring problem solving knowledge, and strengthening problem solving methods in order to reduce search.*

In other words, knowledge-based system development is about making knowledge-based systems behave as much as possible like ordinary systems.

1.3. EVOLVING PERSPECTIVES ON KNOWLEDGE ACQUISITION

In the previous section I have discussed the nature of knowledge-based systems in particular emphasising that is the reliance on problem solving knowledge which distinguishes them from conventional systems. Problem solving knowledge is experiential in nature in the sense that it is either acquired from external sources of expertise, or learnt through problem solving. As a result knowledge acquisition is central to knowledge engineering and is the main activity which distinguishes the development process of a knowledge-based system from that of a conventional system. Given such a central role, it is not surprising that at each stage of its evolution, the knowledge engineering field as a whole tends to be characterised in terms of the knowledge acquisition paradigm prevailing at a particular time. In particular, the *mining view* of knowledge acquisition (Davis, 1979; Kidd, 1987) characterizing early expert systems has over the years gradually given way to a view of knowledge acquisition as a *modelling* activity (Breuker and Wielinga, 1989). The latter informs the work described in this thesis and therefore in the next section I will characterise its main tenets and show how its development was inspired by the cognitively motivated criticisms and practical difficulties associated with first generation expert systems.

1.3.1. Knowledge acquisition as mining

The research programme which is often taken as paradigmatic of the mining approach to knowledge acquisition is the Mycin project on medical expert systems, which was carried out during the seventies at Stanford University (Buchanan and Shortliffe, 1984). There are two reasons for this. First, it is probably right to say that, among the early expert systems, Mycin was the one which had the most impact. It generated large amounts of literature both directly and indirectly, it tackled tasks (diagnosis and therapy of infectious diseases) which are carried out by experts (i.e. physicians) who are highly regarded by the society as a whole, and (shockingly) it exhibited a performance which was comparable to that of specialised physicians. The second reason (which is basically a consequence of the previous one) is that Mycin was thoroughly analysed (and criticised) in two papers (Clancey, 1983; 1985), which were among the most influential in determining a paradigm shift in knowledge acquisition research in the second half of the eighties.

Representation Formalism	Rules
Knowledge Categories	Facts and heuristic problem solving rules
KA Methodology	Direct encoding of elicited knowledge in rule-based system
Levels of Descriptions	Only one, in terms of the rule-based representation
KA Paradigm	Transfer of Expertise
Cognitive Paradigm	Production systems as general problem solving architectures for intelligence
Reuse	Inference Engine

Table 1. Characterization of the Mycin approach.

Table 1 describes the main aspects of the mining approach to knowledge acquisition[5], as exemplified by the Mycin project. The Mycin performance system (Shortliffe, 1976) has a uniform, rule-based representation, which provides the only level of description of the system. New rules can be acquired from an expert, using an interactive knowledge

---

[5] The expression 'knowledge acquisition as mining' refers to the assumption (which underlies this approach) that discrete and distinct 'gems of expertise' can be elicited one by one from the expert and encoded in the system.

acquisition tool (Davis, 1979). What is reused is the underlying rule-based shell, EMYCIN (van Melle et al., 1984), which was developed by abstracting from the rule-based representation and deductive mechanisms used by Mycin.

The underlying knowledge acquisition methodology is therefore one in which knowledge is directly elicited from an expert in a form suitable for computational encoding. The aim here is to built a virtual expert, i.e. a system which can emulate the problem solving behaviour of an expert by relying on the same body of knowledge. Hence, the approach followed here strongly relies on the assumption that expertise does in fact consist of (or can be at least reformulated as) a set of rules. The cognitive basis for such an assumption can be traced back to Newell and Simon (1972) who proposed production systems as a general computational paradigm for describing intelligent behaviour[6].

1.3.2. Cognitive and technical problems with the mining view

A practical problem with this style of expert system development was recognised very early and dubbed the *knowledge acquisition bottleneck* (Feigenbaum, 1977). The expression refers to the fact that the expert system development process was often hindered by the difficulties associated with eliciting knowledge from an expert and coding it into the system. It is easy to see that this problem is an obvious consequence of the approach chosen. The 'KA as mining' development scenario is one in which i) system development is essentially incremental rule acquisition and ii) knowledge acquisition consists of an interactive transfer of expertise from expert to system - Buchanan and Shortliffe (1984) refer to this process as *knowledge programming.* Therefore, in this scenario the expert is not just one of the players involved in a subset of the system development life-cycle but the person who is central to the whole process - i.e. the main bottleneck.

The knowledge acquisition bottleneck seemed to provide evidence to the arguments put forward by those researchers (Winograd and Flores, 1986; Dreyfus, 1979), who rejected the idea that 'true expertise' could be transferred from an expert to a software system and reduced to a rule-based representation. These authors argue that expertise is by its nature *tacit* (i.e. not all human expertise is amenable to verbalization and formalization) and *situated* (human knowledge is context-dependent and this context cannot necessarily be shared with a computer program). As a result, enterprises such as expert systems are misguided in principle and the knowledge acquisition bottleneck an inevitable side effect of a reductionist view of expertise.

---

6 It is important to note that the researchers working on Mycin were indeed aware of the cognitive assumptions of their work (Davis and King, 1977) and that this awareness contributed to the choice of rules as the uniform representation paradigm.

A paradigm shift was therefore needed, in order to address these problems. The work on knowledge representation by Newell and Brachman, and Clancey's analysis of rule-based expert systems provided the main research breakthroughs which eventually led to the formulation of the modelling paradigm. These are discussed in the next section.

1.3.3. Multiple levels of description

The approach exemplified by the Mycin project considers expert knowledge and rule-based representation as essentially equivalent - knowledge acquisition is an interactive transfer of if-then associations. This uniform approach to representation was criticised by Clancey (1983), who showed that - at least in the case of Mycin's knowledge base - it fails to capture important conceptual distinctions in the acquired knowledge. In particular, Clancey points out that Mycin's explanation facilities are not adequate to explain its diagnostic behaviour to medical students. The reason - argues Clancey - is that these facilities are system rather than domain oriented. They describe Mycin's reasoning in terms of goal-driven behaviour (an implementation-level feature) rather than in terms of the problem solving structures relevant to the diagnostic process in medicine, such as the hierarchy of diagnoses and the underlying symptoms-diseases causal model. Mycin does contain this knowledge, but in a 'opaque' form, as if-then associations.

Clancey's analysis is an important step forward because it shows that it is both feasible and useful to decouple the description of problem solving structures and behaviour from the description of system structures and behaviour. In other words even if knowledge can be formalised as rules and effectively used in problem solving, it does not imply that the representation really captures all relevant conceptual distinctions required in alternative contexts (e.g. tutoring).

At approximately the same time as Clancey's analysis of Mycin empirically showed the utility of abstract, domain-oriented system descriptions, Newell (1982) and Brachman (1979) tried to provide clearer frameworks in which to structure the various approaches to knowledge representation which were being pursued at the time. Consistently with Clancey's analysis, they indicated the existence of several levels of description and pointed out that much of the confusion in the knowledge representation area was caused by the fact that researchers were developing (and attempting to compare) formalisms which were situated at different levels. In particular Newell introduced the distinction between *knowledge* and *symbol level*, thus emphasising the importance of separating the analysis and modelling of knowledge-based problem solving behaviour from the activity of representing it in a computationally efficient formalism. Brachman' s paper proposes a more fine-grained breakdown, which distinguishes five representation levels, *implementation*, *logical*, *epistemological*, *conceptual* and *linguistic*.

Regardless of the differences in approach and purpose which can be found in Newell's and Brachman's analyses, both authors essentially stress a common point. It is both useful and necessary to provide multiple levels of descriptions of knowledge-based systems. Blurring these distinctions leads to opaque systems - as shown by Clancey's analysis of Mycin - and to difficulties in situating and comparing approaches to knowledge representation - as shown by the heterogeneous answers to Brachman and Smith's questionnaire on knowledge representation languages (Brachman and Smith, 1980).

The benefits which could be gained from applying Newell's idea of knowledge-level analysis were illustrated by Clancey (1985), who showed that not only it was possible to uncover from Mycin-like systems their knowledge-level (and implicit in the design) problem solving structures, but also that these structures were common to different systems, i.e. *generic*. In particular, Clancey analysed the behaviour of a dozen rule-based systems tackling problems in various domains and found that their problem solving behaviour could be characterised in terms of a generic *heuristic classification* model - see figure 1.2.

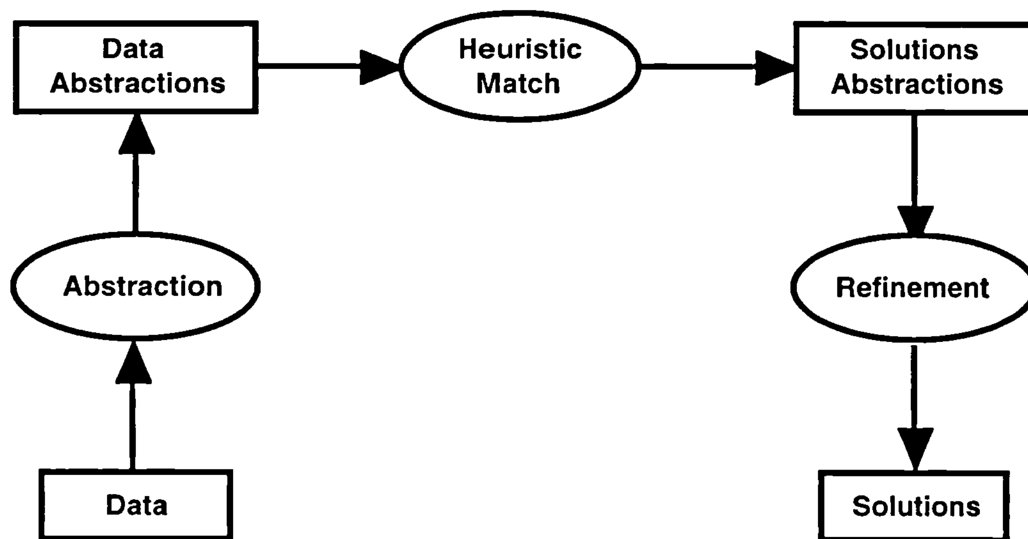


Figure 1.2. Clancey's heuristic classification model.

This model consists of three problem solving inferences which i) generate abstractions from the given data - e.g. infer an abstract characterization of a patient, such as 'immunosuppressed', from then available data, e.g. low white blood count; ii) match these data abstractions to possible solution types (e.g. classes of diseases) and then iii) refine these to produce one or more solutions (e.g. a specific diagnosis).

It is difficult to overestimate the importance of this work. Here Clancey shows that a knowledge-level analysis makes it possible to understand *what* a system actually does, rather than *how* it does it - in the paper Clancey complains about the 'blinding effect' of

the implementation terminology used in rule-based systems which made understanding the problem solving competence of these systems much more difficult. Moreover, by showing that the heuristic classification model is generic, Clancey uncovered the principle of *role differentiation*, which has subsequently informed much knowledge engineering research (Wielinga et al, 1992a; Mc Dermott, 1988; Chandrasekaran et al., 1992). Role differentiation means that it is possible to describe problem solving agents in terms of generic models, which impose specific *problem solving roles* on the domain knowledge. For example, domain structures in different application domains, such as diseases and book classes, actually play the same role (e.g. solution abstraction) when a heuristic classification model is used to describe the problem solving behaviour of a pulmonary infection and a book selection system.

1.3.4. Knowledge acquisition as modelling

Wielinga and Breuker (1984; 1986; Breuker and Wielinga , 1987) were among the first to apply to knowledge acquisition the lessons drawn from the work carried out by Clancey, Brachman, and Newell. In particular, they argued that the so-called bottleneck was caused by the fact that "the mapping between the verbal data on expertise and the implementation formalisms is not a simple, one to one correspondence". Therefore, in developing the KADS methodology (Wielinga et al., 1992a), they proposed an approach in which expertise modelling and design are clearly separated. First, "in an analysis stage, the knowledge engineer develops an abstract model of the expertise from the data.....this model is (then) transformed into an architecture for the KBS" (Breuker and Wielinga, 1989). Thus, they made the case for the development of conceptual modelling frameworks, addressing the issue of characterizing expertise at a level independent from implementation. A similar approach was also taken by my colleagues and myself working on the KEATS project (Motta et al., 1989), in which we distinguished between modelling "overt behaviour" (i.e. understanding problem solving behaviour) and "internal representation" which was concerned with the realization of this behaviour on a computer system.

Other researchers (Mc Dermott, 1988; Musen et al., 1987) set to the task of putting the role differentiation principle into practice, by developing knowledge acquisition tools based on task-specific, but application-independent problems solving models.

Of course there are differences between the approaches followed by all these researchers. Nevertheless, it is possible to group all these efforts around a common paradigm, which considers knowledge acquisition as a *modelling activity*. This paradigm was informed and stimulated by the developments and the problems discussed earlier, in particular i) Clancey's discovery of generic problem solving structures in first generation expert systems; ii) Brachman and Newell's work on system stratification; iii) the practical

knowledge acquisition problems associated with first generation expert systems, and iv) the cognitively-motivated criticisms of the mining approach. Below I list the main features of the modelling paradigm.

- Knowledge engineering is not about cognitive modelling (i.e. 'reproducing' expert reasoning) but about developing systems which perform knowledge-based problem solving and which can be judged on task-oriented performance criteria.
- There are enough similarities between classes of applications, which make it possible to build generic models of problem solving.
- Knowledge acquisition should not be characterised as a process of mapping expert knowledge to a computational representation, but is a model-building process, in which application-specific knowledge is configured according to the available problem solving technology. The knowledge acquisition process is partly negotiation, part reconstruction, part context sharing between a number of stakeholders. The goal of knowledge acquisition is to develop a model of problem solving behaviour. In the words of Ford et al. (1990), "The mining analogy notwithstanding, expertise is not like a natural resource which can be harvested, transferred, or captured, but rather it is constructed by the expert and *reconstructed* by the knowledge engineer".
- It is useful to describe such a model of problem solving behaviour at a level which abstracts from implementation considerations (the knowledge level). This approach has the advantage of separating problem solving from implementation-related issues.
- Given that i) knowledge acquisition is about model construction and that ii) models can be application-generic, it follows that these generic models can be used to provide the interpretation context for the knowledge acquisition process (i.e. the knowledge acquisition process can be *model-based*). In this scenario, much of the knowledge acquisition task can be reduced to acquiring the domain knowledge required to instantiate generic problem solving roles (Marcus, 1988).

Table 2 characterises the modelling approach in terms of the same framework I used to characterise the mining view. In particular, the table shows a paradigm shift from an implementation-oriented to a knowledge-oriented view of knowledge acquisition. Multiple levels of descriptions are introduced and as a result the choice of implementation-level formalisms becomes less important. The knowledge categories are characterised at a conceptual, rather than computational level. The goal is no longer to emulate an expert by means of some kind of 'expertise mapping', but to acquire the domain knowledge required to configure a generic problem solving model. Thus, the context and the aims of the knowledge acquisition process are less amenable to the cognitively-motivated

criticisms aimed at the mining approach. Researchers subscribing to the modelling approach no longer make claims of building rule-based cognitive models of experts and acquiring expertise by 'direct transfer'. The cognitive paradigm underlying the modelling approach can be characterised as a pragmatic one, which is based on a *functional view of knowledge*. Knowledge is what an observer attributes to an agent to explain its problem solving behaviour. It is neither a data structure in a system nor 'stuff' in the expert's mind, it is what enables a knowledge-based system to handle complexity - i.e. the *medium at the knowledge level* (Newell, 1982). The advantage of this approach is that it makes it possible to characterize knowledge modelling as a distinct technology, which focuses on knowledge-based behaviour *per se*, independently of cognitive or machine-centred biases.

Representation Formalism	Level-dependent
Knowledge Categories	Differentiation is driven by generic knowledge roles
KA Methodology	Model-based
Levels of Descriptions	Multiple (e.g. knowledge vs. symbol level)
KA Paradigm	Model Construction
Cognitive Paradigm	Functional view of knowledge
Reuse	Generic Task, Generic Problem Solving Model, Generic Domain Model

Table 2. Characterization of the modelling approach

1.3.5. A minor caveat

The above discussion, being only a short résumé of the evolution of the field, may give the impression that the knowledge modelling and the mining approach are antithetical and that the shift from one to another was radical and clearly marked. Of course, in reality the process was one of slow evolution in which more and more elements of the modelling approach were gradually assimilated in first generation expert systems. For instance, the methodology discussed in (Hayes-Roth et al., 1983), which is often referred to in the context of discussions about first-generation expert systems, distinguishes between a conceptualisation stage and a formalization stage. This distinction is similar to the knowledge vs. symbol level one. At the conceptualisation stage questions such as "what processes are involved in problem solution" and "what types of data are available" are

asked, which are similar to the kind of questions one would need to ask - for instance - to build a KADS model of expertise. It is therefore fair to say that in many cases the differences between the earlier knowledge acquisition methodologies and those based on the modelling view have more to do with emphasis (from representation to conceptualization) and level of detail (from simple, informal conceptualizations to knowledge-level modelling languages) than antithetical world views. Having said so, there is no doubt that the work by Newell and Clancey especially made explicit and systematised notions such as 'intermediate representation' and 'abstract model' which were only hinted at in earlier expert system literature.

1.4. REUSE IN KNOWLEDGE BASED SYSTEMS

1.4.1. Economic and scientific motivations for reusable knowledge bases

In a 1991 paper published in the AI Magazine (Neches et al., 1991) a group of American researchers discussed the state of KBS technology at the time and pointed out that, given the increasing complexity of knowledge-based applications, it had become of crucial importance to move from a 'develop-from-scratch' to a reuse-centred model of KBS development. In the opinion of Neches et al., at the end of the eighties KBS technology had reached a stage in which the cost of building applications from scratch had become too high and therefore "enabling technology for knowledge sharing" was required to reduce such a cost and to make it possible to build larger, more ambitious applications. In a sense one can say that KBS technology was going through the kind of 'coming of age' problems, which conventional software applications had experienced about two decades earlier, when the so-called 'software crisis' led to the development of fourth generation languages. Incidentally, it is important to keep in mind that a reuse-oriented approach to software development provides - at least in theory - much more than a mere reduction of the financial cost incurred when developing an application from scratch. By building an application out of pre-existing, robust components, developers are also expected to reduce the costs associated with application evaluation and maintenance.

In addition to the economical case for reuse, there are also some important scientific reasons which are stimulating research and development on shareable and reusable knowledge bases. For example, the researchers working on the Cyc project (Lenat and Guha, 1990) have developed a large body of common sense knowledge which - they hope - could be used to build systems able to overcome the 'brittleness' typically shown by knowledge-based applications. This effort can be seen both as a serious attempt at understanding common-sense models of the world as well as an exercise in building a large, reusable, *multi-functional* knowledge base. Another example of this beneficial synergy between building reusable models and understanding fundamental knowledge engineering issues can be found in the ongoing work on libraries of *problem solving*

methods (Marcus, 1988; Le Roux et al., 1993; Benjamins, 1993; Breuker and Van de Velde, 1994; Motta and Zdrahal, 1997; Chapter 8). While these efforts have a strong pragmatic connotation, they also help to define the space of problem solving methods, thus contributing to the development of an overall theory of problem solving in knowledge engineering. As a matter of fact, given the current state of KBS technology and the limited amount of software reuse in practice, it is probably accurate to say that current work on KBS reuse is mainly driven by scientific, rather than commercial reasons. Incidentally, Krueger (1992) points out that also in the conventional software engineering area software reuse is still very limited.

The examples given above - libraries of problem solving methods and multi-functional knowledge bases - specify two of the three main categories of reusable knowledge-based components. The third one comprises generic task specifications, such as diagnosis, design, etc. Steels calls these three categories the main *components of expertise* (Steels, 1990). Typically, individual research efforts on reusable knowledge bases have focused on one of these three categories and in particular on the development of category-specific libraries of reusable components. In the previous paragraph I have already mentioned a number of approaches to developing libraries of problem solving methods. Other researchers (Hayes-Roth et al., 1983; Clancey, 1985; Breuker and Van de Velde, 1994) have published structured task classifications, although these are often described at a very generic level. Research on reusable domain models is currently very vigorous, in particular thanks to the interest in *domain ontologies* (Gruber, 1991). The term 'ontology' is used in knowledge engineering to indicate "a reusable specification of a conceptualization" (Gruber, 1993). These 'conceptualizations' can be proposed to foster consensus among researchers on the right set of concepts required to model a domain (Valente and Breuker, 1996), or to describe formally a specialised representation schema, as in the case of the *Frame Ontology* available on the Ontolingua server (Farquhar et al., 1996).

With few exceptions - e.g. see (van Heijst, 1995) - much of the work on ontologies has had little contact with the parallel work on models of problem solving. In this thesis I will try to integrate these two strands of research by i) introducing a language suitable for specifying both problem solving methods and domain ontologies, and ii) using ontologies for driving the development and the organization of problem solving libraries.

1.4.2. Reuse and knowledge modelling

Although I have introduced separately the three main themes of this thesis (i.e. knowledge, models, and reuse), it is probably clear by now that they are strongly interrelated. Knowledge formulation is basically a model construction process, therefore knowledge exists as long as there is a model of it. At the same time I have shown that

most research on knowledge models is actually about reusable knowledge models. Researchers are interested in generic tasks, generic methods, and generic domain models. In addition, the reverse implication also applies, i.e. reuse implies modelling. In the words of Krueger (1992) "all approaches to software reuse use some form of abstraction for software artefacts. Abstraction is the essential feature in any reuse technique. Without abstractions, software developers would be forced to sift through a collection of reusable artefacts trying to figure out what each artefact did, when it could be reused, and how to reuse it". Thus, a modelling approach to knowledge engineering is essentially the same as a reuse-centred approach. Both are about identifying, formalizing, specialising, and integrating reusable, abstract models.

1.5. FROM BATTLEGROUND TO BACKGROUND

The discussion so far has identified the 'battleground' of this thesis. It has defined the subject of enquiry (knowledge-based systems), the paradigm (modelling) and the pragmatics of the exercise (reuse). In the course of this thesis I will instantiate the generic modelling paradigm presented here into a particular approach and propose solutions for constructing and representing application models and model components. First, however, I will review existing approaches to knowledge modelling and illustrate how the main tenets of my approach emerge as answers to some of the problems with current proposals. This review is presented in the next chapter.

Chapter 2
Approaches to Knowledge Modelling

This chapter reviews current approaches to knowledge modelling. The review consists of four parts. In the first one I will discuss the most significant frameworks for knowledge modelling which have been proposed in the literature. The second part reviews current libraries of reusable model components. The third part focuses on problem solving methods and discusses recent work which attempts to provide methodological and theoretical foundations to the notion of reusable problem solving method. Finally, in the concluding section I will highlight some issues emerging from the review, which will be addressed in this thesis.

2.1. INTRODUCTION

In this chapter I will review current approaches to knowledge modelling. In contrast with some other recent reviews of the field (Brazier and Wijngaards, 1997) I do not aim to cover all facets of a particular approach. In particular aspects such as modelling tools and languages will not be discussed. The reason for this choice is that I wish to focus on the *epistemology* of the various approaches, i.e. on modelling frameworks. Specifically, I will evaluate alternative approaches in terms of the support they provide for knowledge analysis and for library and application development. Moreover, because the thesis is mainly about problem solving components, I will look in particular detail at problem solving methods and discuss recent work which tries to provide methodological and theoretical foundations to the notion of reusable problem solving method. Finally, in the concluding section I will highlight some issues emerging from the review, which will be addressed in this thesis.

2.2. AN OVERVIEW OF KNOWLEDGE MODELLING FRAMEWORKS

A knowledge modelling framework defines the basic organization of an approach to knowledge modelling. It distinguishes the basic types of modelling components, their relations, and proposes a model development methodology. Typically, each modelling framework is also associated with a number of support tools and modelling languages. However, these are not needed when discussing the epistemology of a framework: most modelling approaches are nowadays characterized in a tool-independent manner

(Schreiber et al., 1994b; Shadbolt et al., 1993; Chandrasekaran et al., 1992; Steels, 1990)[1].

A modelling framework will be described in terms of three main aspects.

- **Types of Components**. A description of the different types of generic model structures proposed by each approach - e.g. what kinds of distinctions an approach proposes to discriminate between generic, domain, and application specific knowledge.
- **Relations between Components**. A description of the main structuring relations between generic modelling components. In particular different approaches take different views over the so-called *interaction hypothesis* (Bylander and Chandrasekaran, 1988). This states that both the nature of the domain knowledge required by an application and its representation are strongly determined by the chosen task and/or method.
- **Model Development Methodology**. A description, drawn either from empirical evidence or published guidelines, of the model development methodology associated with a modelling approach.

To be useful, an evaluation process has to be task-centred; accordingly, this review will focus on the following two activities:

- **Knowledge Analysis**. How well does the framework support knowledge analysis? Does it identify distinctions which are relevant when analysing a domain or an application? Does it provide the right level of granularity?
- **Reuse-centred Model Development**. How good an *integration framework* (Krueger, 1992) does an approach provide? Does its epistemology provide distinctions which are useful for knowledge sharing and reuse?

In what follows I will discuss seven knowledge modelling frameworks: *KADS/Common KADS*, *Components of Expertise*, *Generic Tasks*, *Role-limiting Methods*, *Protégé*, *DIDS* and *Spark/Burn/Firefighter*. The rationale for selecting these particular approaches was to have a range of different 'modelling philosophies' while at the same time including all the most influential approaches.

2.2.1. KADS/Common KADS

The KADS approach to knowledge modelling has been formulated over more than a decade (1983-1994) during a number of large, collaborative research projects. For this

---

[1] This statement does not necessarily apply to older approaches to knowledge acquisition, whose epistemology has often to be abstracted from the relevant tool.

reason the ideas underlying KADS have evolved over the years in a more or less continuous fashion and (of course) are still evolving. Nevertheless, it is possible and useful to distinguish between two main phases of research, which correspond to the life spans of the two EC-funded research projects on KADS. The first project (1983 - 1989) led to the formulation of the KADS methodology for KBS development, which is described in (Wielinga et al., 1992a). The second project (1990 - 1994) developed the KADS framework further, in particular by integrating it with other modelling approaches, such as *Components of Expertise* (Steels, 1990). The revised methodology - called Common KADS - is described in (Schreiber et al., 1994b; Van de Velde, 1994; Wielinga et al., 1992b) and is the one which will be reviewed here.

2.2.1.1. Types of Components

The Common KADS framework distinguishes between three categories of knowledge which are "necessary and sufficient for the description of application-related knowledge" (Wielinga et al., 1992b): *task*, *inference*, and *domain.*

Task knowledge describes both *what* problem solving needs to be carried out in the domain and *how* (at the knowledge-level). Therefore a task definition includes the specification of the input, output and goal of a task, as well as its task-subtask decomposition and control.

Inference knowledge describes primitive reasoning steps, which specify the 'leaves' of task-subtask hierarchies. A reasoning step is considered primitive if it can be described functionally, i.e. if its internal control structure is not relevant to a knowledge-level analysis. Whether a step is primitive or not is a pragmatic decision of the knowledge analyst.

Domain knowledge "specifies form, structure, and contents of domain specific knowledge that is relevant for an application" (Van de Velde, 1994) - i.e. it specifies both the application domain knowledge and the domain ontology.

Thus, Common KADS provides three basic views over an application, which are similar to the data, functional and control perspectives in software engineering. In addition, the Common KADS framework also considers two generic components, *problem solving methods* and *domain ontologies.* The former abstract from application-specific control and inference views and describe generic ways of "satisfying a class of task definitions" (Wielinga et al., 1992b). The latter describes a "particular viewpoint on application specific knowledge" (Van de Velde, 1994) which makes it possible - for example - to use domain knowledge with a particular problem solving method. For instance, methods such as *Cover&Differentiate* (Eshelman, 1988) require causal models of the domain. These can be defined generically by means of an ontology of causal relations which can then be used to provide a particular viewpoint, often called a *domain model* (Steels,

1990), over application-specific knowledge.

2.2.1.2. Relations between components

The relations between components in Common KADS are defined by means of *knowledge roles*. A knowledge role is defined as "an abstract label that indicates the role that domain knowledge to which the label is attached, plays in an inference process" (Aben, 1994 - p. 91). For instance, the role 'hypothesis' provides a 'handle' at the inference layer to refer to domain structures which could (statically or dynamically) play the role of a hypothesis during the reasoning process.

This characterization of roles as metalevel labels for domain concepts was introduced in the KADS-1 framework and reflected the assumption that only a loose coupling was needed between domain and problem solving knowledge. This approach was later revised in Common KADS, which recognises that there are varying degrees of task-domain coupling, depending on the application - "a good knowledge modelling methodology should span the whole continuum, from weak to strong knowledge interaction" (Wielinga et al., 1992b). This flexibility is (presumably) reflected in the use of more or less task-dependent domain models. Hence, application configuration is characterized as a process of building a mapping between the knowledge roles required by the chosen task model and the *application ontology* which specifies the form of the domain knowledge base. Because different viewpoints are possible over a domain, the application ontology will normally integrate a number of different domain ontologies.

Incidentally, the KADS view of roles as meta-level labels has been criticised by some researchers who point out that this approach does not provide enough 'structure' for describing the properties of task-level concepts - e.g. for characterising the notion of hypothesis (Causse, 1993).

2.2.1.3. Model Building Process

In earlier papers (Wielinga and Breuker, 1986), interpretation models were produced in a bottom-up way, by applying knowledge acquisition techniques such as protocol analysis to build generalised models of expert problem solving behaviour. During the course of the KADS-1 project, a library of 'ready-made' interpretation models was developed (Breuker et al., 1987) with the aim of using these as templates for driving the knowledge acquisition process. In practice however, it turns out that it is very rare for an interpretation model to fit 'perfectly' an application and therefore the model development process was later characterised either as a process of modifying and adapting a pre-existing model (Schreiber, 1994), or as a *task analysis* process (Schreiber and Terpstra, 1996), carried out along the lines suggested by Steels (1990).

The main problem with task analysis and *ad hoc* customizations is that these are more of an art than a set of communicable techniques. Therefore, researchers have tried to put

some sort of methodological structure around this model configuration and adaptation process. The approach taken in the Common KADS library (Breuker and Van de Velde, 1994) addresses this problem by constructing a library which contains not just complete models but also modelling components and *modelling operators*. The latter describe how "knowledge engineers build their model" (Valente et al., 1994). Thus, the idea here is to store not just model components but also the 'rules of compositions', so that a structure can be imposed on the model development process. A similar approach has been adopted in the Acknowledge (Anjewierden et al., 1992) and VITAL (Shadbolt et al., 1993; Domingue et al., 1993) projects, where the model development process consists of a sequence of decomposition steps, in which, at each stage of the process, domain knowledge is used to choose one of the available decompositions. Examples of this approach to the development of task models can be found in (Motta et al., 1994a; 1996).

2.2.1.4. Evaluation of the approach

The main strength of Common KADS is the fact that it provides a comprehensive framework, addressing all the various aspects related to knowledge modelling - i.e. epistemology, languages, reuse, library organization, tools, methodology. Moreover, because it has evolved over many years and has been adopted by several research and industrial groups, it has developed into a very flexible framework, able to 'accommodate' different approaches to modelling. Therefore, while one can of course still point to a specific, fully instantiated modelling framework, it is also the case that KADS is now also a modelling philosophy, which different researchers have 'bent' in different ways. In a sense one can say that the role of KADS today is similar to the role of St Thomas theology in the middle ages: it not only provides answers to questions of doctrine, but it also provides the framework and the language in which further questions can be asked and different approaches can be pursued.

Nevertheless, if we stick to 'orthodox KADS' a number of (arguably minor) difficulties arise if we evaluate it from a reuse-centred viewpoint.

Common KADS proposes a clear-cut distinction between application-specific knowledge and generic models. In practice I have found useful to distinguish not just between application knowledge and generic models, but also between different types of domain knowledge. In particular, an important distinction is the one between *multi-functional* (Murray and Porter, 1988) and application-specific knowledge. This distinction is useful for knowledge analysis because it permits the separation of the application-specific elements of a knowledge base from those which are domain-specific but not application-specific. Moreover, it is especially important for reuse as domain-specific knowledge can be reused in different applications which share the same domain model. For instance, the VT domain ontology developed by Gruber et al. (1996) provides an example of multi-functional domain model. However, as discussed in (Motta and Zdrahal, 1995),

additional, application-specific knowledge is needed to reuse this domain model with a *Propose&Revise* problem solver (Marcus and McDermott, 1989; Zdrahal and Motta, 1995; Chapter 8).

Another problem which has already been mentioned is the view of roles as labels. This problem can be reformulated, as Guarino (1997) does, by highlighting the absence in Common KADS of a notion such as *method ontology* (Gennari et al., 1994), which specifies the ontological commitments associated with a problem solving method. These commitments would have to be defined in Common KADS as the schema of a particular domain viewpoint. However, a domain viewpoint necessarily integrates both the ontological requirements imposed by the application knowledge and those imposed by the problem solving method. Therefore the resulting ontology would be less reusable than one associated only with the method viewpoint[2].

Finally, the basic distinction used in Common KADS between tasks, inferences, and domain does not identify distinct foci for reuse. It provides different views over an application, much like the software engineering distinction between functional and control views. However, from a reuse point of view the distinction between task decomposition and inference structure is not very important: a task decomposition uniquely determines an inference structure. The limited reuse value of the task/inference distinction is also highlighted by the fact that the more general notion of *function* is used in the Common KADS library (Valente et al., 1994), which subsumes both tasks and inferences. The reason for introducing the notion of function is that it is only at the end of a model construction process that a knowledge-flow component can be identified as a task or an inference. Thus, whether a problem solving component specifies a task or an inference is not an essential feature of the component in question but a modelling decision taken in the context of a complete expertise model.

2.2.2. Components of Expertise

The *Components of Expertise* (CoE) approach was proposed by Luc Steels (1990), with the aim of providing a comprehensive modelling framework which could subsume and integrate a number of approaches outlined in the modelling literature of the eighties (McDermott, 1988; Clancey, 1985; Breuker and Wielinga, 1989; Bylander and Chandrasekaran, 1988).

---

[2] Incidentally, the Common KADS solution to the VT problem developed by Schreiber. and Terpstra (1996) makes use of notions such as 'task ontology' and 'method ontology'. This is an example of the aforementioned flexibility of the Common KADS modelling framework.

2.2.2.1. Types of Components

Steels' approach distinguishes three basic components of expertise: tasks, methods, and domains. Tasks specify what needs to be solved; methods specify how to solve them; and domain models specify viewpoints imposed by task models over an application domain. This approach was later integrated in the Common KADS methodology.

2.2.2.2. Relations between components

For a given task there are in general a number of methods which can be used to solve it. In particular Steels introduces the distinction between the *conceptual* and *pragmatic features* of a method. The former specify what a method can do - i.e. its competence; the latter make it possible to distinguish between different methods applicable to the same task. For instance a diagnostic method which requires carrying out some particular diagnostic test would not be appropriate in domains in which the cost of these tests is too high.

2.2.2.3. Model Building Process

A model is developed through a task analysis process, in which a task model of the application is built by starting with a task defining a *problem type* - e.g. diagnosis - and then using a task-method decomposition tree to select the best method for a particular task, until a complete task model for the application has been developed.

2.2.2.4. Evaluation of the approach

The CoE framework introduced a number of ideas which were later taken on by several approaches to knowledge modelling: e.g. the notion of multiple domain models for an application, the task-to-method framework for library organization, and the importance of pragmatic aspects when deciding on the suitability of a method for a particular task. Moreover, in contrast with the domain/inference/task decomposition used by KADS, the three components of expertise proposed by Steels are truly orthogonal dimensions for reuse and can be used for organizing a library of reusable components for knowledge modelling.

On the other hand, as is the case with Common KADS, Steels' framework does not address the distinction between domain knowledge and application knowledge. Its domain dimension is based on generic domain models, which fulfil different reasoning roles, as imposed by task models. So, the framework is very much method-oriented: there are no multi-functional domain models. Thus, Steels subscribes to Chandrasekaran's view of a use-oriented characterization of domain knowledge - see section 2.2.3. Moreover, application-specific knowledge is only weakly characterised as 'heuristic annotations'. This characterization fails to address the fact that application knowledge comes in different types, as mapping knowledge and as application-specific heuristic knowledge - see next chapter.

Finally, Steels' framework and methodology are rather software-oriented. The task structure framework provides a kind of high level perspective on KBS design but it is not really a level of description separated from the implementation level. Typically both primitive methods and domain entities are directly implemented in Lisp. Therefore the knowledge level only provides a task-centred view over the symbol level: it is not a complete, distinct level, as for instance in Common KADS.

2.2.2.5. Conclusions

The CoE framework is historically important because it introduced a number of ideas which influenced much later knowledge modelling research. However, it has a number of limitations: it does not characterize domain knowledge independently of a task model and is rather implementation-oriented, in the sense that only implementation-oriented descriptions of components are provided. Thus, it provides only limited support for knowledge analysis and it is probably better viewed as a tool for KBS design.

2.2.3. Generic Tasks

The *Generic Tasks* (GT) approach was developed over a number of years by Chandrasekaran's group (Chandrasekaran, 1983; Bylander and Chandrasekaran, 1988; Chandrasekaran et al., 1992). As the name suggests, it centres around the notion of generic task. Originally, this denoted both a task (i.e. a goal specification) and a method which could be used to accomplish it; later it developed into the notion of *generic task structure*. This organizes a model of problem solving in terms of a task-subtask decomposition mediated by the application of problem solving methods. Here I will be commenting on the most recent formulation of the generic task approach (Chandrasekaran et al., 1992).

2.2.3.1. Types of Components

The GT approach is an example of the use-oriented view of knowledge which - often characterised as the procedural approach to knowledge representation - has a long and illustrious tradition in AI (Hewitt, 1971). A use-oriented view states that both the nature and the representation of domain knowledge is determined by the particular task (or method) which is carried out. In other words there is no characterization of knowledge which is independent of its use. Therefore, only tasks and methods are considered in the GT approach. A task is specified in terms of a "problem/goal pair", essentially a relation between an input specification and a goal. Methods are characterised in terms of the *problem space computational model* (Newell, 1980) as a set of subtasks which transform the initial state into a goal state. *Search control knowledge* is defined as knowledge which controls the order in which subtasks are solved. *Flexible problem solving* is supported by means of method selection knowledge, which is defined as knowledge which allows an analyst (at design time) or a problem solving system (at run time) to

choose the best method for a task.

2.2.3.2. *Relations between components*

In accordance with the use-oriented approach to modelling only tasks and methods are considered. Methods solve tasks either directly or by introducing new subtasks which are then in turn solved by other methods. The problem space model is used to give a computational meaning to a task structure.

2.2.3.3. *Model Building Process*

The model building process consists of developing a task model for an application by selecting the best method for each task and subtask until all subtasks are solved by *direct methods* (these are methods which solve a task directly, without introducing a task-subtask decomposition), much as in the Components of Expertise approach. Method selection can be carried out either dynamically as in the TIPS architecture (Punch, 1989), or statically.

2.2.3.4. *Evaluation of the approach*

Chandrasekaran's framework is the most complete formulation of the use-oriented view of knowledge modelling. More or less at the same time as Steels' development of the CoE approach, he introduced the notion of task structure as a way to organize flexible libraries of problem solving methods and to support flexible problem solving. Moreover, in collaboration with a number of researchers, he developed several problem solving methods for design, diagnosis, and classification which contributed significantly to the progress of research in problem solving methods.

An obvious limitation of the GT approach is that it does not consider task-independent domain models, thus limiting the possibilities for reuse. Moreover, its main epistemological distinction, between task and search control knowledge, has only a limited value from the point of view of knowledge analysis. This distinction assumes that application knowledge has already been digested in the relevant computational categories. Hence, such distinction should be the target rather than the starting point of an analysis process. Thus, it does not offer a good framework for analysing a domain; its main purpose is to provide a computational semantics to a knowledge model (Laird et al., 1987; Smith and Johnson, 1993).

2.2.4. Role-limiting Methods

The approach based on *role-limiting methods* (McDermott, 1988) was developed during the eighties by a group of researchers at Carnegie-Mellon university, led by John McDermott. The basic philosophy of this work stems from the role differentiation principle (see section 1.3.3): it is possible to identify generic problem solving models, which impose specific *problem solving roles* on domain knowledge. These models can

be used to support knowledge acquisition and to produce robust systems based on reusable problem solving methods. The book edited by Marcus (1988) contains a number of papers describing the philosophy of the approach and five knowledge acquisition tools which are based on different problem solving methods.

2.2.4.1. Types of Components

Like the Generic Task approach, the approach based on role-limiting methods subscribes to the use-oriented modelling paradigm. Domain knowledge is encoded in method specific terms and there is a simple mapping between a problem solving method and a class of tasks for which the method is suitable. Moreover, there is no flexibility in the specification of role-limiting methods: their interface is given solely by their domain roles.

2.2.4.2. Relations between components

There is basically only one component: a complete problem solving method. This is instantiated by filling its knowledge roles in terms of domain knowledge.

2.2.4.3. Model Building Process

This is carried out as a role-filling process which is typically driven by an associated knowledge acquisition tool. The control regime is fixed for each method.

2.2.4.4. Evaluation of the approach

This approach was historically important because it put into practice the role differentiation principle uncovered by Clancey's analysis of first generation rule-based systems. Moreover on the practical side it led to the development of a number of problem solving methods and associated knowledge acquisition tools, which have been a significant contribution to knowledge engineering research. Obviously, in comparisons with the more sophisticated and comprehensive modelling frameworks developed during the nineties, the role-limited approach looks quite basic. In particular, a number of approaches (van Heijst et al., 1992; Chandrasekaran et al., 1992; Steels, 1990; Puerta et al., 1992) can be seen as proposals to overcome the limits of role limiting methods and develop flexible, configurable problem solving models.

2.2.5. Protégé

The Protégé approach has been developed by Mark Musen and his colleagues of the Medical Informatics Laboratory at Stanford University. This work began with the Opal knowledge acquisition tool (Musen et al., 1987), which enabled domain experts to enter the cancer treatment plans used by the Oncocin system (Shortliffe et al., 1981) and receive monitoring and advice on cancer treatment. The Opal system had only limited, domain-specific scope: it had been designed to acquire chemotherapy plans for cancer treatment. To overcome these limitations Musen developed the Protégé tool (Musen, 1989). Much like the tools based on role-limiting methods, Protégé generalises from the

specific domain tackled by Opal and supports the development of task models for applications which can be solved by means of the *Episodic Skeletal Plan Refinement* problem solving method (Tu et al., 1992). Needless to say users of Protégé then encountered the same problems as those who tried role-limiting KA tools. A task model for an application rarely matches a given problem solving method, which means that users require facilities for developing flexible, configurable problem solving methods. These problems led to the development of the Protégé-II architecture (Puerta et al., 1992; Gennari et al., 1994). This supports the configuration of a task model from a library of methods and mechanisms, and the generation of task-specific knowledge acquisition tools. In the rest of this section I will discuss the modelling framework underlying Protégé-II.

2.2.5.1. *Types of Components*

While the original specification of Protégé-II appears to subscribe to a use-oriented view of domain knowledge (Puerta et al., 1992), the most recent formulation of the Protégé-II approach (Gennari et al., 1994) recognises tasks, methods, and domains as three distinct foci for reuse. However tasks are only described informally, in terms of their inputs and outputs. Much of the action in Protégé-II is in the specification of methods and *mechanisms*. The former decompose tasks into subtasks; the latter are direct methods. Methods are described in terms of their ontological requirements, input-output relation, control and data flow.

2.2.5.2. *Relations between components*

In order to maximise reuse both method-independent domain models and domain-independent methods can be specified independently and integrated by means of an *application ontology* (Gennari et al., 1994). This integrates a method and a domain ontology by introducing the required *mapping relations*.

2.2.5.3. *Model Building Process*

Model development is driven by the library of methods and mechanisms. These are indexed by tasks and the model building process consists of generating a task model through a method-driven task decomposition process, much like in the Generic Tasks approach. This task model can then be used to acquire the relevant domain knowledge. If this already exists in a method-independent form, then the relevant mappings need to be performed. The entire process is supported by a suite of knowledge acquisition tools, which are customised in terms of the application ontology.

2.2.5.4. *Evaluation of the approach*

Protégé-II provides the most complete formulation of the task-to-method approach to knowledge modelling. It provides a library of tasks and methods as well as tools to support the selection and configuration of methods. The latter is characterized as a

process of ontology mapping. The approach I'm going to use to build a library of methods for design subscribes to much of the Protégé-II framework, in particular the use of ontologies to express method requirements and the notion of ontology mapping to bridge the gap between domain and methods. The main difference between Protégé-II and my approach is that I wish to give a clear theoretical basis to my library of methods, rather than just cluster them in terms of the relevant tasks. Therefore, as discussed in the next chapter (and, more in depth, in chapter 8), the relation between problem solving methods and a high level task (i.e. a problem type) is mediated in my approach by the construction of a task-specific, but method-independent problem solving model.

2.2.6. DIDS

The DIDS workbench provides a number of knowledge acquisition tools and mechanisms supporting the development of design applications through reusable components. Here I will review the modelling philosophy underlying the DIDS approach, as illustrated in (Runkel et al., 1994).

2.2.6.1. Types of Components

The main goal of the DIDS project was to produce a set of reusable mechanisms which could be used to build design applications. Therefore, the DIDS framework is essentially task-centred, there is no notion of task-independent domain knowledge. Moreover, the problem solving components, called mechanisms, are also task-specific. Given this task-oriented approach, the DIDS researchers have identified the typical problem solving actions which are carried out during configuration design problem solving and produced a library of these (Balkany et al., 1993). Thus, an important aspect of this work is that these problem solving components are reusable.

The approach to reuse adopted by the DIDS researchers centres on the distinction between *task* and *search-control* knowledge. This distinction is grounded on the problem space computational model. Task knowledge defines what the task is about (i.e. the search space) and search control knowledge defines how to navigate this search space efficiently. The main aspect of task knowledge is that it does not make any commitment to a particular problem solving method. Incidentally, this characterization of search-control knowledge is different from the one used by Chandrasekaran et al., (1992), who use this term to refer to knowledge about ordering subtasks. Here, the problem space is not defined by the generic (i.e. problem-type-independent) task structure, as in (Chandrasekaran et al., 1992), but it is instantiated in terms of all possible states in configuration design problem solving.

2.2.6.2. Relations between components

Because the framework is specific to configuration design and because all components are expressed in terms specific to configuration design there is no relation between

components to speak of.

2.2.6.3. Model Building Process

This consists of building a task model out of a subset of the existing problem solving components and then acquiring the relevant domain knowledge through knowledge acquisition mechanisms (MeKAs). Like in Protégé-II these are forms which are automatically constructed from the knowledge requirements of the task model.

2.2.6.4. Evaluation of the approach

Like other approaches discussed here (Generic Tasks, Role-limiting Methods) DIDS subscribes to a use-centred view of knowledge modelling. Therefore, it does not offer any provision for reuse of task-independent, domain knowledge bases. However, in contrast with these other use-centred approaches, there is also no notion of applying methods to tasks. There is only one type of task - configuration design. Moreover, in contrast with the GT approach, developing a problem solver does not involve the construction of a task model: a problem solver is constructed by assembling existing mechanisms from a library. This - in my view - is an important step forward because it replaces the relatively unstructured process of task model construction with a relatively structured one of configuring a problem solver out of pre-existing components.

Finally, as in the case of the GT approach, the basic modelling distinction enforced by DIDS is the one between task and search-control knowledge. As I have already pointed out when evaluating the GT approach, such a distinction is of only limited use when carrying out knowledge analysis.

2.2.7. Spark/Burn/Firefighter

2.2.7.1. Overview

As discussed above, the problems with knowledge acquisition architectures based on complete problem solving methods led to the development of flexible modelling frameworks, which supported the configuration of task models in terms of reusable *mechanisms*. A similar approach was used by McDermott's group in their Spark/Burn/Firefighter (SBF) project (Klinker et al., 1991; 1993), where they developed a set of tools supporting workplace analysis and automation. In particular, this second task was carried out by i) identifying an activity in the workplace which was amenable to automation; ii) choosing a suitable software mechanism from a library; and iii) integrating this in the workplace model. This integration step is supported by the Burn tool, which i) maps the data model associated with a mechanism's I/O specification to the data model of the overall task model, ii) acquires the relevant knowledge by means of mechanism-specific KA tools, and iii) converts the resulting software module into an agent which can be monitored and managed by the workplace controller (Firefighter).

What differentiates the SBF approach from almost[3] any other we have seen in this review is the emphasis on workplace analysis and on "taking integration seriously" (Klinker et al., 1993). The SBF researchers have not only tackled 'typical' modelling issues, such as the identification of the appropriate reusable mechanisms (Klinker et al., 1991), but also the complex integration issues associated with the deployment of KBS technology into the workplace - e.g. the issues related to the integration of human and software agents. Thus, the research agenda addressed by the SBF approach is much more ambitious than that of any of the other approaches discussed here. However, because in this thesis I'm not addressing such 'additional' issues concerning workplace analysis and the integration of human and software agents[4] I will evaluate the SBF approach only with respect to modelling issues.

2.2.7.2. Evaluation

From a modelling point of view the SBF approach is very similar to DIDS. Like DIDS, it centres on the identification of reusable mechanisms which can be assembled and configured to produce a task model. The main difference between DIDS and SBF is one of methodology. The DIDS researchers focused on design problems and identified the set of mechanisms which comprise the DIDS library by means of a bottom-up approach, i.e. by analysing the mechanisms used by a number of design tools (Balkany et al., 1993). In contrast with DIDS, the SBF approach is generic - i.e. it is not just confined to a class of tasks. The mechanisms are created on demand, when a particular activity is identified which is amenable to automation and no mechanism in the library is suitable for the job. However, given that the SBF approach is both application-independent and targeted to non-programmers, it is difficult to identify the right mechanisms to add to the library. Mechanisms which are too generic are not easy to use, while mechanisms which are too specific are not very reusable. This trade-off between usability and reusability is addressed in Klinker et al. (1991), where the authors discuss a number of guidelines

---

[3] The exception here is Common KADS which, in addition to the expertise model, also includes an organization model. This corresponds to the workplace model considered in SBF. The main difference between the Common KADS approach and SBF is that an organization model in Common KADS is 'just a model' - i.e. it is not operational. In contrast with Common KADS, the SBF researchers have tried to provide a set of tools to support both application development and its integration in the workplace.

[4] Of course this sentence should not be understood as implying that I am not interested in these issues. On the contrary much of my research in the early nineties focused on the interoperability and integration issues associated with architectures which integrate software and human agents (Gaspari et al., 1993; 1995; Gaspari and Motta, 1994).

which they have used to determine the set of mechanisms to include in the SBF library. These guidelines are based on heuristics such as "Define mechanisms so that only 1 or 2 configurations will cover most of the significantly different computational alternatives for an activity". While these guidelines have their utility, they are typically too weak to provide a robust methodology for building the SBF library. Indeed, the main problem with the SBF approach is that it is too ambitious. It attempts to tackle modelling, KA, and integration issues at the same time and in a problem-independent scenario. As a result, it does not solve any of these issues in a satisfactory way. In particular, from a modelling point of view the library of mechanisms is very much ad hoc. In contrast with this approach, my aim is to build a library of reusable components which has a clear theoretical basis.

2.2.8. Summing-up

In conclusion, as one would expect when carrying out reviews of a field, different approaches have different strengths and weaknesses. Common KADS provides a very comprehensive and flexible framework which has benefited from contributions produced by many people, at different sites, over a number of projects. Approaches such as Components of Expertise, Generic Tasks, and Role-limiting Methods are very important from a historical point of view. However, they have now been subsumed by more modern approaches, such as Common KADS and Protégé-II, which consider different degrees of interactivity between methods and domain knowledge, and use role mappings and application ontologies to bridge the gap between methods and domain in a flexible, application-dependent way. Finally, the DIDS framework is limited to configuration design tasks, but it introduces an important approach to library organization, which is based on the use of a method-independent, but task-specific model. In the next chapter I will discuss my framework to knowledge modelling, in particular illustrating how it subsumes and integrates the approaches described in this section.

First, I will take a closer look at libraries of reusable model components.

2.3. APPROACHES TO LIBRARY ORGANIZATION

Existing libraries of reusable components for knowledge modelling can be divided into three classes: generic libraries, libraries of domain ontologies, and libraries of problem solving methods.

2.3.1. The Common KADS library: a general-purpose library for knowledge modelling

The most comprehensive library of generic model components is the one produced as a result of the Common KADS project (Breuker and Van de Velde, 1994). It consists of

three main classes of library components: *modelling components*, *modelling operators*, and *generic models*. Generic models are "complete expertise models" (Valente et al., 1994); modelling components are elements of expertise models and modelling operators are relations between generic models. These specify a possible transformation of a generic model. Modelling operators are included in the library to ensure that not only the results, but also the model building steps involved in a model construction exercise are captured by the library.

The Common KADS approach to library organization encompasses a number of modelling approaches and is therefore very generic. A generic model can be anything from a KADS-1 interpretation model to a role-limiting method. Moreover, various indexing mechanisms are supported, which organize the library in terms of classes of generic models, taxonomies of modelling components, and classes of *element features*. These specify compatibility relations between model components, thus providing a way of partitioning the library into classes of components compatible with each other and with the current modelling context (Valente et al., 1994).

The generality of the approach taken in the Common KADS library essentially defines both the strengths and weaknesses of the Common KADS approach. It makes it possible to account for different approaches to modelling and to library organization but necessarily it only provides fairly weak principles for structuring a library.

2.3.2. Libraries of ontologies

Another type of library is one which contains domain ontologies (Farquhar et al., 1996; Falasconi and Stefanelli, 1994). The most comprehensive and best known of these is the Ontolingua library (Farquhar et al., 1996). Typically libraries of domain ontologies are organized in terms of an inclusion hierarchy - each ontology is built on top of a number of sub-ontologies. The main issues associated with research on domain ontologies have to do with characterizing their nature - what exactly is an ontology - and with the ontology development methodology - what terms go into an ontology; what guidelines are used to drive the ontology construction process (Gruber, 1995). In this thesis I will focus mainly on components of problem solving methods (*problem solving components*) and therefore I will not discuss the issues associated with the development of libraries of domain ontologies. Nonetheless, I will use the notion of ontology both as part of my modelling framework and as part of the library of parametric design components. Therefore, in the next chapter, I will come back to the notion of ontology, in particular illustrating i) my view on what constitutes an ontology, ii) the types of ontologies included in my modelling framework, and iii) the specific ontologies included in the parametric design library. Now, I will turn my attention to libraries of problem solving components.

2.4. PROBLEM SOLVING METHODS: ORGANIZATION AND DEVELOPMENT

The main objective of this thesis is to produce a library of reusable problem solving components for parametric design applications. However building a library is not just a problem of identifying and indexing reusable components. In particular a library of problem solving components presupposes a view on the nature of problem solving methods, the relation between methods and task specifications, and the method development process. Therefore in this section I will look not only at existing libraries of problem solving components, but I will also review the work of those researchers who have tried to characterize the nature of problem solving methods and the method development process.

2.4.1. Characterizing and developing problem solving methods.

An approach to characterizing formally the process by which a method is developed from a task specification is illustrated in (Akkermans et al., 1993; Wielinga et al., 1995). This approach is based on the idea that a formal specification of a problem solving method can be derived through a process of stepwise refinement of the *competence theory* (Van de Velde, 1988) associated with a task specification. This refinement process involves both generic operationalization steps, such as expanding a set notation into quantified expressions and substituting provability for truth, and knowledge-intensive ones. These include the selection of a problem solving paradigm for refining the initial, high-level task specification and the introduction of domain relations for representing the available domain knowledge. In particular, the example described by Wielinga et al. attempts to derive a formal description of a Propose&Revise method from the specification of the class of parametric design tasks. The paradigm they select as *trait d'union* between the task and method specification is *Generate&Test*.

This approach provides two important contributions to research in problem solving methods. It describes a formal framework for characterizing PSM development, which goes beyond simple guidelines such as those used in informal approaches (Benjamins, 1993). Moreover, Wielinga et al. clearly show that this method development process mainly consists of introducing *ontological commitments*, which are related to the availability and to the properties of application knowledge.

The idea of PSM development as an assumption-driven process has been further pursued by Dieter Fensel and his colleagues (Fensel and Straatman, 1996; Fensel and Schonegge, 1997a), who try to characterise more precisely the method specification and development process sketched by Wielinga et al. and to semi-automate it by means of theorem proving techniques. In particular, Fensel and Straatman (1996) propose a more structured framework than the one used by Wielinga et al. and characterise a PSM specification in

terms of four parts: a *functional specification*, a *cost description*, an *operational specification* and a list of *assumptions*. From a theorem-proving point of view the role of the assumptions associated with a PSM is to provide the "missing pieces in the proof that the behaviour of a method satisfies its goal" (Fensel and Straatman, 1996). As a result, the acquisition of these assumptions can then be automated by identifying the reasons which prevent the operational specification of a PSM from satisfying its goals (Fensel and Schonegge, 1997a).

In my view the main strength of this kind of approach is that it provides a formal basis to PSM specification and opens the way to automatic verification of problem solving methods (Fensel and Schonegge, 1997b). However, I am not sure that this approach succeeds in clarifying the nature of problem solving methods, in particular the nature of the knowledge structures required by a PSM. For instance, both Wielinga et al. (1995) and Fensel and Straatman (1996) illustrate a method development process by which a Propose&Revise problem solver is derived from a Generate&Test. However, the reconstruction appears to be quite artificial. Knowledge structures, such as propose and revise knowledge, are introduced, but it is not clear where they originate from and how they relate to other method and task concepts - e.g. what is the relation between propose and revise knowledge and the specification of a parametric design task.

Therefore, in this thesis I will propose an alternative approach, which makes use of the search paradigm as a way to bridge the gap between task specification and problem solving methods and to give 'computational semantics' to the knowledge structures required by a problem solving method.

2.4.2. Approaches to the organization of libraries of problem solving methods

The earlier libraries of problem solving components consisted of complete methods (Marcus, 1988; Breuker et al., 1987) and as a result came unstuck very quickly: in general a problem solving method needs to be adapted and configured for each particular application. For this reason, researchers tried to make problem solving methods more flexible by representing them by means of task decomposition hierarchies (Steels, 1990; Chandrasekaran et al., 1992; Puerta et al., 1992; Van Heijst et al., 1992; O'Hara, 1993, 1995). While the details of the various approaches differ in various respects, the underlying idea was essentially the same: constructing a problem solving method for a specific application - the resulting model is often called a *task model* - consists of recursively navigating a task-method decomposition tree, and at each stage selecting one of a number of possible methods applicable to the current task. This selection can be done at run-time - for achieving *flexible problem solving* - or during the design phase. This process is driven by the appropriate method selection knowledge, which can take the

form of *library features* (Breuker and Van de Velde, 1994), *pragmatic constraints* (Steels, 1990), or *method assumptions* (Benjamins and Pierret-Golbreich, 1996).

Task-method structures provide both an economic (in terms of the ratio elements/models) and flexible way of organizing a library of problem solving methods and have been used in areas such as design (Chandrasekaran, 1990) and diagnosis (Benjamins, 1993). Moreover, as shown by the work on *Generalized Directive Models (GDM)* (van Heijst et al., 1992; O'Hara, 1993, 1995) it is possible to build knowledge acquisition tools (Anjewierden et al., 1992) which use these task decomposition structures[5] to support a flexible style of model-driven knowledge acquisition. In the GDM approach a task model is built incrementally, by intertwining model decomposition steps with elicitation sessions: at each stage of the process, domain knowledge is used to choose one of the available decompositions. Examples of this approach to the development of task models can be found in (Motta et al., 1994a, 1996).

While task-method structures provide more flexibility than the previous generation of complete methods, there are nonetheless problems with them. These are discussed in the next sections.

2.4.2.1. Difficulties with local method selection knowledge

Task-method structures rely on local guidelines/rules to drive the method selection process. However, the complexity of these rules vary significantly. For instance the task-method structure used by Benjamins (1993) includes simple questions such as "Is the user able to recognise symptoms?". In other cases these questions may not be answerable at all, e.g. "Are the hypotheses in the hypothesis set unrelated to each other?". These examples provide instances of a general phenomenon: questions about domain features tend to be more difficult than questions about the availability of knowledge. The problem is compounded by the existential predicament of knowledge-based systems: these are by definition systems that take decisions under uncertain conditions. Therefore, no amount of method selection knowledge in a task-method structure can avoid the fact that AI and knowledge engineering have a strong experimental connotation: often the

---

[5] To be precise the GDM approach only uses homogeneous task-subtask trees, in which every node is a task. However, it is a trivial change to augment the approach so that the task-subtask decomposition is mediated by the selection of a problem solving method. To some extent this is what happens anyway, except that the choice of a method is *implicit* in the selection of the subtree, rather than *explicit* in the decomposition structure. The lack of an explicit notion of problem solving method is actually a limitation of the GDM approach, given that a method provides a powerful abstraction for clustering together both pragmatic and knowledge requirements over an application domain.

knowledge is available only after a system has been tested.[6]

Even when the method selection rules can be effectively specified, the model development process may still be problematic, due to problems with the organization of the task-method decomposition tree. For instance, Orsvarn (1996) discusses the problems he encountered when attempting to reuse Benjamins' library and illustrates a scenario in which a method selection rule is predicated on an assumption which arises somewhere else in the task-method structure. Orsvarn suggests a number of principles which should be applied when constructing libraries of task-method structures. The most important one is *method generality*: adaptation of task models is difficult and therefore should be avoided. Hence, methods should be as generic as possible. This principle, in its informal connotation, applies to the library of method components which I will present in chapter 8.

2.4.2.2. Lack of a clear theoretical basis.

Another problem with published method decomposition libraries is that they do not provide a clear framework in which all the tasks and methods are situated. Even when these libraries are task-specific and a clear model of the task is provided - e.g. in (Benjamins, 1993) - there is no common model underlying the tasks and methods included in the library. As a result it is difficult to carry out comparative evaluations of alternative methods or task structures.

2.5. LEGACY OF THE REVIEW: WHAT NEEDS TO BE DONE

2.5.1. Modelling Framework.

As discussed earlier, while Common KADS provides a comprehensive and flexible modelling framework, its basic decomposition in tasks, inferences, and domain provides less an integration framework than a set of viewpoints over an application. Moreover, the Common KADS framework does not explicitly distinguish between domain and application specific knowledge. This is - in my view - a very useful distinction, both for knowledge analysis and component reuse. Finally, the notion of method ontology is only 'indirectly' supported in Common KADS.

These 'limitations' of the Common KADS approach are avoided by the Protégé-II framework, which - at least in the formulation provided by Gennari et al. (1994) - distinguishes between tasks, methods and domains, introduces the notion of method ontology and characterises application configuration as the construction of an application

---

6 Not surprisingly empirical methods are becoming increasingly popular, to identify the regions of a problem space where it is easier to find solutions (Cheeseman et al., 1991).

ontology, which 'mediates' between a reusable problem solving method and a reusable domain model.

The framework proposed in this thesis builds on the basic Protégé-II organization and extends it by providing a precise account of the relation between tasks, methods, and domain knowledge, and by highlighting the different types of application-specific knowledge which characterize an application model.

2.5.2. Development and Organization of Reusable Method Components.

2.5.2.1. Method Characterization and Development

In this review - see section 2.4.1 - I discussed two (related) approaches to method development which characterize this as a process by which a formal task specification is operationalized into a problem solving method by means of formal refinement steps.

As I said earlier, while I believe that this work is important for supporting formal specification and verification of problem solving methods, it seems to me that the method specification frameworks used by Wielinga et al. (1995) and Fensel and Straatman (1996) are too 'impoverished' to provide insights into the nature of knowledge-intensive problem solvers. In particular, consistently with the existential predicament of knowledge-based systems, I would argue that a framework which explicitly operates with search-oriented concepts is needed to understand the nature of the knowledge structures utilized by knowledge-based applications. For instance, in contrast with the negative results concerning the analysis of Propose&Revise obtained by Wielinga et al., a search-centred analysis of this method is able to shed light on the reasons for its 'unpredictable' competence (Motta and Zdrahal, 1996; Chapter 9).

Therefore, in this thesis I will take a different approach to characterising the development of problem solving methods and the relation between task specifications and methods. The main tenets of the proposed approach are as follows.

- The use of search as a mediating generic paradigm between a class of applications (problem type) and the specification of the associated problem solving methods.
- The use of a generic problem solving model as the 'foundation' for all problem solving methods applicable to a class of applications.
- The characterization of problem solving methods as fully specified refinements of the problem solving model associated with a problem type.

In particular, the use of an underlying search model provides both an epistemological device, which makes it possible to move from a task to a method dimension, and a computational model, which gives 'operational semantics' to the knowledge structures

introduced by problem solving methods.

2.5.2.2. The organization of a library of reusable problem solving components

In the above discussion I have criticised the task-method approach to library organization. In a nutshell task-method libraries lack a theoretical basis and rely on local selection rules, which fail to capture the complexity of the model development process.

Therefore here I will follow a different approach, which is closer 'in spirit' to the one taken by the DIDS researchers, and I will structure a library of problem solving components around a task-specific framework, rather than as a relatively unstructured association of methods to tasks. However, in contrast with the bottom-up approach used in DIDS, this task-specific model will be developed in a principled, top-down and task-independent way, by instantiating a generic search model of problem solving in terms of a parametric design task ontology. Thus, the resulting, task-specific model enjoys a clear theoretical basis, while the approach itself is not specific to parametric design.

An overview of the proposed approach is presented in the next chapter.

Chapter 3
An Approach to the Organization of a Library of Problem Solving Methods which Integrates the Search Paradigm with Task and Method Ontologies

This chapter provides an overview of the approach to knowledge modelling and library organization proposed in this thesis. The starting point of the approach is given by a formalization of a class of applications (a task ontology). This task ontology is then associated to a generic problem solving model, which is constructed by instantiating a task-independent model of problem solving as search in terms of the concepts provided by the task ontology. The resulting model, which is specific to a problem type, but method-independent, provides the basis for constructing a library of reusable problem solving components associated with the given problem type. Individual problem solving methods can then be derived from the generic problem solving model through a process of ontology specialization and method-to-task application. The resulting library of reusable components enjoys a clear theoretical basis and provides robust support for reuse.

3.1. FROM WORLD-VIEW TO THEORY

As discussed by Van Heijst (1995), the basic building block of a methodological framework is defined by the underlying *world view* - i.e. the fundamental principles and assumptions characterizing the proposed approach. Thus, chapter 1 can be regarded as introducing the world view underlying this thesis. Such a world view is informed by the characterization of knowledge-based systems as 'systems which perform search', by the view of knowledge acquisition as modelling, and by the emphasis on the identification and integration of generic and reusable components - the three themes of knowledge, models, and reuse.

In this chapter I will move on to the next layer of the methodological pyramid proposed by Van Heijst and illustrate a particular approach - *theory* in Van Heijst's framework - to the development and organization of reusable problem solving components. The proposed approach takes as its starting point the task/method/domain partition proposed by the CoE and Protégé-II frameworks but refines it along a number of dimensions.

- The proposed modelling framework explicitly includes a component, *application configuration*, which accounts for the various types of application-specific knowledge needed to integrate generic problem solving components with domain-specific knowledge. This component plays both an epistemological and an application development role. From an epistemological point of view it enables the proposed framework to account both for approaches, such as Generic Tasks, characterized by a strong *coupling*[1] of domain and task knowledge, and for those, such as KADS (Wielinga et al., 1992a), in which there is only a weak interaction between domain and task layers.
- Different kinds of ontologies are used to formally specify classes of applications (*task ontologies*), the knowledge requirements of problem solving methods (*method ontologies*), the conceptualizations used by reusable domain models (*domain ontologies*) and the application-specific concepts needed to integrate reusable domain models with a domain-independent problem solver (*application ontologies*).
- All problem solving methods applicable to a class of tasks are characterized as refinements of a common, task-specific, but method-generic problem solving model. The aim of this model is to provide a "clear theoretical basis" (Valente and Breuker, 1996) to a task-specific library of methods, thus facilitating the development, evaluation, and comparison of task-specific problem solving methods.
- The construction of the generic problem solving model mentioned in the previous bullet is carried out by instantiating a task-independent model of problem solving as search in terms of a task ontology. Thus, I am able to provide a principled account of the process required to construct the generic problem solving model and to characterize precisely the relation between a class of methods and a problem type. In particular, the method ontology associated with the problem solving model specifies the minimal knowledge requirements which apply to any problem solving method associated with the given problem type.

These ideas will be illustrated in the rest of this chapter.

---

[1] Here I use the word 'coupling' in a sense analogous to the standard use of the term in software engineering, as a measure of the degree of interconnection between two modules.

3.2. CHARACTERIZING GENERIC TASKS

A task specifies a goal for a problem solver, such as producing a correct configuration for an elevator, or diagnosing a pulmonary problem. The notion of task is crucial to knowledge modelling, given that - as discussed in chapter 1 - knowledge systems are essentially performance systems, i.e. they are characterized and evaluated on task-specific criteria[2]. In accordance with this view I will illustrate an approach which centres the construction of a library of problem solving components around the specification of a particular class of *generic tasks*: *problem types.*

3.2.1. Types of Generic Tasks

A generic task specifies a knowledge level, application-independent description of the goal which has to be attained by a problem solver. Optionally, a generic task can also include the specification of a *task body*, a mechanism which describes how to achieve the goal of the task. In what follows I will use the term *executable task* to refer to this class of tasks and the term *goal specification task* to indicate generic task specifications which do not include a task body. *Problem types*, such as parametric design, provide a well-known class of goal specification tasks. Executable tasks divide in turn into *primitive* and *composite* tasks. The former solve a task directly, the latter by introducing a number of subtasks.

A goal specification task is typically described by specifying its inputs and outputs, and the associated goal (Benjamins, 1993; Steels, 1990). For instance, a parametric design task can be informally described as shown in figure 3.1. The goal of this generic task is to produce a complete and *valid* design model (i.e. a model which satisfies all given requirements and violates no applicable constraints), given an input design specification consisting of *design parameters*, *design constraints*, *design requirements*, *preferences* over possible designs and a *cost function*.

Generic Task Parametric Design	
Inputs:	Parameters, Constraints, Requirements, Cost-Function, Preferences
Output:	Design-Model
Goal:	"To produce a complete and consistent design model, which satisfies the given requirements"

Figure 3.1. Informal specification of the parametric design task.

---

2 While this might sound obvious, it is not. On the contrary, as discussed in detail in chapter 1, the old view of knowledge systems characterized these as models of human expert behaviour. Accordingly, KBS evaluation techniques attempted to verify whether the behaviour of these systems was consistent with that of the relevant experts. Such an anthropomorphic view of technology still pervades much of the AI field, as shown by the recent debate on whether Deep Blue is AI.

A goal specification task is solved by a problem solving method. As already mentioned in the previous chapter, decoupling the specification of a task from that of the applicable methods has the advantage of introducing flexibility in the structure of a library and in problem solving: in general, a task can be solved by several different methods. However, in some cases there is no room for flexibility, a task might only admit one, 'organic' problem solving method. In this scenario it makes sense to associate a task body directly to a task specification, to produce an executable task. For instance, figure 3.2. shows part of the task-method structure of the generic model of problem solving associated with the class of parametric design tasks - see chapter 7 for a detailed description of the model.

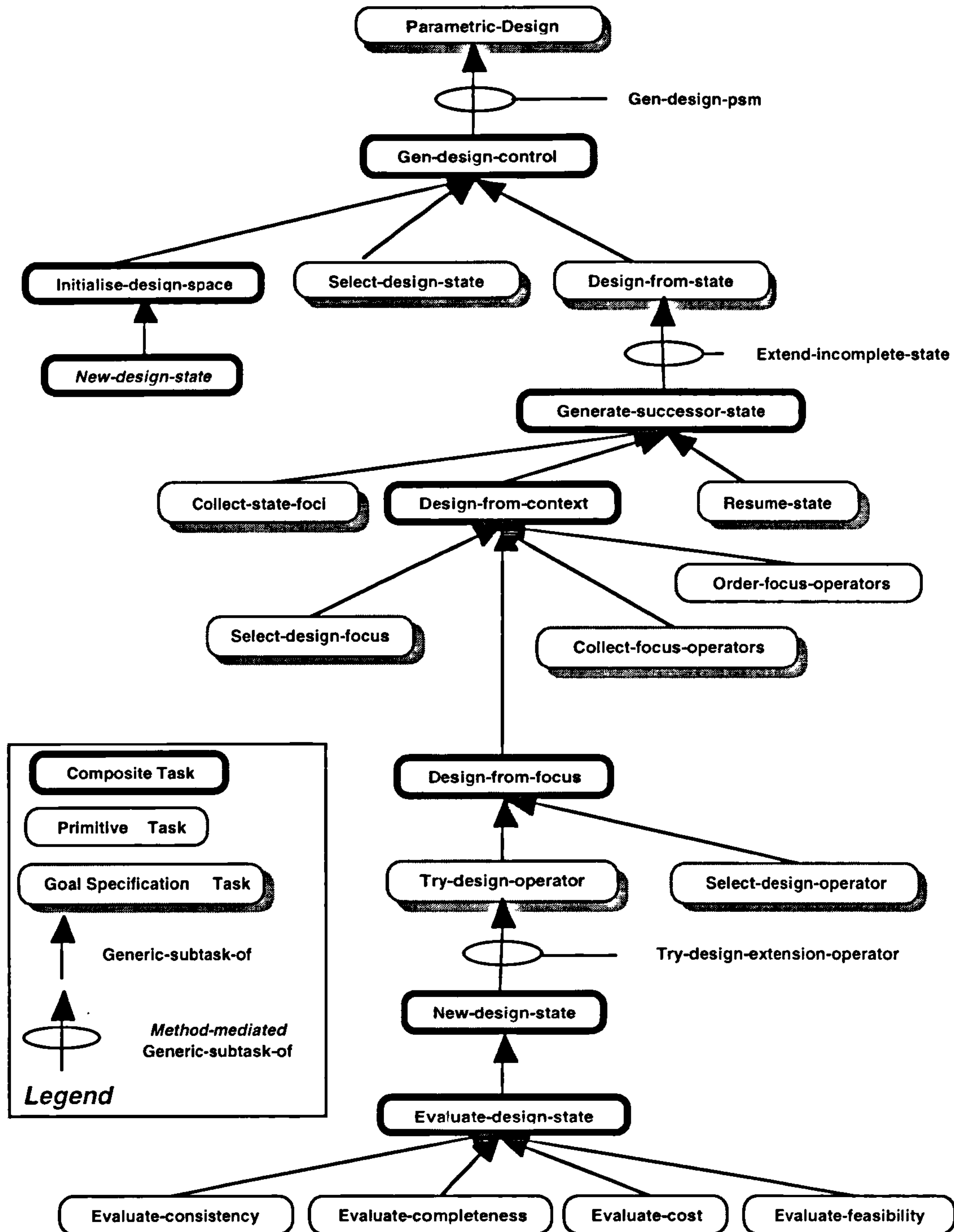


Figure 3.2. Sample task-method structure from parametric design library.

Task-method structures, such as the one shown in figure 3.2, suggest another important distinction, between problem types and *generic subtasks.* The former specify the root of a task-method structure; the latter appear in task-method structures as intermediate or end nodes. Some generic subtasks are related to specific problem solving methods - e.g. revise-design occurs when using a Propose&Revise approach - others, such as evaluate-design, are associated not with a particular method but with a problem type. The fact that generic subtasks occur regularly in classes of applications is an important feature of

knowledge-intensive problem solving. In particular, we can exploit this property to construct a model of problem solving, associated with some class of applications, say T, which comprises the problem solving actions carried out when solving an instance of T. Such a model of problem solving can therefore be seen as specifying the space of generic problem solving components which can be used to construct problem solvers for a class of applications.

3.2.2. Generic tasks: viewpoints over applications

Generic tasks (and in particular, problem types) are identified by abstracting common elements from classes of applications. Of course, the inverse process also applies and the specification of a problem type defines a conceptual framework, which can be used to describe a particular application. From a knowledge acquisition point of view, such a description provides both a task-centred mechanism for validating the completeness of the available application knowledge - i.e. completeness can be achieved by ensuring that all knowledge required to characterize the task has been acquired - and a way to focus the knowledge acquisition process, so that only the knowledge required by the generic task is ever acquired.

Consistently with the view of knowledge acquisition as modelling outlined in the previous chapter, it is important to emphasize that a problem type only provides one particular *viewpoint* over an application and that different viewpoints are in general possible. For example, the special issue on the Sisyphus-I office allocation problem of the International Journal of Human-Computer Studies (Linster, 1994) includes both solutions which characterize Sisyphus-I as a design problem (Balkany et al., 1994) and solutions which model it as a classification problem (Gaines, 1994). In other words, it is not so much that a specific application, or part of it, falls necessarily into a particular problem type. Rather, it is a problem type which provides a conceptual framework for characterizing an application and for focusing the knowledge acquisition process.

3.3. GENERIC TASK SPECIFICATION AS TASK ONTOLOGY

A way to precisely characterize the viewpoint expressed by a generic task is provided by the notion of *ontology* (Guarino, 1997; Valente and Breuker, 1996; Gruber, 1993; 1995; Schreiber et al., 1995; van Heijst et al., 1997). Gruber (1993) defines an ontology as an "explicit specification of a conceptualization". This definition suggests that the role of an ontology is to identify the entities and relations which exist in some universe of discourse and define the conceptual vocabulary to be used to refer to and reason about them. Thus, an ontology formalizes a viewpoint over a target domain (Gruber, 1993; Schreiber et al., 1995).

While the definition given by Gruber highlights the essence of what is an ontology, it is incomplete in two respects. In particular it fails to point out that the main role of an ontology is to provide a *reusable specification* and that any such specification can only be *partial* (Guarino and Giaretta, 1995; Schreiber et al., 1995). Thus, a more complete definition can be given as follows:

> *An ontology is a partial specification of a conceptual vocabulary to be used for formulating knowledge-level theories about a domain of discourse. The fundamental role of an ontology is to support knowledge sharing and reuse.*

This definition is consistent 'in spirit' with the ones proposed by Gruber (1993), Guarino and Giaretta (1995) and Schreiber et al. (1995), but it also exhibits some differences. As already said, in contrast with Gruber's definition, the above definition emphasizes that ontologies only provide partial specifications of a conceptualization. I also prefer to do away with the term 'conceptualization' and use instead the expression 'conceptual vocabulary'. As Guarino and Giaretta point out (1995), the term 'conceptualization' can be ambiguous, as it can be used to refer both to the contents of a model and to the terminology used to describe it. My definition also differs from the one given by Schreiber et al., because I do not subscribe to their view of an ontology as a meta-level specification. As Guarino (1997) points out, whether or not an ontology is characterized at the meta-level is not an intrinsic property of the ontology but depends on the relation between the ontology in question and some other model - e.g. another ontology. Finally, in contrast with the definition given by Guarino and Giaretta and in agreement with the view taken by Schreiber et al., my definition emphasizes that an ontology cannot be recognised as such only by looking at its logical properties, but it is essentially a vehicle for supporting reuse.

Thus, in this work I will use task ontologies to formalize the reusable conceptualizations expressed by generic tasks. A task ontology defines the universe of discourse described by a generic task, provides a set of modelling primitives for representing task instances and specifies the knowledge structures which need to be acquired to apply a task viewpoint over some problem. For instance, a task ontology for parametric design specifies a set of modelling solutions for representing the various facets of a parametric design task - e.g. requirements, constraints, parameters - and, at the same time, identifies these as the target of a knowledge acquisition process. Thus, a task ontology (like any other ontology) provides two sets of definitions, *external* and *internal*. The external definitions, sometimes called *domain views* (Fensel and Straatman, 1996), specify the viewpoint defined by a generic task and the target of the knowledge acquisition process. Internal definitions define the various entities and relations needed to circumscribe the

meaning of the external definitions. Hence, in contrast with normal database schemas, an ontology provides not just a modelling template but also a semantically rich specification.

A graphical sketch of a subset of the parametric design task ontology is shown in figure 3.3 - see chapter 6 for a detailed discussion of the ontology and an explanation of the graphical notation. The concepts shown in bold in the figure indicate the external viewpoint imposed by the ontology. This external viewpoint comprises the classes associated with the input roles of the generic task (i.e. preferences, cost function, parameters, constraints and requirements) and a relation, current-design-model, which is satisfied by the particular design model which, at each stage of problem solving, is the target of the design process.

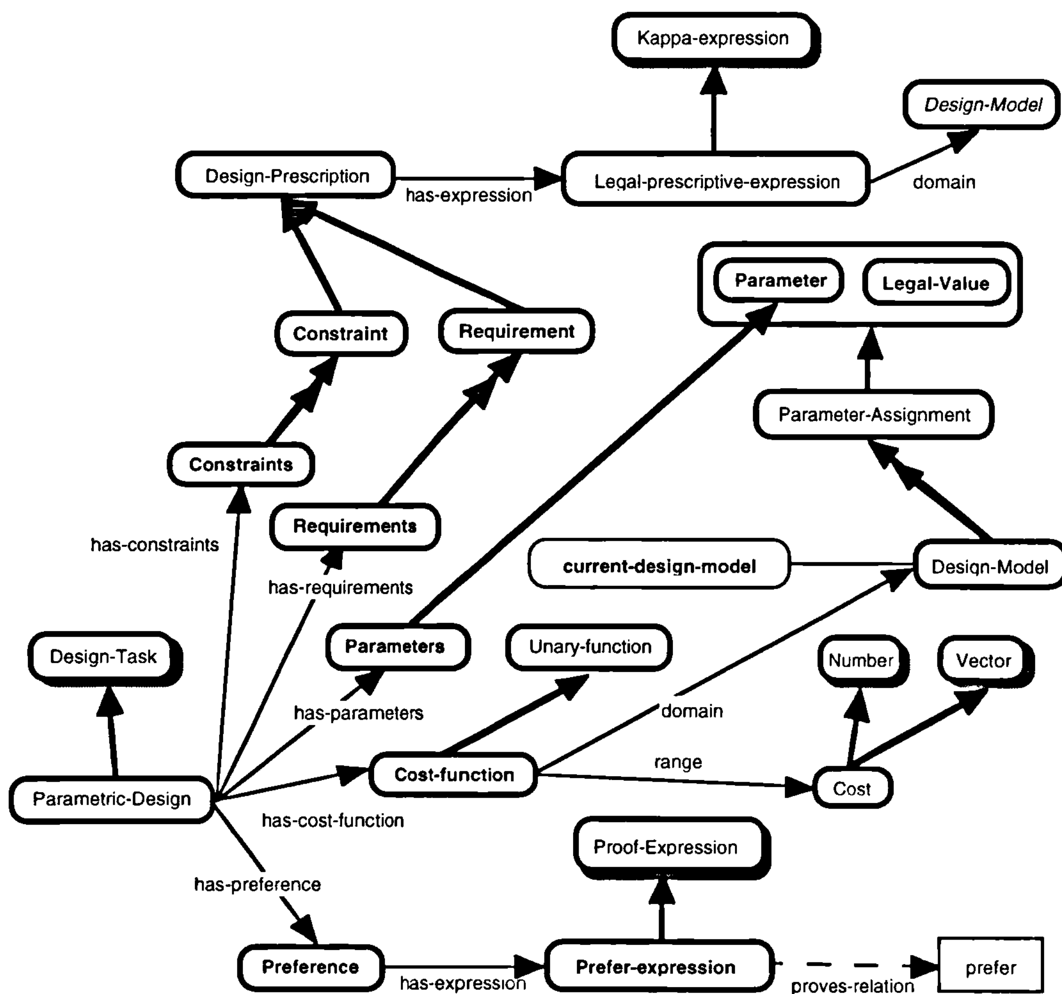


Figure 3.3. A partial view of the parametric design task ontology

3.4. FROM GENERIC TASKS TO GENERIC PROBLEM SOLVING METHODS

3.4.1. Search as an epistemological device to integrate tasks and methods

An important objective of this work is to develop a model providing a tighter integration between task specifications and problem solving methods, than the one offered by the simple 'method-solves-task' type of associations normally found in libraries of problem solving components. Thus, in this section I will outline an approach which takes as input i) a task ontology, say T, and ii) a problem solving paradigm, and produces as output i) a method-independent, but task-specific, model of problem solving, Gen-PSM, and ii) a generic method ontology. Gen-PSM fulfils two roles: it provides a basis for comparing alternative methods and it makes it possible to operationalize method development as a process in which novel refinements of Gen-PSM are produced. The generic method ontology associated with Gen-PSM specifies the minimal ontological commitments which need to be fulfilled by a problem solving method applicable to the class of applications denoted by T.

Specifically, I will show the approach in the context of parametric design problems and will adopt search (Newell, 1980) as the generic problem solving paradigm - see figure 3.4.

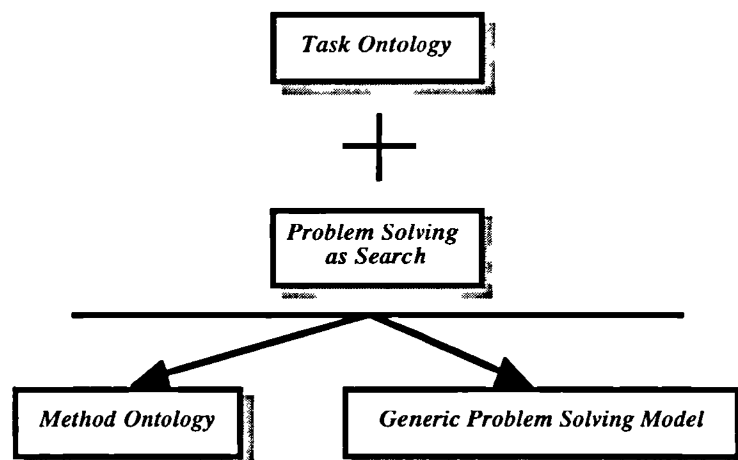


Figure 3.4. Integrating tasks and problem solving methods.

In principle there are two possible objections which can be raised here: i) the search paradigm can be seen as limiting the range of problem solving behaviours and ii) the approach can be criticized as going back to the old view of AI as the development of weak, general-purpose problem solvers (Newell and Simon, 1972). I believe that both objections can be easily refuted. In chapter 1 I have pointed out that search is not just a particular problem solving paradigm but, in a sense, it provides the fundamental metaphor for describing knowledge-based problem solving. Therefore, by adopting the search

paradigm I am not restricting myself to a class of problem solvers smaller than the class of knowledge-based systems, which is exactly the class of systems I am concerned with. The second objection can be refuted 'operationally', by constructing 'strong', i.e. task-oriented, instantiations of the search paradigm, which correspond to method-generic, but task-specific, problem solving models.

In the next section I will briefly describe how this approach has been used to construct a generic problem solving model for parametric design. A full description of this process and the resulting model and ontology will be given in chapter 7.

3.4.2. A search-based model for parametric design problem solving

A task ontology specifies a task in terms of initial and goal states in the universe of discourse. For example, in the case of parametric design the initial state is described in terms of the parameters whose values must be calculated, the requirements which must be satisfied and the constraints which must not be violated (Wielinga et al., 1995). Given an initial and a goal state, the specification of a suitable PSM describes a process for reaching a goal state from the initial one. These problem solving processes are typically described in terms of high level inference steps - e.g. design problem solving is often described as proposing, critiquing and modifying tentative solutions (Chandrasekaran, 1990). The problem with these descriptions is that they are defined in terms of conceptual operations, which differ from PSM to PSM and from task to task. Moreover, they introduce additional ontological commitments on top of those associated with a task specification - e.g. a Propose&Revise model of design problem solving assumes the existence of the relevant procedures and fixes. What we need here is a generic epistemological model which allows us to characterize any generic problem solving process, which can be applied to a given task specification. This model should generalize from any specific PSM by not introducing ontological commitments in addition to those associated with the relevant task ontology.

The answer to this problem is given by the notion of problem solving as search, which characterizes problem solving as the process of navigating a *state space*, starting from an initial state, to achieve one of a number of solution states - see figure 3.5. The search model perfectly satisfies our requirements. On the one hand, given a task specification, unless we assume additional knowledge related to the state space, the only course of action left to a problem solver is to search (existential predicament of intelligent systems). On the other hand all PSMs can be seen as performing search. Specialised problem solving methods essentially increase the efficiency of the search process by making use of additional problem solving knowledge.

3.4.2.1. Parametric design as search

The generic search metaphor can be specialised for a parametric design problem solving context, so that the process becomes one of navigating a *design space* efficiently. Each node in the design space, i.e. each design state, say S_i, can be characterized in terms of a design model, D_i.

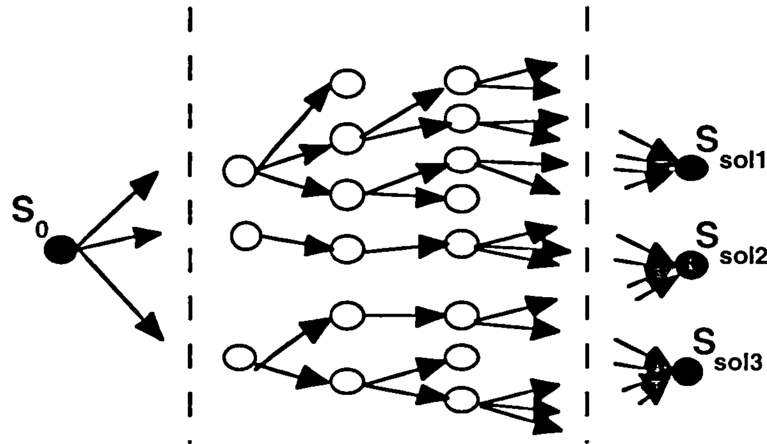


Figure 3.5. Problem space view of the design process.

A *design operator*, which is represented in figure 3.5 as a directed link between two states in the design space, defines a transition between design models. A design operator is a generic concept which describes a procedure which modifies a subset of a design model, thus causing a move in the design space. For example, in an office allocation application a design operator could be a rule which allocates an employee to a room; in an elevator configuration problem it could be a *fix* which modifies the value of a parameter - say the position of the counterweight shown in figure 1.1 - in response to constraint violations. Hence, the notion of design operator can be regarded as a generic way to characterize the interaction between a parametric design problem solver and a design model. In practice - i.e. at the application level - a parametric design problem solver can do very little other than modifying design models. Thus, we can derive a very generic method ontology for parametric design problem solving simply by adding the specifications of the concepts 'design space', 'design operator' and 'design state' to the existing task ontology. The resulting ontology can be regarded as the most generic method ontology for a parametric design problem solver which subscribes to the design space view shown in figure 3.5. The assumption here is that any specific problem solving method will subscribe to this ontology and typically further refine it. For instance, a Propose and Revise problem solver will undoubtedly refine the concept of design operator by differentiating between those design operators used during the Propose task, often called *design procedures*, and those used during the Revise task, *design fixes*. However, it is difficult to envisage a parametric design problem solver which can do away with the notion of design state and

design operator at the knowledge level. In other words the assumption here is that the proposed, task-oriented instantiation of the search metaphor is adequate to describe the behaviour of knowledge-based, parametric design systems.

3.4.2.2. Identifying generic problem solving actions for parametric design

A method ontology specifies the knowledge roles imposed by a problem solving method over an application domain. These are useful to drive the knowledge acquisition process, but are not sufficient to characterize the problem solving behaviour of a method. A description of the control structures and inference mechanisms of a method is also required. Of course, my goal here is not to define one particular parametric design method, but rather to develop a generic problem solving model which can be used to account for the variety of parametric design problem solvers which can be found in the literature. To achieve this goal, I need to identify the generic problem solving actions which characterize parametric design problem solving.

The approach I have taken here to formulate a generic model of parametric design problem solving combines both top-down and bottom-up analysis. Given the design space representation I have adopted, there are essentially only four actions which can be carried out: selecting a design state, selecting a design operator, applying a design operator to the selected state, and evaluating the resulting design model. The latter is needed to assess its properties, e.g. whether it provides a solution, its cost, etc. Although these four subtasks are adequate to describe the process of searching the design space, an empirical analysis of existing methods shows that typically they employ a more fine-grained breakdown. For instance, the selection of a design operator goes normally through a number of decision-making activities, which include the selection of a *design context* (e.g. design extension vs. design revision context in Propose&Revise problem solving) and of a *design focus*. The latter can be seen as an abstraction mechanism which extracts from the current design state the relevant information which is used to restrict the field of possible operators. An example of a design focus is the constraint violation which a Propose and Revise problem solver attempts to resolve during the application of a particular design fix.

The result of this task analysis is a model such as the one shown in figure 3.2, which identifies the space of generic subtasks associated with parametric design applications, rather than with a particular problem solving method. In chapter 7 I will illustrate this generic problem solving model in detail and provide a formal specification of both the model and the associated method ontology.

3.5. CHARACTERIZING PROBLEM SOLVING METHODS

3.5.1. Definition of problem solving method

In earlier approaches to knowledge modelling (Wielinga et al., 1992a; Chandrasekaran, 1986) the notion of task was used to refer both to the definition and to the body of a task. Later, it became clear that by making explicit the notion of problem solving method it was possible to specify richer libraries of problem solving components and also to develop more powerful frameworks supporting flexible problem solving (Punch, 1989; Steels, 1990; Chandrasekaran at al., 1992; Benjamins, 1993). In sum, it is useful to decouple tasks and methods as they provide two distinct foci for reuse.

So far I have been using the notion of problem solving method informally, characterizing it as a way to solve a class of tasks. Having introduced a significant part of the proposed modelling framework, I can now define a problem solving method more precisely, as follows.

A problem solving method is a domain-independent, knowledge-level specification of problem solving behaviour, which can be used to solve a class of problems, say C. A problem solving method can be characterized as a particular specialization of the generic problem solving model associated with C, say Gen-PSM, and its method ontology is a specialization of the method ontology associated with Gen-PSM.

This definition essentially brings together the various aspects of the approach I have outlined so far. I'm interested in modelling, not implementation, and therefore I'm interested in descriptions of problem-solving methods at the knowledge level. The definition also emphasizes that a problem solving method is a task-centred agent. Finally, in accordance with the approach outlined in the previous section, I characterize a problem solving method as a refinement of the inference and control structure of the relevant, task-specific generic problem solving model, Gen-PSM. Its ontology also refines that of Gen-PSM.

For instance, the method ontology of a Propose&Revise method refines the default method ontology for parametric design by (at least) differentiating between two types of design operators, procedures and fixes. Analogously, the A*-design method (a customization of A* for parametric design) is defined by specializing six generic tasks and the notion of design cost used in Gen-PSM.

Thus, the proposed approach makes it possible to structure a library of methods in terms of two lattices. These specify the ontological commitments (*ontology lattice*) and the subtasks and sub-methods (*problem solving lattice*) associated with each problem solving method in the library. The primary role of the ontology lattice is to support knowledge

acquisition; that of the problem solving lattice is to support the construction of problem solving methods through a specialization process.

3.5.2. Modelling problem solving methods

A problem solving method is described in terms of the following components.

- **Name**. A symbol which uniquely identifies a problem solving method in a library.
- **Input Roles**. A specification of the various types of knowledge required as an input by a method.
- **Output Role**. A specification of the type of output produced by a method.
- **Control Roles**. Additional, intermediate knowledge structures introduced by the method during problem solving.
- **Goal**. The goal of the method. This field is usually left blank given that the goal of a method is normally 'inherited' from the associated task. In some cases however, a method may need to weaken the goal of the current task. For instance, while the goal of a design task might be to find a globally optimal solution, such a task might still be 'solved' (in a weaker sense of the word) by a method which finds some solution.
- **Task Structure**. The task-subtask decomposition introduced by a method. This field is empty if the method is *primitive*.
- **Knowledge Flow**. The data flow relationships between the subtasks of a method.
- **Body.** The specification of a procedure which is executed when the method is applied.
- **Generic Tasks Tackled**. The classes of generic tasks for which the method is suitable.
- **Applicability Condition**. An expression which is verified by the task instances to which the method can be applied. While the association between a method and a set of generic tasks is only meant to provide a coarse-grained mechanism for characterizing the applicability of a method, this field provides a fine-grained criterion for deciding whether or not a method can be used to solve a particular task.
- **Method Ontology**. A specification of the ontological requirements introduced by a method. This formalizes the knowledge structures denoted by the input and output roles of a method.

It is important to keep in mind that a knowledge-level description of a problem solving method is always relative to a certain grain-size. For instance figure 3.6 provides a complete description of the knowledge flow of a Propose&Revise problem solver, but its granularity is clearly very coarse. In general, in the rest of the thesis I will use the expression 'complete problem solving method' to refer to a complete instantiation of the above method specification template, for a given level of description. However, in many cases one is interested in *partial* problem solving method specifications. For instance, it is often useful to include method descriptions in a library, which do not specify a particular control regime, thus abstracting from control issues.

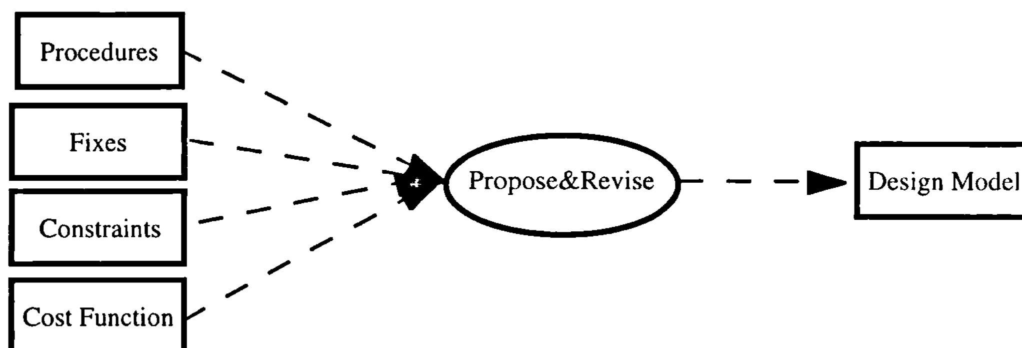


Figure 3.6. Knowledge flow in Propose&Revise

3.6. REUSABLE DOMAIN MODELS

The third component of the modelling framework proposed in this chapter deals with the domain 'dimension' of knowledge based systems. As discussed in the previous chapter, different approaches take different lines on the nature of domain knowledge and on its role in knowledge modelling, sharing and reuse. In particular, of the approaches reviewed in the previous chapter, only Protégé-II considers domain knowledge bases which are truly task-independent (Gennari et al., 1994). Common KADS and CoE suggest that in general multiple *domain views* may be needed over an application domain. For instance, Steels (1990) points out that in a circuit-board diagnosis application, domain experts were using four different types of domain models: causal, structural, functional and shift-register models. Although these domain models have a strong task-oriented flavour, a-la-Generic-Tasks, they are - at least in theory - reusable across applications. In the rest of the thesis I will use the term *mono-functional* to characterize domain models which impose a strong, functional view over a domain. Further down the task-centred philosophy, approaches such as Generic Tasks or DIDS take a very strong, use-oriented line and do not consider domain models as separate from task models, but rather see these as determining both the nature and the form of the application domain knowledge (Bylander and Chandrasekaran, 1988).

At the opposite end of the task-centred spectrum work on *multi-functional* knowledge bases attempts to build large bodies of knowledge which, while domain-specific, are not committed to a particular task or problem solving method. By far the most famous of these large multi-functional knowledge bases is Cyc (Lenat and Guha, 1990), which comprises (or aims to comprise) the millions of notions which make up 'consensus reality'. These include common-sense notions, e.g. time and space; knowledge about the organization of human institutions, e.g. family, school; naive physics and biology, e.g. things fall and organisms die; and innumerable other concepts which human beings use routinely to make sense of phenomena in the world.[3] The rationale for building multi-functional knowledge bases is quite simple: they provide a useful focus for reuse. As Murray and Porter (1988) point out, "The grid of potential knowledge bases has three dimensions: domain, task, and problem solving method. Building knowledge based systems for individual cells of this grid is both costly and short-sighted".

Unsurprisingly, the line I will take here is consistent with the reuse-oriented approach of this work. Thus, as a first approximation, I characterize the domain component of my framework as specifying multi-functional, reusable domain models. Having taken this line, I need of course to solve a number of issues. First I need to address the point concerning the interaction between task and domain knowledge. Second, I have to tackle the relation between mono-functional and multi-functional knowledge bases. Finally, as pointed out in the previous chapter, much of the work on reusable domain models is actually about domain ontologies. How do these relate to domain models and what role do they play in the proposed framework? In the next sections I will address these issues, starting with the interaction problem.

3.6.1. The Knowledge Interaction Problem

The knowledge interaction hypothesis (Bylander and Chandrasekaran, 1988) states that both the type of knowledge required by an application and its representation are strongly determined by the chosen task and/or method. In terms of the framework developed so far, this hypothesis indicates that both the knowledge acquisition process and the domain modelling schema are determined by the method ontology associated with the current problem solving method. In a sense this is trivially true. If I assume that a knowledge

---

[3] The assumption underlying the Cyc project is that by relying on this huge body of common-sense knowledge future AI systems will be able to overcome their current 'brittleness', i.e. their incapability of coping with novel situations. Whether or not the Cyc approach is sound is not really the issue here. My interest here is to understand what is domain knowledge, more precisely what epistemologically useful types of domain knowledge do exist, and how to structure the domain component in the proposed modelling framework.

acquisition process starts with a pre-existing problem solving method and no domain component, then the quickest route to complete an application model is of course to acquire the knowledge required by the method and represent it in terms of the modelling schema specified by the method ontology. In a parametric design problem, this process will involve acquiring - among other things - the various parameters and constraints and representing them in terms of the relevant constructs provided by a particular parametric design method ontology. This approach was used by the researchers building knowledge acquisition tools according to the role-limiting approach (Marcus, 1988) and by my colleagues and myself when tackling the VT elevator design problem in the VITAL project (Motta et al., 1996). A more interesting scenario is one in which I can already select or configure from my library of reusable components not only a problem solving method for parametric design, but also a pre-existing domain knowledge base (say a database of employees) for the application domain in question (say office allocation). In this case I would like to reuse not just the problem solving method but also the existing domain model. The knowledge interaction hypothesis poses two objections here: the pre-existing domain model will not comprise the 'right' knowledge, and its modelling schema will be inappropriate. The second objection is easy to address. Any representation mismatch can be addressed by adding appropriate *mapping mechanisms* (Gennari et al., 1994) which allow the problem solver to retrieve and make use of the relevant definitions from the domain model. Of course, this can be in principle inefficient. However, this is not the issue here, as we are not looking for efficient representation solutions but for modelling mechanisms enabling us to build knowledge-level models of applications out of reusable components.

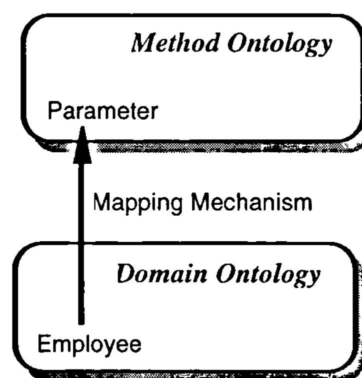


Figure 3.7 Mapping domain to method ontologies.

The other objection concerns the mismatch between the knowledge required by the method and that embedded in the multi-functional domain knowledge base. For example, in an office allocation problem it is important to acquire knowledge about office preferences and allocation requirements. This knowledge is very much application-specific and therefore cannot be part of a multi-functional domain knowledge base.

Hence, it needs to be acquired on an application-specific basis. In order to account for these two modelling situations, formulating the relevant mapping mechanisms and acquiring application-specific knowledge, the framework proposed here comprises a fourth type of component, *application configuration knowledge*. In contrast with the other three components this does not specify a class of reusable components. Rather, it provides the 'glue' for integrating reusable problem solving and domain elements.

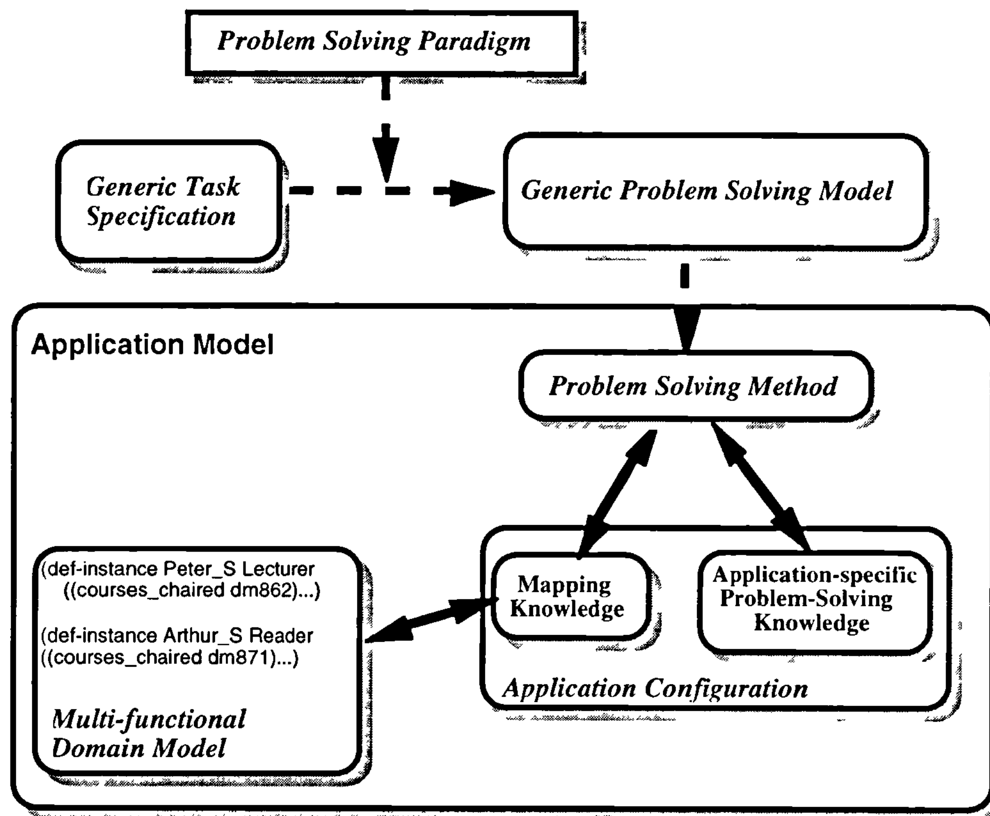


Figure 3.8. Overall Modelling Framework.

The approach described here tries to reconcile the trade-off between *usability* and *reusability*. The more generic a model, the more reusable it is. The more specific a model, the more usable it is, although in a restricted space of applications. The proposed framework addresses these issues at the knowledge level, i.e. in terms of modelling solutions, rather than efficiency considerations. At the knowledge level, the problem can be reduced to one of different degrees of coupling between different components of a knowledge model. In particular, the framework makes the following assumptions.

- **Strong coupling between generic tasks and problem solving methods.** Problem solving methods are designed to perform efficient problem solving and they are typically designed with a class of tasks in mind. A close coupling between a generic method and a generic task is therefore not so much a requirement, as a consequence of the *raison d'être* of problem solving methods. In my framework I strengthen this 'natural' coupling by using the choice of a problem

solving paradigm as a mechanism for providing a principled approach to developing a generic problem solving model and method ontology for a given problem type.

- **Weak coupling between generic problem solving models and multi-functional domain knowledge.** By definition, multi-functional knowledge is domain knowledge which characterizes task independent aspects of a domain, i.e. domain knowledge which can be used in many different ways. For the sake of reusability this knowledge is modelled in a task and method independent way. It follows that only weak coupling of multi-functional and problem solving knowledge can be supported. This weak coupling is expressed by means of mapping mechanisms.
- **The knowledge interaction problem can be tackled at the knowledge level by introducing an application-configuration component.** In particular, the assumption here is that the interaction problem can be operationalized at the knowledge level in terms of the activities of acquiring application-specific knowledge and establishing the appropriate mappings between the problem solving and domain components.

Thus, the application configuration component comprises two types of application-specific knowledge, *mapping knowledge*, which is required to integrate weakly coupled components, and *application-specific problem solving knowledge*, which is required by the chosen problem solving method and needs to be acquired for each application. This second type of knowledge has often a heuristic connotation. For instance, the design rules discussed in section 1.2.2.2 are indeed examples of application-specific, heuristic knowledge. However, not all application-specific problem solving knowledge is heuristic. For instance, the cost function in design applications is typically application-specific but not heuristic. Therefore, it seems to me that the often-encountered dichotomy between "the conceptual aspects of a domain theory and the *heuristic annotations*" (Steels, 1990) is of limited use. The main issue when constructing reusable models is to separate the domain elements which can be reused across applications - i.e. multi-functional domain knowledge - from those which are application-specific. The latter might include, but cannot be reduced to, heuristic knowledge.

In the rest of the thesis I will refer to the model of application and library organization shown in figure 3.8 as the *TMDA* (*Task/Method/Domain/Application*) framework.

3.6.2. Integrating domain ontologies and mono-functional models into the framework

Earlier I mentioned that approaches to domain modelling include not only multi-functional models, but also domain ontologies and (what I termed) mono-functional models. How do these fit into the proposed modelling framework?

3.6.2.1. Mono-functional models as customised domain views

Mono-functional models are *abstractions of method ontologies*. For instance, a causal domain model can be regarded as fulfilling the requirements of a diagnostic method which makes use of causal links to trace back the possible reasons for a faulty component. In this case we can provide a more flexible scenario and make use of mapping mechanisms to link a multi-functional domain model to several possible mono-functional models, each providing a view required by a problem solving method. The resulting scenario is shown in figure 3.9.

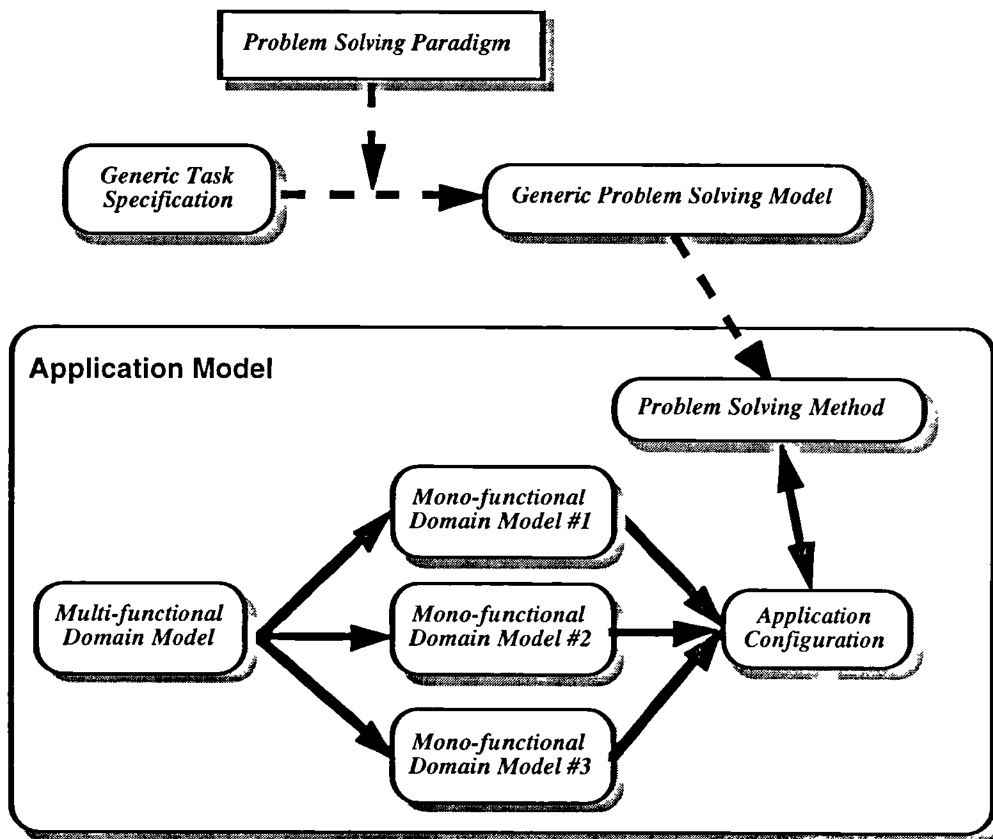


Figure 3.9. Integrating multiple domain views in the modelling framework.

In particular the same configuration mechanism used to integrate a domain-independent problem solving method with a method-independent domain model can also be used to integrate mono-functional and multi-functional models - see figure 3.10.

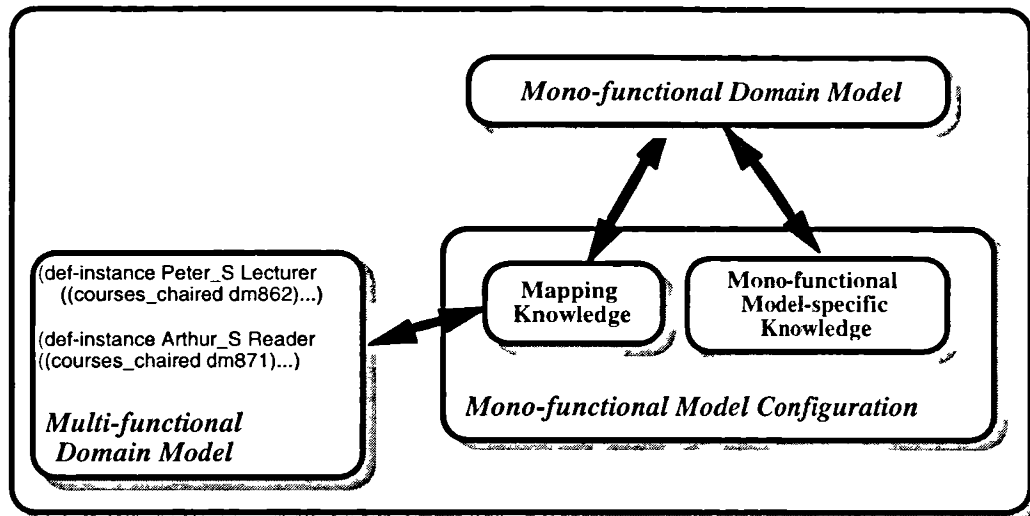


Figure 3.10. Configuring mono-functional models

In general, several levels of mappings might be required to integrate multiple domain views with a method ontology on the one hand and a multi-functional model on the other hand. This process can be characterized as the construction of an *application ontology* (Gennari et al., 1994).

3.6.2.2. The role of domain ontologies

In the context of the proposed approach a domain ontology can be defined as a reusable domain model, multi-functional or mono-functional, which is independent of an application domain. In other words a domain ontology does not contain knowledge about a particular application domain. For instance, a medical ontology could contain definitions for the terms normally used in medical applications - e.g. therapy, disease, diagnosis, symptom - but would not contain information about a particular application domain, e.g. information about a particular instance of a disease which has occurred to a particular patient.

Domain ontologies are often specified for large application domains, such as medicine (Falasconi and Stefanelli, 1994) and law (Valente and Breuker, 1996). While it is possible to find in the literature quite a few task-independent ontologies, it is important to note that not all task-independent ontologies are strictly-speaking domain ontologies - in the sense of providing a conceptualization of a class of application domains. In particular, it is possible in the literature to find examples of the following types of task-independent ontologies.

- Ontologies proposing a conceptualization of an application domain or application area, which can be shared by researchers working on the same application, or class of applications. An example of this kind of ontology is provided by the legal ontology discussed in (Valente and Breuker, 1996).

- Ontologies specifying the modelling schema used by a particular knowledge base. These ontologies are different from those discussed in the previous bullet because here the goal of the ontology is not so much to provide a framework which can be agreed upon by all researchers in an area, but merely to make explicit and formalise the modelling schema used by a knowledge base. An example of this kind of ontology is the VT-design ontology (Gruber et al., 1996), which describes the modelling schema employed to encode the VT knowledge base (Yost and Rothenfluh, 1996).
- Ontologies specifying a general-purpose, knowledge representation schema. For instance the Frame Ontology in Ontolingua (Gruber, 1993) describes the primitives characterizing frame-based knowledge representation systems.

Of these three classes of ontologies only the first one is strictly speaking a domain ontology according to the definition of the term given in section 3.3. Ontologies such as the frame ontology are not specifications of application domains - as is the case with a medical or legal ontology. A frame ontology provides the layer underneath that of conceptual specifications: it is a specification of the primitives to be used for building conceptualizations. Therefore, following Gruber (1993), I will henceforth use the expression *representation ontology* when referring to ontologies such as the frame ontology, to distinguish them from 'proper' conceptual ontologies.

The difference between the other two classes of ontologies is much more subtle and has to do with the aims and the nature of the ontologies. The primary aim of an ontology such as the legal ontology discussed by Valente and Breuker is to suggest a consensus on a terminology for modelling legal reasoning. The ontology provides a very abstract specification, almost the topmost layer of any legal model; it essentially identifies the main types of knowledge in legal reasoning, e.g. normative knowledge, common sense knowledge, etc. From the discussion it is clear that the authors do not suggest that such an ontology should be used directly to implement legal systems. Their aim is to shape a consensus on the relevant terminology and viewpoints applicable to legal modelling. While ontologies such as VT-design can be also regarded as suggesting a consensus on terminology, this aim is much less in evidence than the desire to make explicit the schema used to encode the VT knowledge base. The difficulty with this type of ontology is that knowledge representation schemas are not the same as knowledge-level modelling schemas. A knowledge representation schema has to strike a balance between modelling and computational requirements. In other words these schemas have an efficiency bias which distinguishes them from 'pure' knowledge-level ontologies, where the only issue is the modelling of knowledge. In the rest of this thesis I will exclusively focus on (so-called) 'pure' modelling ontologies and I'll address computational aspects separately.

3.7. CONCLUSIONS

In the previous chapter I have reviewed the state of the art in knowledge modelling and highlighted a number of open issues concerning the development and organization of a library of problem solving components. Here I will briefly illustrate how the proposed approach addresses these issues.

- **No clear theoretical basis for libraries of problem solving components.** The proposed approach characterizes each problem solving method as a refinement of a problem solving model associated with a class of problems. This model is constructed by instantiating a generic model of problem solving as search in terms of a generic task ontology. The model provides a foundation for all problem solving methods applicable to a class of tasks by specifying the minimal ontological commitments and the set of problem solving actions associated with a task-specific class of problem solvers.
- **No clear model of the method development process.** Method development is characterized in two stages. In the first stage a generic model of problem solving is built, which is associated with a class of applications. In the second stage problem solving methods are defined as refinements of the given model.
- **Limits of local selection knowledge in task-method structures.** Although the proposed library is hierarchically organised, method selection and configuration are not driven by simple decision-making rules. Methods are organized in a specialization hierarchy and the method selection criteria are global, rather than local.

In chapters 6-8 I will provide evidence for these claims by illustrating how the proposed approach has been used to construct a library of reusable components for parametric design problem solving. Before presenting the library it is however necessary to introduce the modelling language used to model the library components.

Chapter 4.
Knowledge Modelling in OCML

This chapter provides an overview of OCML, the modelling language I have used for formalizing the components of the parametric design library and for building the various application models described in chapter 9. In this chapter I illustrate the philosophy underlying OCML, describe the main modelling constructs it provides and compare it to other modelling languages.

4.1. INTRODUCTION

This chapter provides an overview of OCML[1], the language I have used for modelling the components of the parametric design library and for building the various application models described in chapter 9. OCML was originally developed in the context of the VITAL project (Shadbolt et al., 1993) to provide operational modelling capabilities for the VITAL workbench (Domingue et al., 1993). Over the years the language has undergone a number of changes and improvements and in what follows I will provide an overview of the current version of the language (v5.1), illustrate its underlying philosophy and compare it to alternative knowledge modelling languages. Moreover, I will also illustrate a subset of the application development interface supporting the construction of OCML models. This interface consists of a number of Lisp macros and functions which can be used for retrieving and modifying OCML constructs, for evaluating *functional* and *control terms*, and for querying an OCML model.[2]

4.2. LANGUAGE TENETS

A number of ideas/principles have shaped the development of the OCML language. These are discussed in the following sections.

---

1 The acronym "OCML" stands for Operational Conceptual Modelling Language.

2 Graphical interfaces to OCML also exist. These include a web-based tool for collaborative construction of OCML models, which is currently under development (Zdrahal and Domingue, 1997).

4.2.1. Knowledge-level modelling support.

The main goal of OCML is to support knowledge-level modelling. In practice this role implies that OCML focuses on logical, rather than implementation-level primitives. Thus it provides mechanisms for expressing items such as relations, functions, rules, classes and instances, rather than arrays or hash tables. This approach is consistent with several other proposals for knowledge modelling (Newell, 1982; Gruber, 1993; Fensel and Van Harmelen, 1994). In the case of an operational language such as OCML this approach causes severe limitations in the support that the language can offer for efficient execution of application models. This problem can be (partially) addressed by providing a good compiler and by adding extra logical mechanisms for efficient reasoning - e.g. procedural attachments (Weyhrauch, 1980). Thus, while it is possible to specify and prototype knowledge models in OCML, the language does not aim to support efficient delivery of applications.

4.2.2. Support for the TMDA modelling framework

OCML is meant to support both the specification of library components and the development of partial or complete application models, according to the TMDA framework illustrated in the previous chapter. In particular this requirement implies the provision of primitives supporting the specification of domain models, task and method components and mapping knowledge. Earlier versions of the language (Motta, 1995) provided support for task and method specification by means of special-purpose modelling constructs. The current version does away with these task and method-specific constructs and only provides a basic set of domain modelling facilities; the extension to the language required to specify tasks and methods is then defined as a particular representation ontology (the *task-method* ontology)[3]. The advantage of this approach is that it separates the core set of logical primitives (the OCML kernel) from additional, framework-specific epistemological commitments. Thus, only special-purpose primitives for defining mapping knowledge are required, in addition to 'standard' domain modelling capabilities.

4.2.3. Compatibility with emerging standards

The development of the OCML language has been driven by several pragmatic considerations. An important one concerns the compatibility with established or emerging standards. In particular, OCML includes domain modelling facilities which closely mirror a significant subset of those provided by Ontolingua (Gruber, 1993). This property allows easy translation between OCML and Ontolingua. Moreover, these

---

[3] This ontology will be described in the next chapter.

capabilities allow Ontolingua users to use OCML as a kind of 'operational Ontolingua' providing theorem proving and function evaluation facilities for Ontolingua constructs. Such facilities are interactive and therefore support incremental model construction, rather than the 'batch mode' style of interaction associated with the translation approach used for operationalizing Ontolingua models in LOOM (Mac Gregor, 1991) or in other languages.

OCML can also be used for KADS-style modelling: the relevant modelling constructs required for representing KADS interpretation models can be defined by configuring the task-method ontology discussed in the next chapter for the KADS framework.

4.2.4. Integration of formal/informal/operational modelling

Different users require different modelling styles. Moreover, the same user may require different kinds of modelling support at different stages of the KBS development process. For instance, it is quite common to develop an informal application model first and formalize and operationalize it at a later stage - see for instance (Motta et al., 1994a; 1996). Therefore it is important to provide a language (or an integrated set of languages) which can support various modelling styles, such as formal, informal and operational.

Operational modelling is supported in OCML by providing interpreters for evaluating control and functional terms as well as theorem proving facilities. OCML can also be used for formal specifications: formal semantics can be given to functional terms and logical expressions by translating them to the equivalent Ontolingua expressions[4].

Finally, informal modelling is supported by means of a graphical notation and the provision of pseudo-OCML code[5].

4.2.5. Support for quick prototyping of knowledge models

An important principle underlying the design of OCML concerns the provision of facilities supporting rapid prototyping of knowledge models. This stance is grounded on both fundamental and pragmatic reasons. The former are related to the experimental nature of AI and, in particular, knowledge-based systems. Although much knowledge engineering work over the past decade - e.g. the KADS (Wielinga et al., 1992a) and VITAL (Shadbolt et al., 1993) projects - can be seen as a reaction to the rapid prototyping approach dominating AI in the previous decade, the complexity of AI problems and the

---

[4] No formal semantics is provided for the control language. Moreover, OCML also includes some extra-logical facilities, such as procedural attachments.

[5] This however is not used in this work and therefore will not be discussed here.

exploratory nature of much AI programming[6] makes AI systems much less amenable then conventional programs to a structured development approach based on a rigid separation between a formal, non-executable specification and an implementation. Non-executable specifications are of limited utility when dealing with problems which have non-polynomial complexity. As discussed in chapter 1, AI problems are typically ill-defined, which means that problem solvers necessarily need to search for a solution. Search is reduced by acquiring knowledge about the search space. This knowledge can be acquired either by doing (i.e. by navigating the search space) or by being told (i.e. by eliciting the relevant problem solving knowledge). In both situations quick prototyping is crucial to support knowledge acquisition in a situated problem solving context[7].

Incidentally, while executable specifications are especially important for KBS, similar arguments have been raised also in the software engineering field. Fuchs (1992) points out that early validation is crucial to improve the problem of software reliability and refutes the argument made by Hayes and Jones (1989) that executable specifications over-constrain the space of possible software designs. In particular Fuchs argues that executable, logic-based, declarative languages can adequately express functionalities at a level of abstraction similar to that of non-executable specification languages. It is interesting to note that Fuchs emphasizes that the key to maintain a high level of

---

[6] Such 'exploratory nature' of AI programming is the main reason for the curious phenomenon that happens when an AI problem is given a solution. At that point it is common to hear sceptics pointing out that the problem in question was not an AI problem in the first place, or that the particular solution is not an instance of AI. In my view there is a simple, generic explanation for this phenomenon. AI is about navigating large search spaces and understanding their structure. Once a search space has been explored and we know how to reach solutions, then it is natural to stop perceiving the problem as AI. The debate surrounding Deep Blue provides a typical instance of the phenomenon. For years one of the main goals of AI was to produce a chess program capable of beating the world champion. Now that the goal has been reached (to a limited extent), then the goal itself ceases to be AI. In a (rather perverse) sense this is true. Once Deep Blue is able to manage the complexity of the search space involved in chess games, then the problem loses its connotation of "decision-making under uncertain conditions", which is the defining feature of AI.

[7] For instance it is interesting to note that several contributions to the Sisyphus-II initiative (Schreiber and Birmingham, 1996) report 'discoveries' related to the nature of the domain knowledge and the behaviour of the problem solver, which emerged only after the implementation of the end system. And this (fairly typical) phenomenon emerged in the context of an application, the VT elevator design problem (Yost and Rothenfluh, 1996), which has been extensively analysed and reconstructed several times!

abstraction within an executable specification language is that this should be declarative and able to search. In AI in general and KBS in particular this property is crucial, not just to formalize a problem but also to model the problem solving behaviour of any feasible implementation.

Therefore, it is, in my view, essential for a knowledge modelling language to be able to support rapid prototyping and incremental development and testing. To this purpose OCML provides operational modelling support and extends the language with non-logical facilities such as procedural attachments, which can be used for carrying out efficient proofs or evaluations, or for interfacing to pre-existing code. A simple example of the role of procedural attachments can be seen in the case of basic modelling primitives for list manipulation and numeric computation. While the relevant ontologies can provide formal specifications of numbers and lists, when prototyping a model it is convenient to extend a formal model of arithmetic with attachments which can effectively compute the result of numerical calculations.

4.3. TYPES OF CONSTRUCTS IN OCML

OCML supports the specification of three types of constructs: *functional* and *control terms*, and *logical expressions*.

4.3.1. Functional terms

A functional term - in short, a term - specifies an object in the current domain of investigation. A functional term can be a *constant*, a *variable*, a *string*, a *function application* or can be constructed by means of a special *term constructor*. This can be one of the following: `if`, `cond`, `the`, `setofall`, `findall`, `quote` and `in-environment`[8] - see appendix 1 for a description of the semantics of these terms. Variables are represented as Lisp symbols beginning with a question mark, e.g. ?x is a variable. Strings are sequences of characters enclosed in double quotes, e.g. `"string"`. A function application is a term such as (*fun* {*fun-term*}*), where *fun* is the name of a function and *fun-term* a functional term. Functions are defined by means of the Lisp macro `def-function`, which is described in section 4.4.2.

Functional terms are evaluated by means of the OCML function interpreter, which is described in detail in appendix 1.

4.3.1.1. Control terms

Modelling problem solving behaviour involves more than making statements and describing entities in the world. Control terms are needed to specify actions and describe

---

[8] In what follows I will use `this font` to refer to expressions in the OCML language.

the order in which these are executed. OCML supports the specification of sequential, iterative and conditional control structures by means of a number of control term constructors, such as `repeat`, `loop`, `do`, `if`, and `cond`[9]. A Lisp macro, `def-procedure`, makes it possible to label parametrized control terms - i.e. to define *procedures*. Control terms are evaluated by means of a control interpreter. This is described in appendix 1.

4.3.1.2. Logical expressions

OCML also provides the usual machinery for specifying logical expressions. The simplest kind of logical expression is a *relation expression*, which has the form (*rel* {*fun-term*}*), where *rel* is the name of a relation and *fun-term* is a functional term. More complex expressions can be constructed by using the usual operators - `and`, `or`, `not`, `=>`, `<=>` - and quantifiers - `forall` and `exists`. Operational semantics is provided for all operators and quantifiers. Relations are defined by means of the Lisp macro `def-relation`, which is described in section 4.4.1.

4.4. BASIC DOMAIN MODELLING IN OCML

In the previous section I have introduced the three types of constructs which are supported by OCML. In this section I will go down at a more fine-grained level of description and I will illustrate the various primitives which are provided in OCML to support the specification of logical expressions, functional and control terms. In particular, OCML provides mechanisms for defining *relations*, *functions*, *classes*, *instances*, *rules* and *procedures*.

4.4.1. OCML relations

Relations allow the OCML user to define labelled n-ary relationships between OCML entities. Relations are defined by means of a Lisp macro, `def-relation`, which takes as arguments the name of a relation, its *argument schema*, optional documentation and a number of *relation options*. An argument schema is a list (possibly empty) of variables. Relation options play two roles, one related to the formal semantics of a relation, the other to the operational nature of OCML. These roles are discussed in the next two sections.

---

[9] The careful reader will have noticed that I have already mentioned `if` and `cond` when discussing functional terms. In fact, `if` and `cond` can be used to construct both functional and control terms. However this does not cause any problem. For instance, if an `if` construct is encountered when expecting a functional term then an error will be generated if control terms are used in the body of the `if`. The opposite however is not true: functional terms can always be used in place of control terms - see appendix 1 for more details on the behaviour of the control interpreter.

4.4.1.1. *Relation specification options*

From a formal semantics point of view the purpose of a relation option is to help characterize the extension of a relation. Table 4.1. shows the relation options which can be used to provide formal relation specifications and, for each option, informally describes its semantics. A formal semantics to these options can be given in terms of the homonymous Ontolingua constructs.

Relation Option	Role in Specification
`:iff-def`	Specifies both sufficient and necessary conditions for the relation to hold for a given set of arguments.
`:sufficient`	Specifies a sufficient condition for the relation to hold for a given set of arguments.
`:constraint`	Specifies an expression which follows from the definition of the relation and must be true for each instance of the relation.
`:def`	This is for compatibility with Ontolingua: it specifies a constraint which is also meant to provide a partial definition of a relation.
`:axiom-def`	A statement which mentions the relation to which it is associated. It provides a mechanism to associate theory axioms with specific relations.

Table 4.1. Relation specification options in OCML

4.4.1.2. *Operationally-relevant relation options*

Relation options also play an operational role. Specifically, some relation options support *constraint checking* over relation instances while others provide *proof mechanisms* which can be used to find out whether or not a relation holds for some arguments. Table 4.2 lists the relation options which are meaningful from an operational point of view and informally describes their relevance to constraint checking and theorem proving.

Relation Option	Supports constraint checking	Provides proof mechanism
:sufficient	No	Yes
:prove-by	No	Yes
:lisp-fun	No	Yes
:iff-def	Yes	Yes
:constraint	Yes	No
:def	Yes	No

Table 4.2. Operationally-relevant relation options in OCML

As shown in the table, constraint checking is supported by the following keywords: `:constraint`, `:def` and `:iff-def`. While these have different model-theoretic semantics - see table 4.1 - from a constraint checking point of view they are equivalent. They all specify an expression which has to be satisfied by each known instance of the relevant relation.

The relation options `:iff-def`, `:sufficient`, `:prove-by` and `:lisp-fun` provide mechanisms for verifying whether or not a relation holds for some arguments. The first two - `:iff-def` and `:sufficient` - also play a specification role - see table 4.1. The others - `:prove-by` and `:lisp-fun` - only play an operational role.

Both `:iff-def` and `:sufficient` indicate logical expressions which can be used to prove whether some tuple is an instance of a relation. From a theorem proving point of view there is an important difference between them. Let's suppose we are trying to prove that a tuple, say T, satisfies a relation, say R. If a `:sufficient` condition is tried and failed, the OCML proof system will then search for alternative ways of proving that T satisfies R. If an `:iff-def` condition is tried and failed, then no alternative proof mechanism will be attempted.

The relation options `:prove-by` and `:lisp-fun` are meant to support rapid prototyping and early validation by providing efficient mechanisms for checking whether a tuple satisfies a relation. The difference between `:prove-by` and `:lisp-fun` has to do with the expressions which are used as values to the two options: `:prove-by` points to a logical expression, `:lisp-fun` to a non-logical one (specifically a Lisp function).

The box below provides an example of how the various types of relation options can be used concurrently to specify a relation and to support constraint checking and efficient proofs. The relation `has-value`, shown below, associates a *design parameter* to its value

in a *design model*. The definition specifies that ?v is the value of a parameter, ?p, in a design model, ?dm, if and only if the pair (?p . ?v) is an element of ?dm (see chapter 6 for more details on the parametric design task ontology). In addition it also specifies the constraint that the value of a parameter has to be a member of its value range, if this has been specified. Finally, the definition includes a :prove-by option whose value is an expression which can be used for verifying whether the relation is satisfied for a triple, (?p ?v ?dm). This expression provides an efficient proof method (by weakening the :iff-def statement which formally defines relation has-value) but does not contribute to the specification.

```
(def-relation HAS-VALUE (?p ?v ?dm)
  "Parameters have values w.r.t a particular design model"
  :iff-def (and (parameter ?p)
                (design-model ?dm)
                (element-of (?p . ?v) ?dm))
  :constraint (or (and (exists ?vr
                               (has-value-range ?p ?vr))
                       (element-of ?v ?vr))
                  (not (exists ?vr
                               (has-value-range ?p ?vr))))
  :prove-by (element-of (?p . ?v) ?dm))
```

4.4.1.3. A meta-option for non-operational specifications

As shown above, OCML provides a number of relation options (specifically: :iff-def, :sufficient, :def and :constraint), which play both a specification and an operational role. However, in some cases we might want to use a keyword only for specification and not operationally, for instance when we know that the value of the keyword in question is a non-operational expression. To cater for these situations, OCML provides a special meta-keyword, :no-op, which can be used to indicate that the enclosed relation option only plays a specification role. An example of its use is shown by the definition of relation range, which is shown below. In the example the keyword :no-op is used to indicate that the :iff-def specification of the relation is not operational - in particular it is not normally feasible to test the range of a function on all its possible inputs![10]

[10] Sharp, Ontolingua-aware readers may have induced from the definition of the relation range that, in contrast with Ontolingua, OCML does not considers functions as relations. This has to do with the operational nature of the language: functions are dealt with by the OCML interpreter, relations by the proof system.

```
(def-relation RANGE (?f-r ?relation)
  "The range of a function or a binary relation is a relation which is
   true for any possible output of the function or second argument of
   the binary relation"
  :no-op (:iff-def (or
                    (and (function ?f-r)
                         (forall (?args ?result)
                                 (=> (= (apply ?f-r ?args) ?result)
                                     (holds ?relation ?result))))
                    (and (binary-relation ?f-r)
                         (forall (?x ?y)
                                 (=> (holds ?f-r ?x ?y))
                                     (holds ?relation ?y))))))
```

In the above definition it is worth highlighting the use of the special meta-relation `holds`. An expression such as (`holds` <r> <arg$_1$>....<arg$_n$>) is satisfied if and only if the expression (<r> <arg$_1$>....<arg$_n$>) is satisfied. Thus `holds` has variable arity: it can take one or more arguments. In particular the number of additional arguments in a `holds` statement reflects the arity of the relation passed as first argument. The relation `holds` has a 'special status' because it is the only relation with variable arity supported by OCML.[11]

4.4.1.4. OCML relations: summing up

The set of relation options discussed here aims to provide a flexible and versatile range of modelling constructs supporting various styles of modelling. While the emphasis is on operational modelling, OCML also supports formal specification. Moreover, it provides facilities for integrating a specification with efficient proof mechanisms.

4.4.2. OCML functions

A function defines a mapping between a list of input arguments and its output argument. Formally functions can be characterized as a special class of relations, as in KIF (Genesereth and Fikes, 1992). However, in operational terms there is a significant difference between a function and a relation: functions are applied to ground terms to generate function values; relation (i.e. logical) expressions can be asserted or queried. Thus, in accordance with the operational nature of OCML, functions are distinguished from relations.

---

[11] This constraint is only a limitation of the current implementation of the language: in principle there is no reason why variable-arity relations should not be supported. Moreover, in contrast with relations, OCML functions can have variable arity. To specify that a function can take an indefinite number of arguments OCML uses the same convention as Lisp: the symbol `&rest` is used before the last argument of a function argument schema.

Functions are defined by means of a Lisp macro, `def-function`. This takes as argument the name of a function, its argument list, an optional variable indicating the output argument (as in Ontolingua this is preceded by an arrow, ->), optional documentation and zero or more *function specification options*. These are `:def`, `:constraint`, `:body` and `:lisp-fun`.

The option `:constraint` provides a way to constrain the *domain* (i.e. the set of possible inputs) of a function. It specifies a logical expression which must be satisfied by the input arguments of the function. The `:def` option indicates a logical expression which 'defines' the function. This expression should be predicated over both (some) input arguments and the output variable. Operationally, the expression denoted by the `:constraint` option provides a mechanism for testing the feasibility of applying a function to a set of arguments; the expression denoted by `:def` provides a mechanism for verifying that the output produced by a function application is consistent with the formal definition of the function.[12]

Finally, the options `:body` and `:lisp-fun` provide effective mechanisms for computing the value of a function. The former specify a functional term which is evaluated in an environment in which the variables in the function schema are bound to the actual arguments. The latter makes it possible to evaluate an OCML function by means of a procedural attachment, expressed as a Lisp function. The arity of this Lisp function should be the same as that of the associated OCML function.

```
(def-function filter (?l ?rel) -> ?sub-l
  "Returns all the elements in ?l which satisfy ?rel"
  :def (and (unary-relation ?rel)
            (list ?l))
            (list ?sub-l)
            (=> (and (member ?x ?sub-l)
                     (holds ?rel ?x))
                (member ?x ?l)))
  :body (if (null ?l)
          ?l
          (if (holds ?rel (first ?l))
            (cons (first ?l)
                  (filter (rest ?l) ?rel))
            (filter (rest ?l) ?rel))))
```

The above definition shows an example of the use of `def-function`. The OCML function `filter` takes as arguments a list, `?l`, and a unary relation, `?rel`, and returns the elements of `?l` which satisfy `?rel`. As illustrated by the definition, the `:def` option provides a declarative way of specifying a function; the option `:body` an effective way of computing its value, for a given set of input arguments.

---

12 These constraint-checking mechanisms can be switched off, if they are not required.

4.4.3. OCML classes

OCML also supports the specification of classes and instances and the inheritance of slots and values through *isa hierarchies.*

Classes are defined by means of a Lisp macro, `def-class`, which takes as arguments the name of the class, a list (possibly empty) of superclasses, optional documentation, and a list of *slot specifications*, as illustrated by the definitions in the next box. These show a number of classes taken from the domain model for the Sisyphus-I office allocation problem (Linster, 1994; Chapter 9).

```
(def-class YQT-member ()
  ((has-project :type project)
   (smoker :type boolean :cardinality 1)
   (hacker :type boolean :cardinality 1)
   (works-with :type YQT-member)
   (belongs-to-group :type research-group :value yqt)))

(def-class researcher (YQT-member))

(def-class secretary (YQT-member))

(def-class manager (YQT-member))
```

OCML provides support for the usual slot specification machinery which is found in frame-based languages. Specifically, it provides the following slot options.

- **`:value`**. A value which is inherited by all instances of a class.
- **`:default-value`**. A value which is inherited by all instances of a class, unless overridden by other values.
- **`:type`**. The value of this option should be a class, say C. This option specifies that all values of the associated slot should be instances of C.
- **`:max-cardinality`**. The maximum numbers of slot values allowed for a slot.
- **`:min-cardinality`**. The minimum numbers of slot values required for a slot.
- **`:cardinality`**. The numbers of slot values required for a slot. This option subsumes both `:min-cardinality` and `:max-cardinality`.
- **`:documentation`**. The value of this option is a string providing documentation for a slot.
- **`:inheritance`**. The inheritance mechanism used for dealing with default values. If `:merge` is used, then all default values inherited from different ancestors are collected. If `:supersede` is used, then default values inherited from more specific ancestors override those inherited from more generic ones - see appendix 1 for more details on the inheritance mechanism in OCML.

4.4.4. OCML instances

Instances are simply members of a class. An instance is defined by means of `def-instance`, which takes as arguments the name of the instance, the *parent* of the instance (i.e. the most specific class the instance belongs to), optional documentation and a number of slot-value pairs. An example of instance definition, taken from the Sisyphus-I domain model, is shown in the box below.

```
(def-instance harry_c researcher
  ((has-project babylon)
   (smoker no)
   (hacker yes)
   (works-with jurgen_l thomas_d)))
```

In particular the above definition shows that a slot can have multiple values. In this case `harry_c` works both with `jurgen_l` and `thomas_d`.

4.4.5. Object-oriented and relation-oriented approaches to modelling

When describing classes and instances I made use of standard object-oriented terminology and talked about slots having values and instances belonging to classes. This object-centred approach is in a sense orthogonal to the relation-centred one which I used when discussing relations and logical expressions. The former focuses on the entities populating a model and then associates properties to them; the latter centres on the type of relations which characterize a domain and then uses these to make statements about the world. These two approaches to modelling/representation have complementary strengths and weaknesses and for this reason they are often combined in knowledge representation and modelling languages, to provide *hybrid* formalisms (Fikes and Kehler, 1985; Yen et al., 1988).

In the context of a knowledge modelling language (rather than a knowledge representation one) the main advantage gained from combining multiple paradigms is one of flexibility. Both object-oriented and relation-oriented approaches provide conceptual frameworks which make it possible to impose a view over some domain. The choice between one or the other can be made for ideological or pragmatic reasons - e.g. whether the target delivery environment is a rule-based shell or an object-oriented programming environment. In the specific context of an operational modelling language, such as OCML, another benefit, which is gained by providing support for both object-oriented and relation-oriented modelling, is that these approaches are naturally associated with particular types of inferences. Object-orientation provides the structure for inheritance and automatic classification; relation-orientation is normally associated with constraint-based and rule-based reasoning.

While the integration of multiple paradigms provides the aforementioned benefits, when describing or interacting with a knowledge model, it is useful to abstract from the various modelling paradigms and inference mechanisms integrated in the model and characterize it at a uniform level of description. Specifically, in accordance with a view of knowledge as a *competence-like* notion (Newell, 1982), it is useful to decouple the level at which we describe what an agent knows from the level at which we describe how the agent organizes and infers knowledge. Such an approach is used - for instance - in the Cyc system (Lenat and Guha, 1990), which integrates multiple knowledge representation techniques (the *heuristic* level), but provides a uniform interface to the Cyc knowledge base (the *epistemological* level). A similar approach is also followed by Levesque (1984), which describes a logic-based query language which can be used to communicate with a knowledge base at a functional level, independently from the data and inference structures present in the knowledge base.

In particular, in the case of OCML, this generic idea of providing a uniform level of description to a hybrid formalism has been instantiated by providing a *Tell-Ask* interface (Levesque, 1984), which use logical expressions (i.e. a relation-oriented view) when modifying or querying an OCML model, independently of whether the query in question concerns a class, a slot, or a 'ordinary' relation. The key to this integrated view is the fact that classes and slots are themselves relations; classes are unary relations and slots are binary ones. In addition to supporting a generic Tell-Ask interface, this property makes it possible to provide 'rich' specifications of classes and slots. In particular, because these are relations, it is possible to characterize them by means of the relation options discussed in section 4.4.1. For instance, the definition below specifies the class of empty sets in terms of an `:iff-def` relation option.

```
(def-class EMPTY-SET (set) ?set
   :iff-def (not (exists ?x (element-of ?x ?set))))
```

4.4.6. The generic Tell-Ask interface

4.4.6.1. Tell: a generic assertion-making primitive

OCML provides a generic assertion-making primitive, `tell`, which provides a uniform mechanism for asserting *facts*[13], independently of whether these refer to slot-filling assertions, new class instances, or simply relation instances. For example I can use `tell` to add a new value to the list of projects carried out by `harry_c` as follows.

---

[13] Formally a *fact* (or *assertion*) is a *ground relation expression*. A relation expression is an expression such as (<r> <arg$_1$>....<arg$_n$>), where <r> is a relation and arg$_i$ is a functional term. A ground expression (or term) is an expression (or term) which does not contain variables.

```
? (tell (has-project harry_c mlt))

(HAS-PROJECT HARRY_C MLT)
```

Analogously I can add a new instance of class researcher simply by stating:

```
? (tell (researcher mickey_m))

(RESEARCHER MICKEY_M)
```

4.4.6.2. Ask: a generic query-posing primitive

The relation-centred view makes it possible to examine the contents of an OCML model simply by asking whether a logical statement is satisfied. The OCML proof system will then carry out the relevant inference and retrieval operations, depending on whether the relation being queried is a slot, a class, or an 'ordinary' relation. The process is however transparent to the user. For instance, I can find out about the projects in which `harry_c` is involved - after the assertion shown above these are now `babylon` and `mlt` - by using the Lisp macro `ask` to pose the query `(has-project harry_c ?c)`. The resulting interaction with the OCML proof system is shown below.

```
? (ask (has-project harry_c ?c))

Solution: ((HAS-PROJECT HARRY_C BABYLON))

More solutions? (y or n) y

Solution: ((HAS-PROJECT HARRY_C MLT))

More solutions? (y or n) y

No more solutions
```

This uniform, relation-centred view over an OCML models also provides a way to index inferences. For instance, when answering the above query, the OCML proof system will first retrieve and order all inference mechanisms applicable to a query of type `has-project` - e.g., these might include assertions of type `has-project`, relation options associated with `has-project` and the relevant *backward rules*[14] - and will then try these in sequence, to generate one or more solutions to the query (more details on the OCML proof system are given in appendix 1).

---

[14] See section 4.4.8.1 for a description of OCML backward rules.

4.4.7. OCML procedures

Procedures define actions or sequences of actions which cannot be characterized as functions between input and output arguments. For example, the procedure below defines the sequence of actions needed to set the value of a slot. These include a `unassert` statement, which removes any existing value from the slot, and a `tell` statement, which adds the new value. Both `tell` and `unassert` are procedures. The former takes a ground logical expression and adds it to the current model. The latter takes a relation expression and removes from the current model all assertions which match it. Note that in accordance with the uniform view of a knowledge model, slot changes are carried out by means of generic assertion and deletion operations (i.e. in terms of `tell` and `unassert`).

```
(def-procedure SET-SLOT-VALUE (?i ?s ?v)
  :constraint (and (instance-of ?i ?c)
                   (slot-of ?s ?c))
  :body (do
          (unassert (list-of ?s ?i ?any))
          (tell (list-of ?s ?i ?v))))
```

4.4.8. Rule-based reasoning in OCML

4.4.8.1. *Backward rules*

OCML also supports the specification of *backward* and *forward* rules. A backward rule consists of a number of *backward clauses*, each of which is defined according to the following syntax:

backward-clause ::= (*rel-expression* {`if` {*log-expression*}+})[15]

[15] I use the following notational conventions when describing the syntax of OCML constructs. Braces, { and }, are used to indicate that the enclosed item is optional. For instance, the notation {x} means that x may or may not be present but, if it is present, it can only appear once. The notation {x}+ means that x can appear 1 or more times (i.e. it must appear at least once), while the notation {x}* indicates that x can appear 0 or more times. Moreover, I use square brackets within braces in situations in which a number of alternatives for a *non-terminal* item are possible, but each alternative can be used only once. For example, let's consider a non-terminal item, x, for which alternatives a and b are possible. In this context the notation {[x] y}* indicates that any number of xy sequences are possible, but a or b can only appear once (in practice this means that we can have between 0 and 2 sequences). Braces can also be nested. For instance, the notation {x {y}} means that both x and y are optional but y can only appear if x does. Finally, I use italics to denote non-terminal items and a vertical bar, |, to indicate mutually exclusive alternatives.

Each backward clause specifies a different *goal-subgoal decomposition.* When carrying out a proof by means of a backward rule the OCML interpreter will try to prove the relevant goal by firing the clauses in the order in which these are listed in the rule definition. As in Prolog depth-first search with chronological backtracking is used to control the proof process.

Both semantically and operationally a backward chaining rule is the same as a `:sufficient` relation option: they both provide an expression which is sufficient to verify that a tuple holds for a relation. Thus, one might wonder whether rules are needed at all. In practice the advantage of including backward rules in the language is that these provide a modular mechanism for refining existing (possibly generic) relation specifications, for instance in cases where application-specific knowledge is needed to complete the specification of a relation. To clarify this point let's consider an example taken from the KMI office allocation problem (see chapter 9 for an extensive analysis of this application).

The relation `has-value-range` is defined in the parametric design task ontology to characterize the set of possible values which can be assigned to a design parameter - see chapter 6. When building an application model the generic `has-value-range` specification is usually refined to characterize exactly the type constraints over the values of each parameter. A modular way to do this is to refine the definition of the relation by means of the appropriate backward chaining rules. The rules shown below fulfil this purpose for two of the classes of parameters present in the KMI office allocation domain: professors and secretaries.

```
(def-rule has-value-range-1
  ((has-value-range-gen ?m ?l)
   if
   (professor (domain-reference ?m))
   (= ?l (setofall ?r
                   (double-a-type-room ?r usable yes)))))

(def-rule has-value-range-2
  ((has-value-range-gen ?m ?l)
   if
   (secretary (domain-reference ?m))
   (= ?l (setofall ?r
                   (and (room ?r size ?n usable yes)
                        (> ?n 1)
                        (close-to ?r kmi-entrance))))))
```

4.4.8.2. Forward rules

OCML also allows the user to define forward rules. A forward rule comprises zero or more antecedents and one or more consequents. Antecedents are restricted to relation expressions, while any logical expression can be a consequent. When a forward rule is executed, OCML treats each consequent as a goal to be proven and attempts to prove

them, until one fails. This mechanism makes it possible to integrate data-driven and goal-driven reasoning and to specify arbitrarily complex right hand sides.

A special operator, `exec`, is provided to allow OCML users to introduce control (and therefore functional) terms in the right hand side of a rule. In particular, two useful procedures are `tell`, to assert new facts, and `output`, to produce output. A simple example showing how to use these in a forward chaining rule is given below.

```
(def-rule foo
  (has-project ?x ?y)
  then
   (exec (tell (project-covered-by ?y ?x)))
   (exec (output "has project ~S ~S" ?x ?y)))
```

While forward rules can be useful in a number of situations when building application models (e.g. to define *watchers*, which are triggered whenever some situation arises in a knowledge base), they are not essential to the model building process. The reason for this is that knowledge-level modelling is mainly about constructing definitions, while forward-chaining rules are about behaviour. Thus they can be used in place of procedures to describe behaviour but they cannot replace relation or function specification constructs.

4.5. FUNCTIONAL VIEW OF OCML

A functional view of a knowledge representation system focuses on the services the system provides to the user (Levesque, 1984; Brachman et al., 1985). Basically, there are three kinds of services provided by OCML: i) operations for extending/modifying a model; ii) interpreters for functional and control terms; and iii) a proof system for answering queries about a model. Extensive details on the model extension/modification facilities provided by OCML were given in earlier sections. The interpreters and the proof system are described in detail in appendix 1.

4.6. MAPPING

The constructs presented so far provide extensive support for domain modelling. In order to fully support the TMDA framework additional constructs are needed, for task and method specification and for mapping entities at the task/method level to entities at the domain level. As already said, the conceptual vocabulary required to model tasks and methods is not hardwired in the language but is defined as a specialized ontology. This will be presented in the next chapter; in the next sections I will discuss the two kinds of mapping constructs supported by OCML: *relation mapping* and *instance mapping*.

4.6.1. Instance mapping

In figure 3.7 I showed a simple example in which a class of concepts at the task/method level (parameter) is mapped to a class of concepts at the domain level (YQT-member). This is a very common situation when developing systems through reuse: a problem solving, domain-independent model imposes a particular view over a set of domain concepts (Fensel and Straatman, 1996).

A simple case is one in which a domain view is constructed by direct association of task level concepts to domain level concepts. For instance, a parametric design view over the Sisyphus-I domain can be imposed simply by creating parameter instances and associating them to YQT members. Thus, the set of parameters and the set of YQT members are associated but kept distinct. This solution is appealing for two reasons: it supports reuse of modular components and does not mix two different types of concepts. In particular, parameters and YQT members maintain different set of properties and different semantics, thus avoiding a situation in which a design parameter has a wife and a YQT member has a value range!

Instance mapping is supported in OCML by means of the Lisp macro `def-upward-class-mapping`. This takes the names of two classes as arguments and associates each instance of the first class to a purpose-built instance of the second class. By default the relation `maps-to` is used to associate the task level instance to the domain level one.

In the Sisyphus-I application model we should then state:

```
(def-upward-class-mapping yqt-member yqt-parameter)
```

The above form iterates over each instance of class `yqt-member`, say I, creating a new instance of class `yqt-parameter` and associating this to I. The relation used to model the association is called `maps-to`. Thus, if we now ask for the mapping between parameters and YQT members we get the following results.

```
? (ask (maps-to ?z ?x)t)

Solution: ((MAPS-TO YQT-PARAMETER-EVA_I EVA_I))

Solution: ((MAPS-TO YQT-PARAMETER-MONIKA_X MONIKA_X))

Solution: ((MAPS-TO YQT-PARAMETER-ULRIKE_U ULRIKE_U))

Solution: ((MAPS-TO YQT-PARAMETER-UWE_T UWE_T))

Solution: ((MAPS-TO YQT-PARAMETER-JOACHIM_I JOACHIM_I))

Solution: ((MAPS-TO YQT-PARAMETER-HANS_W HANS_W))

Solution: ((MAPS-TO YQT-PARAMETER-MICHAEL_T MICHAEL_T))

Solution: ((MAPS-TO YQT-PARAMETER-ANGY_W ANGY_W))

Solution: ((MAPS-TO YQT-PARAMETER-JURGEN_L JURGEN_L))

Solution: ((MAPS-TO YQT-PARAMETER-KATHARINA_N KATHARINA_N))

Solution: ((MAPS-TO YQT-PARAMETER-THOMAS_D THOMAS_D))

Solution: ((MAPS-TO YQT-PARAMETER-HARRY_C HARRY_C))

Solution: ((MAPS-TO YQT-PARAMETER-ANDY_L ANDY_L))

Solution: ((MAPS-TO YQT-PARAMETER-MARC_M MARC_M))

Solution: ((MAPS-TO YQT-PARAMETER-WERNER_L WERNER_L))
```

The advantage of this solution is that parameters and YQT members maintain their separate identities, as shown in the next box.

```
? (describe-instance 'YQT-PARAMETER-WERNER_L)

Instance YQT-PARAMETER-WERNER_L of class YQT-PARAMETER

HAS-VALUE-RANGE: (C5-123 C5-122 C5-121 C5-120 C5-119 C5-117 C5-116 C5-
113 C5-114 C5-115)

? (describe-instance 'WERNER_L)

Instance WERNER_L of class RESEARCHER

HAS-PROJECT: RESPECT
SMOKER: NO
HACKER: YES
WORKS-WITH: ANGY_W, MARC_M
GROUP: YQT
```

Formally, a mapping can be characterized as an association between an object, say `o`, and its meta-object, `m-o`, so that the entity denoted by the entity denoted by `m-o` is the same as the entity denoted by `o` (Genesereth and Nilsson, 1988). This approach is used in the definition of relation `maps-to`, shown below. This definition makes use of the function

`denotation`, defined in the Ontolingua base ontology, which specifies a non-operational association between an object and its denotation.

```
(def-relation maps-to (?x ?y)
 "This relation allows the user to specify an association between
  an object at the task layer and one at the domain layer.
  Formally ?y denotes the object denoted by the object denoted by ?x"
 :no-op (:iff-def (= ?y (denotation ?x))))
```

4.6.2. Relation mapping

Instance mapping works only in those cases in which imposing a view over a domain can be reduced to creating task-level 'mirror images' for a finite number of domain-level objects. A more general scenario is one in which there is some relation defined at the task/method level which needs to be reflected to the domain level in a dynamic fashion. A well known example is that in which domain concepts or statements are viewed as hypotheses at the problem solving level. This association is typically dynamic, given that hypotheses are considered as such only for a particular time-slice of the problem solving process. These situations can be modelled in OCML by means of *relation mappings.*

A relation mapping provides a mechanism to associate rules and procedures to a relation, say R, so that when a query of type R is posed, or assertions of type R are made at the task/method level, these events can be *reflected* to the domain level. The purpose of these reflection actions is to ensure that the consistency between domain and task/method levels is maintained.

An example of an *upward relation mapping* is illustrated by the definition below, which is taken from an application model developed for the Sisyphus-I problem. The mapping is an upward one, in the sense that it is used *to lift* (van Harmelen and Balder, 1992) the office allocation statements existing at the domain level to the problem solving level. Specifically, the goal of this mapping is to associate the relation `current-design-model`, which is used by the parametric design problem solver to indicate the design model associated with the current design state, to the set of `in-room` assertions present in the current snapshot of the domain knowledge base. Note the use of relation `maps-to` to retrieve the parameter associated with each particular YQT member.

```
(def-relation-mapping current-design-model :up
   ((current-design-model ?dm)
    if
    (= ?dm (setofall (?p . ?v)
                     (and (in-room ?x ?v)
                          (maps-to ?p ?x))))))
```

An upward relation mapping ensures that when a task/method level relation is needed the relevant information is obtained from the domain level. Of course, problem solving is

also about inferring knowledge and retracting previously held assertions. Hence, OCML also supports *downward relation mappings*. These divide into two categories, `:add` and `:remove`. The former specifies a procedure which is activated when a new relation instance is asserted. The latter specifies a procedure which is activated when a relation instance is removed. In the case of relation `current-design-model` relation mappings are needed to ensure that when the design model considered by the problem solver is modified, the relevant changes are reflected onto the domain model - see definition below.

```
(def-relation-mapping current-design-model (:down :add)
  (lambda (?x)
    (do
      (unassert (in-room ?any-m ?any-r))
      (loop for ?pair in ?x
            do
            (if (maps-to (first ?pair) ?z)
              (tell (in-room ?z (rest ?pair))))))))
```

Finally, the definition below shows the `:remove` downward mapping associated with relation `current-design-model`: it simply removes the domain level assertions associated with the design model which is passed as argument to the relation instance being retracted.

```
(def-relation-mapping current-design-model (:down :remove)
   (lambda (?x)
     (loop for ?pair in ?x
           do
           (if (maps-to (first ?pair) ?z)
             (unassert (in-room ?z (rest ?pair)))))))
```

4.7. ONTOLOGIES

OCML also provides the basic support for constructing models out of pre-existing components. When an ontology is defined, say O, it is possible to specify which ontologies are included in O and as a result O will include all the definitions from its sub-ontologies. When conflicts are detected (e.g. the same concept is defined in two different sub-ontologies) a warning is issued. Primitives for loading and selecting ontologies are also provided.

By default all ontologies are built on top of the *OCML base ontology*. This comprises twelve sub-ontologies which include the basic definitions required to reason about basic data types (e.g. lists, numbers, sets and strings), the OCML system itself and the OCML frame representation. Specifically, the following sub-ontologies are provided:

- **Meta**. This ontology defines the concepts required to describe the OCML language. It includes constructs such as 'OCML expression', 'functional term',

'rule', 'relation', 'function', 'assertion', etc. This ontology is particularly important to construct reasoning components which can verify OCML models.

- **Functions**. Defines the concepts associated with functions, e.g. it includes relations such as `domain`, `range`, `unary-function`, `binary-function`, etc.
- **Relations**. Defines the various notions associated with relations. These include the universe and the extension of a relation, the definition of reflexive and transitive relations, partial and total orders, etc.
- **Sets**. This ontology defines the notions associated with sets, e.g. 'empty set', 'union', 'intersection', 'set partition', 'set cardinality', etc.
- **Numbers**. Defines the various concepts and operations required for reasoning about numbers and for performing calculations.
- **Lists**. Defines the concepts and operations associated with lists. For instance it includes classes such as `list` and `atom`; functions such as `first`, `rest` and `append`; and relations such as `member`.
- **Strings**. Specifies the concepts and operations associated with strings - e.g. `string`, `string-append`, etc.
- **Mapping**. This ontology defines the concepts associated with the mapping mechanism described earlier. It includes only three definitions: relation `maps-to` and functions `meta-reference` and `domain-reference`. The former takes a domain-level instance and returns the associated task/method level instance. The latter performs the inverse function.
- **Frames**. Defines the concepts associated with the frame-based representation used in OCML. It comprise classes such as `class` and `instance`; functions such as `direct-instances` and `all-slot-values`; relations such as `has-one` and `has-at-most`; and procedures such as `append-slot-value`.
- **Inferences**. The purpose of this ontology is to provide a repository for defining functions and relations supporting the specification of KADS-like inferences. So far only a few such inferences have been added to this ontology to support different types of selection and sorting.
- **Environment**. This ontology provides a kind of 'environmental support' for the construction of OCML models. It includes special operators like `exec`, which is used to invoke procedures from rules, and procedures such as `output`, which prints out a message.

- **Task-Method**. This ontology provides the concepts required to model tasks and problem solving methods, i.e. to support the construction of task and problem solving models.

This set of ontologies provides a rich modelling platform from which to build other ontologies and/or problem solving models. It is natural to compare the OCML and Ontolingua base ontologies. There are two aspects which distinguish these two sets of ontologies: their nature and their scope.

The first difference is related to the operational nature of OCML. The Ontolingua base ontology is not concerned with operationality and therefore includes many non-operational definitions. The OCML base ontology is concerned with providing support for the construction of operational models. As a result it attempts to minimize the number of non-executable specifications. A typical approach is to weaken a non-operational definition to make it executable. For instance let's consider the function `universe`. In the Ontolingua base ontology the universe of a relation is defined as the set of all objects for which the relation is *true*. Of course this is not an operational definition. However, we can provide a weaker version of universe, called `known-universe`, which returns the set of all entities which are part of a tuple satisfying the relation in question. This function can be either defined separately from `universe` or attached to it to provide an operational definition.

The second difference concerns the scope of the two base ontologies. Both the Ontolingua and OCML base ontologies provide a rich set of definitions for domain modelling. In order to comply with the requirements imposed by the TMDA framework, the OCML base ontology provides support also for specifying tasks and problem solving methods.

4.8. COMPARISON WITH OTHER LANGUAGES

In the previous section I compared the base ontologies provided by OCML and Ontolingua and emphasized that the differences between them have mainly to do with the conceptual requirements imposed by the TMDA framework on OCML and with its operational nature. If we compare OCML and Ontolingua purely as modelling languages, it is easy to see that their main difference has to do with the fact that while Ontolingua is concerned exclusively with ontologies - i.e. term specification - OCML aims to model behaviour as well. For this reason OCML also provides support for defining control terms. Apart from this aspect, the current version of OCML closely mirrors many of the constructs in Ontolingua. Thus, while OCML extends Ontolingua in various respects it is possible (if reductive) to view OCML as an environment for prototyping Ontolingua

models, thus moving away from the batch-oriented, translation-based operationalization model suggested for Ontolingua (Gruber, 1993).

The design philosophy underlying the KARL language (Fensel, 1995a) has many points in common with OCML. In particular both KARL and OCML are operational modelling languages and are therefore suitable for rapid prototyping of knowledge level models. Moreover, both KARL and OCML are part of comprehensive knowledge engineering frameworks providing methodological support for KBS development. Needless to say, there are also differences. KARL is an executable, formal specification language where the emphasis is on formalization: its main strength is the provision of a formal semantics for its modelling constructs. The design philosophy of OCML has more to do with pragmatic considerations: its main goal is to provide a flexible modelling environment able to support different approaches to modelling: rapid prototyping and structured development, executable and non-executable constructs, formal and informal specifications. The two languages also differ in terms of the modelling frameworks they employ: KARL is based on the KADS four-layer framework, while OCML (more precisely the task-method ontology) is based on the TMDA framework. An important difference is also that while the primitives in the KARL language closely reflect the KADS approach, the OCML kernel is approach-neutral. Its commitment to the TMDA framework is defined by means of the appropriate ontology. This approach has the advantage of flexibility: different modelling frameworks can then be supported through the specification of the relevant ontologies.

Recently a new version of KARL (*New-KARL*) has been proposed (Angele et al., 1996) which does away with the strong KADS-oriented approach used by KARL and focuses instead on the specification of task-method structures and ontology mappings. Thus New-KARL subscribes to a modelling framework which has much more in common with the OCML task ontology than the one underlying KARL. However, in contrast with OCML, New-KARL also 'hardwires' these task and method-centred primitives in the language itself, rather than in a particular ontology.

Other formal specification languages exist for KADS models - see (Fensel and van Harmelen, 1994) for an overview. While the formal details of these languages of course vary, similar conclusions to those drawn above can be reached when comparing them to OCML: these languages tend to emphasize formal aspects and are based on a KADS approach. OCML emphasizes operationality and flexibility and does not presuppose (although it can support) a KADS approach. Indeed, the original raison d'être for OCML was to provide an operational alternative to a formal specification language, $K_{BS}SF$ (Jonker and Spee, 1992), in the context of the VITAL workbench (Domingue et al., 1993).

Among the informal notations available for knowledge modelling the most notable is the CML language (Schreiber et al., 1994a), which supports ontological specifications and the construction of Common KADS models. CML supports the definition of various constructs, including concepts, attributes, tasks, methods, relations, structures and expressions. Obviously, the main difference between CML and OCML is that the former is only meant to be an informal notation while the latter is a fully operational language. Another important difference is that CML is committed to supporting the Common KADS framework, while the kernel of the OCML language is framework-independent. In addition to the KADS-related commitments CML also embodies other modelling commitments: it provides primitives for representing structures and part-of relations. This approach has both advantages and disadvantages. On the plus side it extends the range of modelling primitives provided by the language and supports notions which occur frequently in conceptual modelling. On the other hand there are two possible problems with this approach. The first one has to do with embedding ontological commitments in a conceptual modelling language. In particular different approaches to modelling structure and aggregation can be found in the literature - e.g. compare the analysis by Martin and Odell (1995) with that by Lenat and Guha (1990). Embedding one particular approach in the kernel of a conceptual modelling language prevents users from extending the language according to an alternative approach. The second problem is caused by the provision of different levels of description - i.e. logical, epistemological and conceptual (Brachman, 1979) - within the same formalism. Much work in knowledge representation over the past twenty years has focused on identifying the different levels at which knowledge representation languages can be specified (Brachman, 1979; Guarino 1994). In particular the paper by Brachman clearly illustrates that much of the confusion surrounding the field of knowledge representation in the seventies was caused by the fact that researchers were comparing formalisms which were situated at different levels. Thus, it seems to me that including ontological primitives (e.g. structures) in a language characterized at the logical level is a potential source of confusion. This is especially the case with informal languages, given that eventual ambiguities are not explained by the underlying formal theory.

The LOOM language (MacGregor, 1991) is strictly speaking a knowledge representation rather than knowledge modelling language (i.e. it also includes symbol-level representation constructs). However, its formal kernel - i.e. the terminological and assertional components - provides purely logical and epistemological primitives and therefore it can be used for knowledge modelling and ontology specification. The main feature of LOOM is its powerful classification mechanism which integrates a sophisticated concept definition language with rule-based reasoning. This approach allows a wider range of inferences to be drawn than those available from 'traditional' frames+rules

systems such as KEE. The existence of a powerful classifier is an important advantage that LOOM maintains over OCML. On the other hand, viewed purely as a knowledge modelling language, LOOM exhibits a number of limitations.[16] When building knowledge models it may be necessary to make statements about the model being developed, which do not have a direct inferential purpose. For this reason OCML allows the inclusion of non-operational statements and supports the specification of axioms about the current model. In contrast, LOOM only provides operational constructs which might restrict the usability of this language for knowledge modelling. Another possible problem related to LOOM is that while it supports different knowledge representation paradigms (frames, rules, message-passing) it integrates the various constructs according to a classification-centred viewpoint. While this approach obtains nice results in terms of inferential capabilities, it nevertheless can be a constraint for the knowledge analyst, who is forced to frame the current problem within a classification-centred framework. In contrast with this approach OCML provides a number of alternative modelling constructs, e.g. rules, functions, classes and procedures which, while integrated, are themselves 'primitive modelling components' and can be used within different modelling approaches.

4.9. CONCLUSIONS

The OCML language is meant to provide a useful tool for knowledge modelling. Its primary purpose is to provide operational knowledge modelling facilities and to this end it includes interpreters for functional and control terms, as well as a proof system which integrates inheritance with backward chaining, function evaluation and procedural attachments. While the emphasis here is on operationality, of course I recognize that different styles of knowledge modelling ought to be supported. Therefore OCML supports full first-order logic definitions and allows the user to explicitly distinguish non-operational from operational definitions. Moreover, it supports an extensive set of Ontolingua constructs and therefore can be used as an interpreter for Ontolingua definitions.

In the rest of this work I will illustrate several models and library components which were developed and tested using OCML. As they say, the proof is in the pudding!

---

[16] The following points should not be construed as criticisms of the LOOM language. This is primarily a knowledge representation language and should be of course judged with respect to knowledge representation criteria. However, given the high-level of support provided by the language, its 'organic relationship' with the Ontolingua effort and its sound theoretical basis, it makes sense to consider it as a plausible candidate for knowledge modelling. Indeed LOOM has recently been used for ontological work (Swartout et al., 1996).

Chapter 5.
An Ontology for Task-Method Structures

In this chapter the task-method framework used for analyzing problem solving methods and for organizing the library of parametric design components is formalized by means of an OCML ontology.

5.1. BASIC TASK TYPES

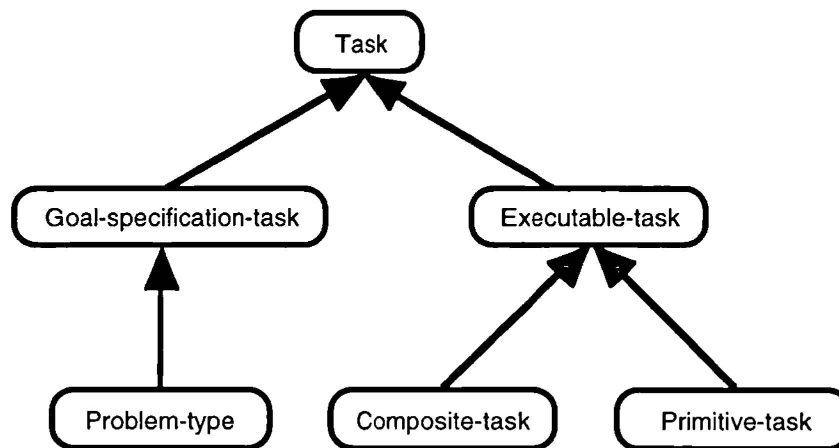


Figure 5.1. Main types of tasks in task-method ontology.

The `subclass-of` hierarchy shown in figure 5.1 shows the main types of tasks which are defined in the task-method ontology. These classes have already been illustrated in section 3.2 and therefore there is no need to (re-)introduce them here. Hence, I will skip the 'introduction formalities' and will move directly on to the representation issues which arise when formalizing these concepts.

5.1.1. Modelling tasks in OCML

As already discussed in section 3.2.1, a generic task can be characterized in terms of its *input* and *output roles* and a *goal expression*. The definition of class `task` given below formalizes this approach and provides the entry point for the task-method ontology.

```
(def-class TASK () ?task
  "An OCML task is characterized by its
   input roles, output role, and goal. The goal expression is a
   kappa expression which takes as argument the task itself and a
   value (which is meant to be a possible result from carrying out the
   task).  The goal is satisfied if the kappa expression holds for its
   two arguments.
   A role is a slot of a task, which admits only one value.
   Tasks divide into two main subclasses:
   goal-specification-task and executable-task.   The former
   provides only a goal specification, while the latter provides
   also an 'organic' method for achieving the task"
  ((has-input-role :type role)
   (has-output-role :type role)
   (has-goal-expression :type legal-task-goal-expression
                        :max-cardinality 1))
  :constraint (=> (has-role ?task ?role)
                  (and (slot-of ?role ?task)
                       (functional-relation ?role)))
  :axiom-def (exhaustive-subclass-partition
              task
              (set-of goal-specification-task
                      executable-task)))
```

The association between a task and a goal is modelled in the above definition by means of the slot `has-goal-expression`, whose slot specification uses the option `:max-cardinality` to specify that a task can at most have one goal, but does not necessarily need to have one. This ‘flexibility’ in the specification ensures that all types of tasks are subsumed by this definition. In particular, while most tasks express goal-oriented specifications, some executable tasks only define generic control constructs, not directly associated with a goal[1]. Thus, in order to account for both types of tasks the ontology does not enforce the constraint that each task must be given a goal.

In accordance with figure 5.1 and the discussion in section 3.2.1, the above definition partitions the class of generic tasks into two subclasses - `goal-specification-task` and `executable-task`. Note that these two subclasses are mutually disjoint. The former includes all task instances which have a goal and no body; the latter all task instances which have a body and (optionally) a goal.

Finally, the definition also formalizes roles as slots of a task. In addition, the slots `has-input-role` and `has-output-role` provide a way to distinguish explicitly the slots of a

[1] For instance, a generic Generate&Test control construct can be used in various problem solving models to perform search. However, the specification of such a construct does not need a goal. The construct can be used as a plug-in component for a larger task or method, for which a goal has been defined.

task which define roles from other, 'ordinary' ones. Roles are discussed in detail in section 5.4.

5.1.2. Executable vs. goal-specification tasks

The class of goal specification tasks can be simply defined as a subclass of class `task`, with the additional constraint that a goal but no body should be specified. The former constraint can be imposed by setting the cardinality of slot `has-goal-expression` to 1.

```
(def-class GOAL-SPECIFICATION-TASK (task) ?task
  "A goal-specification-task is a task with a goal
   expression and no body"
  ((has-goal-expression :cardinality 1))
  :constraint (not (exists ?body
                           (has-body ?task ?body))))
```

The definition of `executable-task` given below distinguishes this class by the existence of a task body and partitions it into two subclasses: `primitive-task` and `composite-task`.

```
(def-class EXECUTABLE-TASK (task)
  "An executable task is a task with a body - i.e. a
   task whose specification also includes a mechanism for
   achieving it"
  ((has-body :type unary-procedure :cardinality 1))
  :axiom-def (exhaustive-subclass-partition
              executable-task
              (set-of primitive-task
                      composite-task)))

(def-class PRIMITIVE-TASK (executable-task) ?task
  "An executable task which is not a composite task"
  :iff-def (not (exists (?c ?subs)
                        (and (instance-of ?task ?c)
                             (has-generic-subtasks ?c ?subs)))))

(def-class COMPOSITE-TASK (executable-task) ?task
  "A composite task is a task which introduces a subtask
   decomposition.  Something is an instance of this task if
   its parent introduces a generic task-subtask
   decomposition"
  :iff-def (and (instance-of ?task ?c)
                (has-generic-subtasks ?c ?subs)))
```

The distinction between primitive and composite tasks is enforced by pointing out that the latter specify *generic subtasks*, while the former do not. A *generic task-subtask hierarchy* is defined as one in which the nodes are generic tasks, such as parametric design, rather than concrete ones, such as the VT task.

A problem with representing a notion such as 'generic task' in an ontology is that this concept is typically overloaded with cognitive and modelling connotations, not all of

which can be captured in the representation. In particular, the ontology described here does not attempt to formalize one of the main aspects of generic tasks, which is their domain-independence. This aspect will instead be enforced methodologically, by constructing models where generic tasks are characterized in a domain-independent style. Once stripped of its methodological connotations, then the notion of 'generic task' equates to 'class of tasks' and therefore I simply represent generic tasks as classes of tasks. An important consequence of this decision is that much of the ontology building process consists then of defining second order relations between classes of tasks. An example of such a second order relation is `has-generic-subtasks`, which relates a generic task to its generic subtasks.

```
(def-relation HAS-GENERIC-SUBTASKS (?task-type ?subs)
  "Use this to model generic task-subtask decompositions"
  :constraint (and (subclass-of ?task-type composite-task)
                   (every ?subs task-type)))
```

An advantage of this approach - i.e. equating generic tasks to task classes - is that it allows a uniform treatment of the two distinctions between generic and concrete problems (in the case of goal specification tasks) and between task specification and execution (in the case of executable tasks).

Thus, in the rest I will distinguish between *generic* and *concrete* task-subtask hierarchies. The former relate generic tasks and define the *potential* structure of a problem solver. The latter specify the *actual* task-subtask structure constructed during problem solving.

5.2. MODELLING PROBLEM SOLVING METHODS

5.2.1. Representing methods in OCML

As shown below, a method is characterized as a special type of executable task, which is associated - by means of a slot called `tackles-task` - to a goal specification task. Thus, I characterize methods and method execution as particular cases of tasks and task execution. The feature distinguishing methods from other executable tasks is that they cannot be carried out independently of an associated task. In other words, the raison d'être of a method is given by the task to which it is applied.

```
(def-class PROBLEM-SOLVING-METHOD (executable-task)
  "A problem solving method is an executable task which
   is associated with (tackles) a class of tasks.
   The slot has-output-mapping specifies a function which maps
   the result returned by the method to a task.
   The reason for this mapping is to allow the decoupling
   of the type of result returned by the method from that expected
   by the task.  This provides greater flexibility and also makes it
   possible to specify solution conditions for a method which use
   different types of output from that used by a task"
  ((tackles-task :type goal-specification-task)
   (has-output-mapping
    :value '(lambda (?psm ?result)
             ?result))))
```

This characterization of methods as a special case of tasks is also informed by the view of AI as an experimental science, already expressed in section 2.4.2.1. Because a method is formalized as a task, it is carried out to achieve some goal. Hence, it follows that a method can also fail - i.e. goals are not always achieved. This goal-centred formalization is different from other approaches which describe methods in terms of their competence - e.g. (Wielinga et al., 1995; Angele et al., 1996). The problem here is that the notion of competence implies an axiomatic specification of the functionality of a method. However, such a specification might only have relative utility for heuristic search methods. For instance, a Propose&Revise can be functionally described by stating that its output consists of a complete and correct state (Fensel et al., 1997). However, such a specification does not imply that in practice a Propose&Revise can achieve such complete and correct state, or that such a state exists in the given problem space. Because Propose&Revise problem solvers do not use domain-independent converging criteria, their competence depends on the quality of the available heuristic knowledge (Zdrahal and Motta, 1996; Chapter 8). More in general, as shown by recent work on empirical AI, the effective competence of heuristic search methods has often to be assessed empirically (Cheeseman et al., 1991)[2]. For this reason I prefer to characterize methods as goal-centred, rather than competence-centred, problem solving components.

Method application is defined by means of the procedure `apply-method-to-task`, shown below.

---

[2] Recent work on formal specifications of problem solving methods is aware of this problem and is trying to overcome it by parametrizing a functional specification in terms of a number of *assumptions* (Fensel and Schoenegge, 1997b; Fensel et al., 1997). Although this work is still at a fairly early stage, it appears to me that it provides the most promising approach to reconciling the advantages of formal specifications with the heuristic nature of AI.

```
(def-procedure APPLY-METHOD-TO-TASK (?method-inst ?task-inst)
  :body (do
          (tell (tackles-task ?method-inst ?task-inst))
          (in-environment
           ((?output-role . (the-slot-value
                             ?task-inst
                             has-output-role))
            (?fun . (the ?fun (has-output-mapping ?method-inst ?fun)))
            (?result . (perform-executable-task ?method-inst)))
           (set-slot-value ?task-inst
                           ?output-role
                           (call ?fun ?method-inst ?result))
          ?result)))
```

The procedure `apply-method-to-task` registers the application of a method to a task by linking them through a `tackles-task` relation, and then sets the output role of the task to the result of executing the method. The procedure uses the value of slot `has-output-mapping` to convert the result obtained by the method to a format which satisfies the associated task. The reason for this mapping is to allow the decoupling of the type of result returned by the method from that expected by the task. For instance, the definition below uses this feature to convert the value returned by a method (a design state) into a format appropriate to the associated task (a design model).

```
(def-class GEN-DESIGN-PSM
  (problem-solving-method-for-parametric-design
   decomposition-method)
  ((has-input-role :value has-design-operators)
   (has-output-role :value has-solution-state)
   (has-solution-state :type design-state)
   (has-design-operators :type design-operator)
   (has-output-mapping
    :value '(lambda (?psm ?state)
              (the ?dm (has-design-model ?state ?dm))))
   (has-body :value
             '(lambda (?psm)
                (in-environment
                 ((?s . (achieve-generic-subtask
                         ?psm gen-design-control
                         has-current-pardes-task
                         (the ?task (tackles-task ?psm ?task)))))
                 (if (design-state ?s)
                   ?s)))))
  :own-slots ((has-generic-subtasks '(gen-design-control))))
```

5.2.2. Modelling support for library organization

Consistently with the task-centred approach taken in the organization of the library, the task-method ontology provides two relations for associating classes of methods to classes of tasks, `tackles-task-type` and `applicability-condition`. The former makes it possible to enforce the task-specificity of the problem solving methods included in the library by providing a way to associate a method to the class of tasks to which it can be

applied. The latter can be used to specify more fine-grained applicability conditions than is possible by means of simple links between method and task classes. Thus, when looking for a method to solve a particular task, a meta-reasoner can check the applicability conditions of the potentially applicable methods and filter out those methods which are not applicable to the task in question.

It is important to note that, in contrast with the relation `tackles-task`, which we saw in the previous section, the relations below are defined on method classes, rather than method instances. The reason is that the purpose of these relations is to structure the library, which consists of generic components (i.e. classes).

```
(def-relation TACKLES-TASK-TYPE (?method-class ?task-type)
  "This relation provides a fairly coarse-grained indexing of the
   library: each method is associated to a class of tasks to which
   it can be applied"
  :constraint (and (subclass-of ?method-class
                                problem-solving-method)
                   (task-type ?task-type)))

(def-relation APPLICABILITY-CONDITION (?method-class ?exp)
  "This relation provides a more fine-grained test to check
   the applicability of a class of methods to a specific task"
  :constraint (and (subclass-of ?method-class
                                problem-solving-method)
                   (unary-relation ?exp)))
```

The relations given here are only meant to provide generic structuring mechanisms for organizing the library of reusable problem solving components. In addition to these the task-method ontology also provides mechanisms for specifying application-specific method selection knowledge. These are described in section 5.5.

5.2.3. Types of methods

As in the case of executable tasks, methods are divided into two subclasses, *primitive* and *decomposition methods*. The former solve a goal specification task directly, the latter introduce a task-subtask decomposition. Thus, the generic structure of the task-method hierarchies which can be constructed by means of the proposed ontology includes both task-mediated and method-mediated decompositions, as shown in figure 5.2.

Problem type

Decomposition-Method-1 .... Decomposition-Method-n

Primitive-task-1-1 .... Composite-task-1-m

Primitive-task-n-1 .... Goal-specification-task-n-i

Primitive-task-1-m-1 .... Goal-specification-task-1-m-k

Primitive-Method-1-m-k-1

Decomposition-Method-1-m-k-1

Primitive-task-1-m-k-1 .... Primitive-task-1-m-k-j

Figure 5.2. Generic structure of task-method hierarchies.

5.3. MODELLING GOAL EXPRESSIONS

A legal goal expression - see definition below - is a *kappa expression*, which takes as argument a task instance, say `?task`, and a value, say `?value`. If a pair `<?task, ?value>` satisfies the goal expression associated with `?task`, then we say that `?value` is a solution to `?task`. This informal notion of 'achieving a task' is formalised by means of the relation `achieved`, which is shown in the following box.

```
(def-class LEGAL-TASK-GOAL-EXPRESSION (kappa-expression) ?exp
  "A task goal expression is a kappa expression with arity 2,
   which does not contain free variables.  The first argument to
   the kappa expression represents a task-instance, the second
   the result of the task"
  :iff-def (and (= (arity ?exp) 2)
                (= ?exp (kappa ?schema ?sent))
                (= ?vars (all-free-vars-in-sentence ?sent))
                (= (length ?vars) 2)
                (member (namestring (first ?vars))
                        (map namestring ?schema))
                (member (namestring (second ?vars))
                        (map namestring ?schema))))
```

```
(def-relation ACHIEVED (?task ?result)
  "A task has been achieved if its goal holds in the current model
   or if no goal has been specified.
   A method has been achieved either if its goal has been achieved
   or if its associated task has.
   If the task has no goal expression, then it has been trivially
   achieved "
  :iff-def (or (and (has-goal-expression ?task ?exp)
                    (holds ?exp ?task ?result))
               (and (problem-solving-method ?task)
                    (tackles-task ?task ?task2)
                    (achieved ?task2 ?result))
               (and (not (problem-solving-method ?task))
                    (not (has-goal-expression ?task ?exp)))))
```

The definition of `achieved` distinguishes between three cases. The first is the one in which a goal expression has been specified for the task in question. In this case the task, `?task`, is achieved with respect to a particular value, `?value`, if the goal expression is satisfied by the pair `<?task ?value>`. The second case is that of problem solving methods. The execution of a method which either has no goal, or which has a goal which has not been achieved, is declared successful if the goal of the associated task has been satisfied. The third case is the one in which no goal expression has been declared for the argument task. In this case the task is trivially achieved.

While the given definition specifies quite obvious conditions for achieving 'ordinary' tasks, the case of problem solving methods is more interesting. In particular the definition accounts for the frequent situation in which no goal is associated with a method and therefore the method 'inherits' the goal of the task to which it has been applied. In this case method application is successful if and only if the goal of the associated task has been achieved.

5.4. ROLES AND ROLE VALUES

5.4.1. Roles as meta-level concepts

Generic reasoning roles, called *metaclasses*, were first identified by Clancey in his landmark paper on heuristic classification (Clancey, 1985). The essential feature of roles, which distinguishes them from argument schemas in software engineering, is that they are characterized at the metalevel, in a domain-independent style. In particular the KADS approach characterizes roles as 'labels' or 'pointers' to domain-specific knowledge structures, indicating the role (!) these domain structures play in the reasoning process (Wielinga et al., 1992a).

Here, I will do away with the notion of roles as labels and characterize them instead as slots of a task or a method (i.e. as binary relations). This approach makes it possible to

provide 'semantically rich' descriptions of roles, using the relation specification machinery afforded by OCML.

Like other approaches based on the task-method framework (Benjamins, 1993; Steels, 1990) I distinguish between *input*, *output* and *control* roles. The latter specify intermediate knowledge structures which are generated and modified during problem solving. Control roles can only be associated with composite tasks and decomposition methods. More precisely, a role, say `?role`, is a control role of a decomposition task, `?task`, if i) it is the input or the output of a subtask of `?task` and ii) it does not denote a class of knowledge structures which must be acquired in order to achieve the subtask. In other words control roles only define intermediate reasoning structures and are not meant to introduce *ontological commitments*[3].

5.4.2. Modelling roles in OCML

As already pointed out and as shown in the definition of task `parametric-design` - see below - input and output roles in the OCML task-method ontology are represented as task slots. Thus, an OCML role, e.g. an input role, strictly speaking does not denote the class of knowledge structures which are input to a task, but rather a relation between these and a task. For instance, while I might informally talk about parameters being the input to a parametric design task, the solution used in the OCML task-method ontology implies that the relation `has-parameters` is the actual input role of the task. Nevertheless, in this and the next chapters, I will in some cases speak informally of (for instance) parameters as roles of a parametric design task with the understanding that this is only 'informal talk'.

---

[3] An ontological commitment is a specification of a requirement describing either i) a particular class of knowledge structures which has to be acquired in order to perform a task, or ii) additional properties which must be satisfied by already available knowledge.

```
(def-task parametric-design (design-task) ?task
  ((has-input-role :value has-parameters
                   :value has-constraints
                   :value has-requirements
                   :value has-cost-function
                   :value has-cost-algebra
                   :value has-preferences)
   (has-output-role :value has-design-model :cardinality 1)
   (has-design-model :type design-model :max-cardinality 1)
   (has-parameters :type list :cardinality 1)
   (has-constraints :type list :max-cardinality 1)
   (has-requirements :type list :max-cardinality 1)
   (has-preferences :type list :max-cardinality 1)
   (has-cost-function :type cost-function :max-cardinality 1)
   (has-cost-algebra :default-value '(+ - <) :cardinality 1)

   (has-goal-expression
    :type legal-parametric-design-goal
    :default-value  (kappa (?task ?design-model)
                           (design-model-solution
                            ?design-model
                            ?task))))
  :lisp-class-name parametric-design)
```

The definition of class `role` given below defines a role as an entity of the universe of discourse, say `?role`, for which exists a class of tasks, say `?c`, such that the sentence `(has-role ?c ?role)` is satisfied. This definition also enforces the constraints that roles are task slots and that they specify *functional relations*, i.e. that the slot defining a role can only have one, rather than multiple values. This constraint is imposed both in order to simplify the retrieval and modification of role values, and also to ensure that roles can be treated as if they were variables in a programming language - i.e. so that we can talk about retrieving and modifying the *value* of a role.

```
(def-class ROLE  (slot) ?role
  "A role is a binary relation associated with a task by
   means of the 'has-role' relation.  The value cardinality
   of a role-defining slot is 1."
  :constraint (and (slot-of ?role ?task)
                   (forall (?i)
                    (=> (and (has-role ?class ?role)
                             (instance-of ?i ?class))
                        (has-one ?i ?role))))
  :iff-def (exists ?c
                   (and (task-type ?c)
                        (has-role ?c ?role))))
```

The relation `has-role`, whose definition is given below, associates a class of tasks or a task instance to all its roles, thus generalizing from input, output, and control roles.

```
(def-relation HAS-ROLE (?thing ?role)
  "Generalises from input output and control roles.
   This definition applies to both task instances and task types"
  :iff-def (or (and (task ?thing)
                    (has-role (the-parent ?thing) ?role))
               (and (task-type ?thing)
                    (member ?role
                            (union (setofall ?r (has-input-role
                                                 ?thing ?r))
                                   (setofall ?r (has-control-role
                                                 ?thing ?r))
                                   (setofall ?r (has-output-role
                                                 ?thing ?r)))))))
```

5.4.3. Roles as variables: issues of scope

While the notion of task role is intuitive enough at an informal level of description, it is less clear how to operationalize this notion when specifying a model of task execution. In particular, it is often the case that a subtask shares the input roles of its supertask. For instance most subtasks of a parametric design problem solver access the value of basic problem inputs such as parameters and constraints. One way to ensure that this is possible is of course to specify explicitly parameters and constraints as input roles for all relevant subtasks. However, this solution leads to much redundancy in the specification and therefore I have opted for making the role of a task visible to each of its subtasks. This solution corresponds to declaring that the scope of a role includes all subtasks of the task in which the role has been specified. The ensuing benefit is that each subtask only needs to specify those knowledge roles which have not been already specified by some of its supertasks, thus leading to more concise specifications.

This approach is implemented by means of the function `role-value`, which is defined below. The body of this function attempts first to retrieve the value of a task role locally; if this is not found (i.e. if the role is not local to the task), then it looks for it through a search up the task-subtask hierarchy.[4]

---

[4] It is important to keep in mind that such task-subtask hierarchy is the concrete hierarchy developed during problem solving, which is modelled by means of the `subtask-of` relation, rather than the generic one defined in terms of task-subtask relations between generic tasks. This is modelled by means of the relation `has-generic-subtasks`.

```
(def-function ROLE-VALUE (?task ?role)
  "The value of a role is its local value if it exists.
   If it does not then the subtask-of hierarchy is searched
   for a value."
  :body (in-environment ((?value . (local-role-value ?task ?role)))
          (if (and (= ?value :nothing)
                   (subtask-of ?task ?supertask))
            (role-value ?supertask ?role)
            ?value)))
```

5.5. CARRYING OUT TASKS

If a task is executable, it can be carried out by evaluating its body. Otherwise a relevant class of methods is selected, its applicability condition tested, and - in the case of a positive result - an instance of the method is created and applied to the task - see definition below.

```
(def-procedure SOLVE-TASK (?task-instance)
  "A task, ?task, is executed by evaluating its body
   in an environment in which the schema of the class
   corresponding to the parent of ?task is bound to
   ?task."
  :body (if (executable-task ?task-instance has-body ?body)
          (execute-task-body ?body ?task-instance)
          (if (= ?best-psm
                 (choose-best-method-class
                  (setofall ?psm-type
                            (and
                             (subclass-of ?psm-type
                                          problem-solving-method)
                             (tackles-task-type ?psm-type  ?c)
                             (instance-of ?task-instance ?c)
                             (applicable-to-task ?psm-type
                                                 ?task-instance)))))
            (in-environment
             ((?method . (instantiate-generic-subtask
                          ?task-instance ?best-psm)))
             (apply-method-to-task ?method ?task-instance)))))
```

The most interesting aspect of the above definition is given by the case in which multiple methods are applicable to the current task. In this case the function `choose-best-method-class`, shown below, is used to select the 'best' from all the applicable methods.

```
(def-function CHOOSE-BEST-METHOD-CLASS (?psm-types)
  :body (if (null ?psm-types)
          :nothing
          (if (= (length ?psm-types) 1)
            (first ?psm-types)
            (if (exists ?x
                        (and
                         (member ?x ?psm-types)
                         (use-method ?x ?c ?m)))
              (choose-from-use-method-statements ?psm-types)
              (first ?psm-types)))))
```

Method selection conflicts are resolved by means of application- or method-specific knowledge, which is expressed using statements of the form `(use-method ?method ?task ?context)`. These statements indicate that `?method` should be used when tackling `?task` in context `?context`. A context is a task (or a method) which is above `?task` in the current task-subtask hierarchy. For instance, the statement `(use-method hc-control design-from-state hc-design)` indicates that the method `hc-control` should be used to tackle task `design-from-state` in a situation in which we are using a hill climbing approach to design (i.e. method `hc-design`). Relation `use-method` is defined as follows.

```
(def-relation USE-METHOD (?sub-method ?sub-task ?thing)
  "Use instances of this relation to specify which sub-method
   to use when solving a generic subtask of a problem. The third
   argument can be used to contextualise this statement within a
   problem solving method or a particular problem.
   EXAMPLE: (use-method HC-CONTROL DESIGN-FROM-STATE HC-DESIGN)"
  :constraint (and (subclass-of ?sub-method problems-solving-method)
                   (subclass-of ?sub-task task)
                   (or (task ?thing)
                       (subclass-of ?thing problem-solving-method))))
```

When multiple `use-method` statements are applicable, the function `choose-from-use-method-statements` selects the one which is defined in the most specific context - see definition below[5].

---

[5] Strictly speaking functions `choose-best-method-class` and `choose-from-use-method-statements` are not part of the ontology. These functions define a particular way of solving method selection conflicts, rather than contributing to specify the task-method conceptualization. On the other hand, it is convenient to include them here, to provide a complete overview of both the concepts and the actual functionalities provided by the formalization of the task-method framework.

```
(def-function CHOOSE-FROM-USE-METHOD-STATEMENTS (?psm-types)
  :body (if (and (use-method ?x ?c (the-current-task))
                 (member ?x ?psm-types))
            ?x
            (in-environment
             ;;try to pick the most specific use-method
             ;;statement for this subtask
             ((?psm-type . (the ?x
                              (and
                               (member ?x ?psm-types)
                               (use-method ?x ?c ?m)
                               (instance-of (the-current-method) ?m)
                               (not
                                (exists
                                 (?m2 ?x2)
                                 (and (member ?x2 ?psm-types)
                                      (use-method ?x2 ?c ?m2)
                                      (subclass-of ?m2 ?m)
                                      (instance-of
                                       (the-current-method)
                                       ?m2))))))))
             (if (= ?psm-type :nothing)
               (first ?psm-types)
               ?psm-type))))
```

The approach to dynamic method selection described here has the advantage of combining application-specific and method-specific knowledge and also of providing a simple, but effective, context-based mechanism for resolving conflicts. Moreover, it can be easily generalized to take into account additional method selection knowledge, such as *environmental features* (Benjamins, 1993).

It is also important to note the use of procedure `instantiate-generic-subtask` in the definition of `solve-task`, to create an instance of a method class. This is needed because the methods stored in the library are defined as classes (i.e. they are generic), while only instances can be executed[6].

Finally, the relation `applicable-to-task` is used in the definition of `solve-task` to check whether an applicability condition is specified for the method and, if this is the case, whether it is verified by the task which we are trying to solve.

```
(def-relation APPLICABLE-TO-TASK (?method-class ?task-inst)
  :iff-def (or (not (applicability-condition ?method-class  ?exp))
               (holds (the ?exp (applicability-condition
                                 ?method-class  ?exp))
                      ?task-inst)))
```

6 As already said, it is useful to model generic methods as classes as this approach makes it possible to distinguish between the specification of a method and its (possibly multiple) activations.

5.6. APPLICATION MODELLING

Having defined the concepts discussed in the previous sections, the task of modelling an application does not present any major difficulty. The framework presented in chapter 3 characterises an application in terms of a domain, a task, a method and application specific knowledge. However, the latter is not explicitly modelled in the current version of the ontology, but it is instead integrated directly in an application model - see chapter 9 for examples of these. Thus, the current version of the task-method ontology simply models an application in terms of task, method and domain, as shown below.

```
(def-class APPLICATION ()
  ((tackles-domain :type application-domain :cardinality 1)
   (uses-method :type problem-solving-method :cardinality 1)
   (tackles-task :type problem-type :cardinality 1)))
```

For instance, the application constructed to solve VT by means of a Propose&Revise method can be defined as follows.

```
(def-instance  vt-as-p&r  design-application
  ((tackles-domain vt-domain)
   (uses-method vt-propose&revise)
   (tackles-task vt-pardes-task)))
```

Finally, application-specific problem solving can be defined simply as the process of applying a method to an application task. This process is defined by means of the procedure `solve-application`, which is shown below.

```
(def-procedure SOLVE-APPLICATION (?appl)
  :body (if (application ?appl
                         uses-method ?method-inst
                         tackles-task ?task-inst)
          (do
            (unassert (current-application ?any))
            (tell (current-application ?appl))
            (apply-method-to-task ?method-inst ?task-inst))))
```

5.7. CONCLUSIONS

In this chapter I have illustrated the main aspects of the task-method ontology, which is part of the basic set of OCML ontologies. A full description of the ontology is available in appendix 2.

The main purpose of the ontology presented here is to give a formal basis to the informal task-method framework illustrated in the previous chapter. In particular, the ontology provides formal definitions of the concepts required to model libraries of reusable tasks as well as a model of task and method execution. In the next chapters I will use the concepts

presented here to model reusable problem solving components for parametric design applications.

Chapter 6.
An Ontology for Parametric Design Tasks

This chapter situates parametric design applications in the context of the wider class of AI design problems, proposes a task ontology for parametric design and compares this to alternative proposals.

6.1. THE NATURE OF PARAMETRIC DESIGN APPLICATIONS

6.1.1. Creative Design

Design can be characterized in generic terms as the process of constructing artefacts. Thus, the essential feature of design problem solving is its *constructive* nature: solutions are constructed rather than retrieved from a pre-existing set. Non-design problem solving is often characterized as *analysis* (Clancey, 1985).

In order to construct an artefact one needs some *building blocks* - i.e. a *technology* (Chandrasekaran, 1990). These building blocks can take many different forms, depending not only on the domain but also on the granularity of the design process. For instance, while an architect designing a skyscraper might consider an elevator as a single component, at some later stage the elevator itself will become the target of a more fine-grained design process. Moreover, building blocks do not need to be physical components. For instance, a scheduling problem can be characterized as a design problem where the building blocks are given by time-dependent activities (Friedland and Iwasaki, 1985).

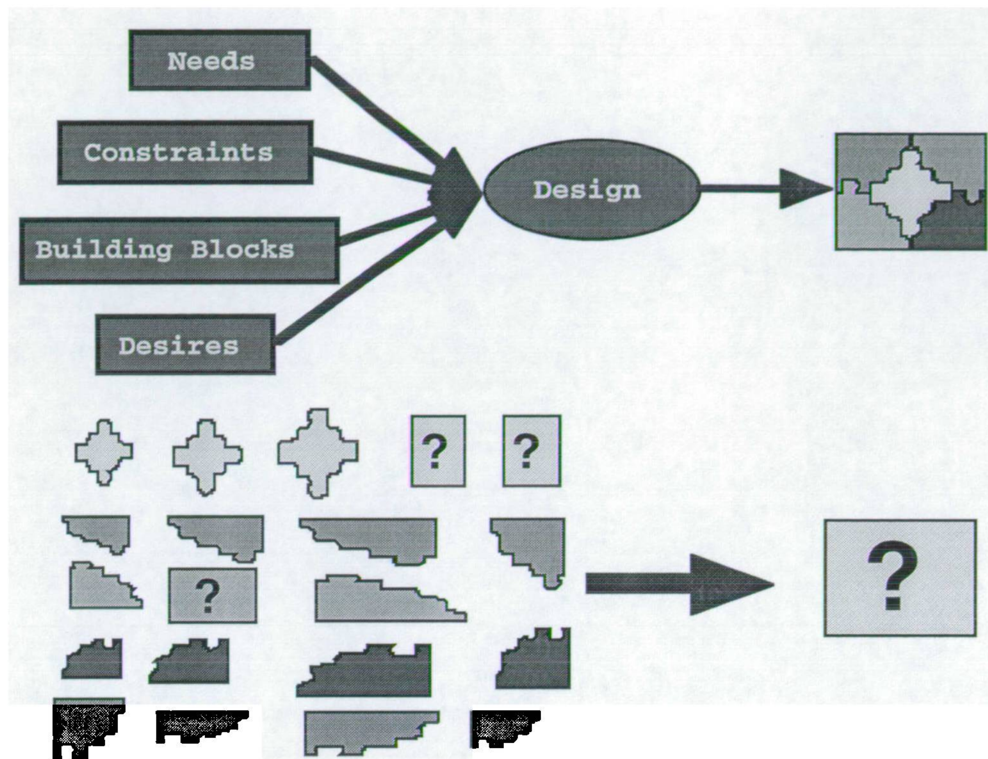


Figure 6.1. A generic characterization of design problem solving

Design is a goal-driven process, where the goals are specified in terms of a number of functionalities which the target artefact should provide. Not all functionalities have necessarily the same degree of importance. For instance, while it is crucial for an estate car to have a large boot, this is only a 'desirable' feature when designing a sports car. A sports car designer is normally prepared to reduce the size of the boot if handling and aesthetic considerations require him/her to do so. A good way of informally characterizing the goal-oriented nature of the design process is to see it as driven by *needs* and *desires* (Wielinga et al., 1995).

Finally, a design process is normally subject to a number of *constraints*, which can be related either to the design technology - e.g. technological limitations impose constraints on the minimum size of supporting walls - or with external factors - e.g. most civilized countries require a minimum ceiling height in living rooms[1].

In sum, the design process can be characterized as a function which takes needs, desires, constraints and building blocks as input, and produces an artefact as output. As shown in figure 6.1, in general not all 'building blocks' are necessarily known at the start of the design process. Often sub-design processes are set up to produce the components required by the original design goals. For instance, car designers have been able to

---

1 It is unclear whether the United Kingdom is civilized in this respect!

increase the size of the boot in modern cars by devising new types of suspensions which sit horizontally rather than vertically, thus taking very little space off the load area. Moreover, in the general case the structure of the target artefact is not known in advance - i.e. there is no solution template. This type of design activity is often called *creative design* (Gero, 1990).

6.1.2. Configuration design

The 'open-ended' model of the design problem shown in figure 6.1 can be circumscribed by considering a simpler scenario in which all the building blocks are known at the beginning of the design process. For instance, this is the case when building a house with a set of Lego™ components, or when furnishing an apartment. In the latter case the designer usually knows the types of building blocks in advance - e.g. chairs, tables, beds, wardrobes, etc. - although he or she won't know beforehand how these *design elements* are going to be *arranged* in order to provide the desired functionalities. Design problems where the set of input components is fixed are called *configuration design* problems (Stefik, 1995). As shown in figure 6.2, the main difference between design in general and configuration design is that in the latter case there is no 'uncertainty' in the available design technology: the set of building blocks is known at the beginning of the process. However, as in the case of a generic design scenario, the structure of the solution is not necessarily known at the start of a configuration design process.

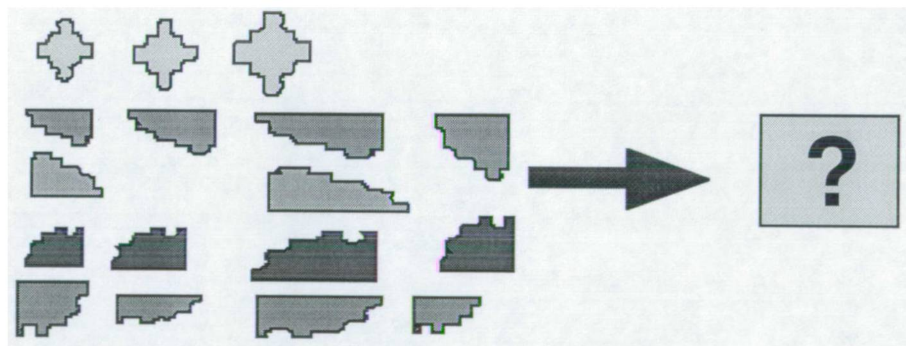


Figure 6.2. Configuration design problem solving

6.1.3. Parametric design

The general case of configuration design problem solving assumes a free arrangement of components, which in many situations is too generic an assumption: designers typically have some rough idea of what a solution is going to look like. For instance, when configuring a computer a designer expects the final artefact to comprise a processor, printing connections, memory, data communication, etc. In this case we say that the designer in question makes use of a *functional solution template* to guide the design process. As discussed by Mittal and Frayman (1989), such assumptions concerning the

existence of "known functional architectures" greatly decrease the complexity of the design process.

A stronger assumption, which further restricts the space of possible designs is that which postulates the existence of a *parametrized solution template* for the target artefact - see figure 6.3. In this scenario design problem solving can be described as the process of assigning values to *design parameters* in accordance with the given needs, constraints, and desires. Applications for which this assumption holds are called *parametric design* applications.

The VT elevator design problem (Marcus et al., 1988; Yost and Rothenfluh, 1996) provides a well-known example of a parametric design task. Here the problem is to configure an elevator in accordance with the given requirement specification and the applicable constraints. The parametrized solution template consists of 199 design parameters which specify the various structural and functional aspects of an elevator - e.g. number of doors, speed, load, etc.

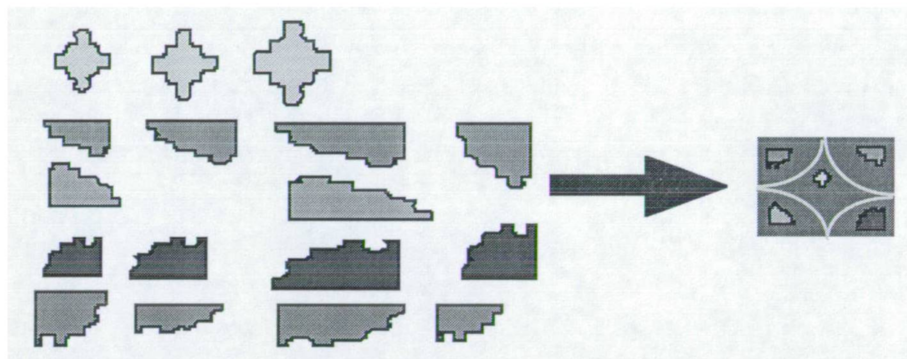


Figure 6.3. Parametric design problem solving

Obviously, it is important to keep in mind that these definitions of different classes of design tasks are best seen as modelling templates, rather than essential features of an application. For instance, while a house furnishing problem is strictly speaking a configuration design task, it can often be modelled as a parametric design problem by introducing skeletal solution templates which provide furnishing schema for individual rooms. On the other hand not all design problems can be reduced to parametric design. In these cases, "designing involves... exploring which variables might be appropriate..." (Gero, 1990). That is, a parametric design specification is often the output, rather than the input of a generic design process.

In a nutshell, there are four main reasons why the study of parametric design applications is an important area of KBS research.

- All design problems - even ill-structured ones - comprise a number of parametric design sub-problems.

- The design process itself can be seen as a way of transforming complex problems into problems with a well-understood structure - i.e. the study of parametric design tasks helps to clarify the structure of the output of the meta-design process.
- In practice many real-world applications can be directly modelled as parametric design problems.
- Because of their relative simplicity, parametric design problems are more amenable to automation than other, more complex classes of design problems.

In the next section I will discuss a conceptual framework for characterizing parametric design applications.

6.2. PARAMETRIC DESIGN PROBLEM SPECIFICATIONS

In the previous section I informally described the design activity as the construction of an artefact given a set of building blocks, constraints, needs, and desires. More precisely a parametric design application can be characterized as a mapping from a six-dimensional space <P, Vr, C, R, Pr, cf> to a set of solution designs, $\{D_{sol1},......., D_{soln}\}$, where

P = Parameters = $\{p_1,......, p_n\}$;

Vr = Value Ranges = $\{V_1,......, V_n\}$, where $V_i = \{v_{i1},....., v_{il}\}$;

C = Constraints = $\{c_1,....., c_m\}$;

R = Requirements = $\{r_1,......, r_k\}$;

Pr = Preferences = $\{pr_1,......., pr_j\}$;

cf = Cost Function.

In the next sections I will illustrate and formalize these concepts. Before doing this I should however emphasize that while my concern here is with the *formal design problem* - how to construct a solution design from a formal task specification - there is of course more to design than mapping design models to input specifications. In particular task specification is itself a complex, collaborative process during which various stakeholders negotiate a common view of a design problem (Ehn, 1989; Greenbaum and Kyung, 1991). Moreover, this negotiation process, often called *problem framing* (Schoen, 1983) is typically an iterative process, which is intertwined with both problem solving and design evaluation (Bonnardel and Sumner, 1996). Hence, the fact that the work presented here is concerned exclusively with the formal design problem should not be taken as implying that the other aspects of the design process are less important or that the design life-cycle can be characterized by means of a waterfall model where design formulation and problem solving are carried out sequentially.

6.2.1. Parameters and design models

6.2.1.1. Types of design models

In accordance with the problem space model - see figure 3.5 - the parametric design process can be characterized as a search through a space of possible *design states*, where each design state is uniquely defined by an associated *design model*, consisting of an assignment of values to a set of parameters. In the rest of the paper I will use the notation $D_k = \{<p_i, v_{ij}>\}$, where $p_i \varepsilon P$ and $v_{ij} \varepsilon V_i$, to indicate a design model. For a given parametric design specification, <P, Vr, C, R, Pr, cf>, it is possible to define the following types of design models.

- A design model, say D_k, is *complete* if each parameter in P has a value in D_k.
- A design model is *consistent* if it does not violate any constraint in C.
- A design model is *suitable* if it satisfies all requirements in R.
- A design model is *valid* if it is suitable and consistent.
- A design model is a *solution* if it is complete and valid.
- A solution design model, say D_{sol-k}, is an *optimal solution* if there is no other solution, say D_{sol-j}, such that $cf(D_{sol-j}) < cf(D_{sol-k})$ - i.e. no design model has a cost lower than D_{sol-j}.

Figure 6.4 shows the taxonomy defined by the different types of design models. The rounded rectangles in the figure indicate classes while the arrows denote `subclass-of` links.

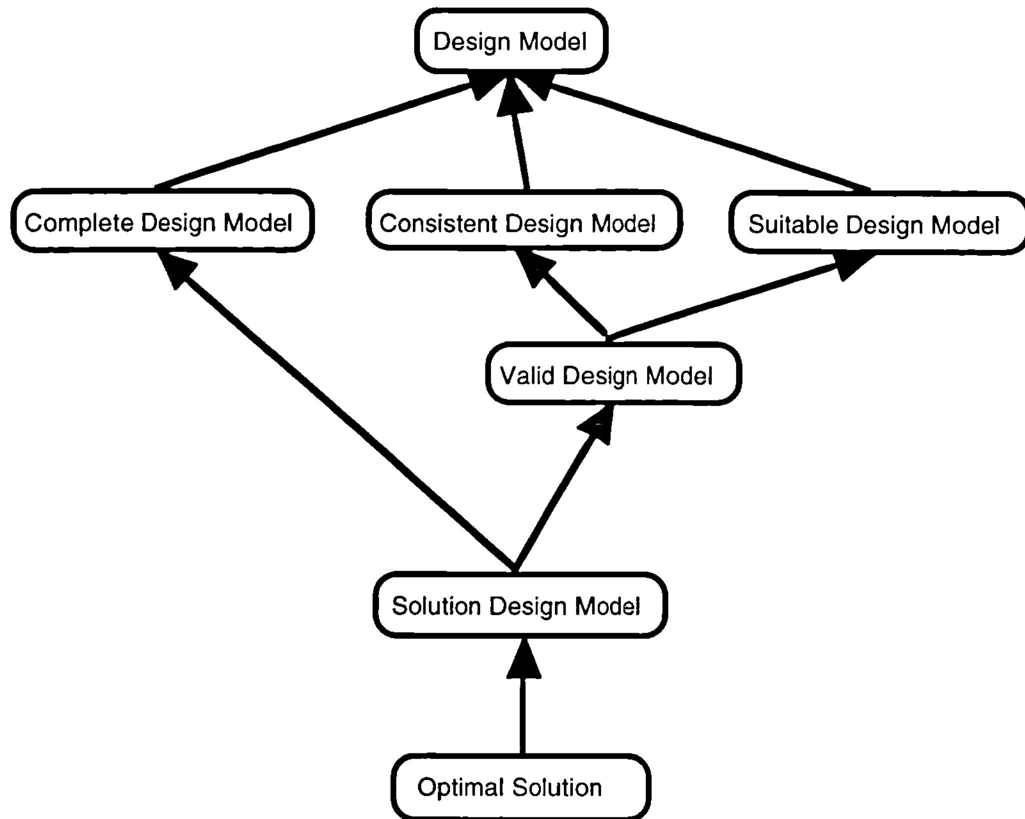


Figure 6.4. Taxonomy of design models

6.2.1.2. Legal values

Each parameter, say p_i, is associated with a value range, V_i, which specifies the set of legal values which can be assigned to p_i. Strictly speaking the concept of value range is not needed in the specification of parametric design problems, given that a value range is just a particular type of constraint. However, because value ranges are typically specified separately from other types of constraints (in particular when a constraint-satisfaction style of problem specification is used) it is convenient and quite natural to explicitly distinguish them from other types of constraints.

In the following I will use the notation $V = \{V_1 \cup ..... \cup V_n\}$ to indicate the union of all value ranges - i.e. the set of all legal parameter values.

6.2.2. Constraints and requirements

A constraint specifies a condition which has to be satisfied by a design. For instance, the VT elevator design application includes constraints such as "The cab height must be between 84 and 240 inches, inclusive". Requirements are also constraints and, as discussed in (Wielinga et al., 1995), the difference between requirements and constraints is conceptual rather than formal. Requirements typically have a 'positive' connotation, in the sense that they describe the desired properties which must be satisfied by the solution, while constraints have a 'negative' connotation, in the sense that they limit the space of

admissible designs, by expressing the applicable technological, physical, or legal restrictions. Moreover, constraints normally specify case-independent restrictions, while requirements tend to be case-specific. For instance, a requirement in the VT application specifies the desired capacity of the target elevator, which is of course case-specific. In what follows I will use the term *design prescription* to refer generically to a requirement or a constraint.

6.2.3. Key design parameters.

Assuming an average of k possible values for a parameter, and n parameters, it follows that the size of the design space associated with a parametric design application is k^n. This of course is a very large number even for relatively small applications. However, it turns out that many parameters do not contribute to the size of the search space, given that there is no degree of freedom associated with their assignment. This is the case when the possible value of a parameter is functionally determined by a constraint or requirement. For example, the value of the parameter 'door operator weight' in the VT domain (Yost & Rothenfluh, 1996) is calculated as the sum of the door operator engine weight and the door operator header weight. This means that from a design point of view the value of this parameter is functionally determined by a domain constraint, and therefore a design problem solver does not need (or indeed cannot) make any decision about it. In what follows I will use the expression *functionally bound* to denote parameters whose value is uniquely determined by a functional constraint or requirement. The parameters which are not functionally bound are called *key design parameters*. Key design parameters and their possible values define the degrees of freedom in the design process and, consequently, the 'real' size of the design space. In other words, the essential decision-making activity during the design process is to decide upon the values to be assigned to key design parameters; the values of the other parameters can then be determined by propagating the relevant functional constraints.

Requirements and constraints which functionally determine parameters are called *constructive*. Non-constructive constraints and requirements are referred to as *restrictive*. The role of restrictive requirements and constraints in the design process is to eliminate certain combinations of parameter values.

It is important to point out that deciding which are the key design parameters is not just a matter of analysing the syntactical format of constraints and requirements; domain knowledge is needed to distinguish key parameters from others. For instance, the expression "door operator weight = door operator engine weight + door operator header weight" can be formulated in three different ways, each of which defines different key

parameters. It is domain knowledge[2] which tells us that the dependent parameter in the expression is door operator weight and not one of the other two.

The concept of key design parameter is useful both because it makes it possible to focus the design process only on the 'important' design parameters and also because it reduces the size of the search space. For example, only 24 of the 199 parameters given in the VT problem specification are key design parameters, which means a reduction in the complexity of the search space from k^{199} to k^{24}, assuming again that on average each parameter has k possible values. Moreover, given a parametric design task specification, it is possible to define an isomorphic problem, which is specified only in terms of key design parameters. Hence, I can take advantage of this property and define parametric design as the process of assigning values to key design parameters. A consequence of this approach is that, with no loss of generality, I can limit the discussion to design models comprising only key design parameters.

6.2.4. 'Better' and 'worse' solutions

The concepts presented so far (i.e. parameters, constraints, requirements, and design models) specify the conceptual vocabulary required to characterize parametric design applications in which the goal is to produce valid, complete designs. However, practical parametric design tasks are not just concerned with finding valid and complete designs, but they often introduce *optimization* aspects. In order to account for these the framework proposed here includes two concepts: *preferences* and *cost function*, which are discussed in the next two sections.

6.2.4.1. Preferences.

Preferences describe task knowledge which, given two design models, D_1 and D_2, is used to specify which of the two - if any - is the 'better' one, in accordance with some criterion. For instance, the specification of the Sisyphus-I office allocation problem (Linster, 1994) includes informal statements such as "Secretaries should be as close as possible to the head of group" and "Project synergy should be maximized". Although one could be tempted to model these statements as requirements, a closer inspection shows that they provide less criteria for distinguishing between solution and non-solution models, than a way of ranking alternative solution models. In other words, to interpret these statements as requirements would unduly restrict the space of solutions. A better approach is therefore to consider the above statements as expressing *preference knowledge* and as defining two criteria by which solutions to the Sisyphus-I problem can

---

[2] Although in this example, one only needs common-sense knowledge to decide on the 'direction' of the constructive constraint.

be assessed. In particular, the first statement can be interpreted as stating that, given two design models for the Sisyphus-I problem, say D1 and D_2, D_1 is 'better' than D_2 if the distance between the secretaries' room and that of the head of the group in D_1 is less than the same distance in D_2.

Of course, deciding whether an element of a design specification indicates a requirement or a mere preference is very much the result of an iterative analysis and negotiating process between designers and clients and it is outside the scope of the present discussion. The important aspect here is that the triadic partition of the problem specification into preferences, constraints, and (proper) requirements provides an adequate conceptual framework to analyse and represent a parametric design problem.

6.2.4.2. Global and preference-specific cost functions

As illustrated by the aforementioned statements concerning the Sisyphus-I problem, a task specification typically includes a number of preferences which express different criteria for distinguishing good from 'less good' solutions. In order to harmonize the possible multiple preference criteria uncovered during the analysis of a design application, it is useful to introduce the notion of *global cost function*, to provide a global criterion for ordering solution designs. Such a criterion is normally constructed by combining preference-specific cost criteria. In order to do this, it is necessary to specify preference-specific cost functions first, expressing the cost criteria defined by each preference. Having done this, it is then possible to define the global cost function for a parametric design application as the combination of the preference-specific cost criteria. More precisely, given a task specification <P, Vr, C, R, Pr, cf>, where Pr = $\{pr_1,......., pr_j\}$, the global cost function is defined as cf = $cf(pr_1)$ o......o $cf(pr_j)$, where $cf(pr_i)$ is the cost function associated with preference pr_i and the symbol 'o' indicates a 'combination' operator. The main constraint on such combination operator is that the resulting global cost function has to be *admissible*. Formally, a global cost function, say cf, is said to be admissible if, for each local preference pr_i, the extension of the partial order induced by cf over the space of design models is a consistent superset of the extension of pr_i.

A simple example of a global cost function can be constructed by combining the local cost functions associated with the two Sisyphus-I preferences discussed earlier. In particular the degree of project synergy of a solution to the Sisyphus-I problem can be measured by scoring negatively (say -1) each shared allocation which involves researchers belonging to different projects and positively (say +1) each shared allocation which violates this

criterion[3]. The criterion for the other preference - minimizing the distance between the secretaries and the head - can be measured simply by counting the number of offices which separate them. These two local cost functions can be combined either by making the two measures commensurable (thus producing an *Archimedean* cost function), or by giving priority to one over the other (*non-Archimedean* cost function). In this second case, assuming that the 'closeness' preference is deemed to be more important than the 'project synergy' one, I can define the overall cost criterion as a two-dimensional vector, $<cf_1, cf_2>$, where cf_1 is the cost function associated with the 'closeness' preference, and cf_2 is the cost function associated with the 'synergy' preference.

In many real-world problems the cost of a design is defined in financial terms. A simple scenario is one in which the cost of a design is given by the sum of the monetary costs of its design elements. More complex metrics can be devised to account for additional, hidden costs, such as the cost of the design process itself or the expected maintenance costs. For example, the cost criterion used by VT design experts characterizes the cost of a solution design in terms of the 'distance' between this design and an ideal one. This distance is computed by considering all the 'corrective steps' (*fixes* in the VT domain terminology) required to produce a design solution. Each fix is given a weight - from 1 to 10 - which indicates how 'expensive' the fix is. Thus, the cost of a design is obtained by combining the costs of each individual fix. Intuitively, the idea here is to provide a cost function which makes it possible to integrate different types of costs, e.g. those deriving from changes to the specification, the monetary cost of each component, and the expected maintenance cost. In (Zdrahal and Motta, 1995) we formalized the informal criterion given in the VT problem specification by means of a number of different metrics, which used both Archimedean and non-Archimedean criteria for combining the cost of each individual fix.

Formally a cost function, whether global or preference-specific, defines a mapping from a design model to a *cost*. This can be a number or, in the case of non-Archimedean costs, a n-dimensional vector.

6.2.5. Summing-up

The description of preferences and cost functions concludes the analysis of the conceptual structure of a parametric design task. Essentially this is a design problem characterized by a parametrized solution template. Constraints, requirements, preferences, and cost functions provide the additional concepts required to structure the space of designs, guide

---

[3] The association between good assignments (i.e. assignments which maximize synergy) and negative numbers may sound counterintuitive. The rationale here is to score bad assignments in such a way that they increase the overall cost of the design model.

the design process, and distinguish solution from non-solution models and more desirable from less desirable ones. In what follows I will formalize these concepts by representing them as elements of a task ontology for parametric design.

6.3. A GRAPHICAL OVERVIEW OF THE PARAMETRIC DESIGN TASK ONTOLOGY

Figure 6.5. gives a graphical representation of the main classes in the parametric design task ontology. Double-headed arrows are used in the figure to indicate that a class is defined as the powerset of another class. For instance, the double-headed arrow linking class `design-model` to class `parameter-assignment` indicates that a design model is a set of parameter assignments. A parameter assignment is in turn defined as a pair, whose first element is a parameter, and the second is a legal value. A labelled thin arrow from one class, say A, to another one, say B, indicates that B defines the type of the possible values of the attribute of A indicated by the label. For instance, this notation is used to indicate that the domain of a cost function is a design model and that its range is a cost. A dashed arrow indicates the value of an attribute. In particular the dashed arrow shown at the bottom of figure 6.5 indicates that the value of the attribute `proves-relation` of class `prefer-expression` is the constant `prefer`. Constants and instances are represented by means of rectangles. Finally, I use a shadow to indicate a concept which is not part of the parametric design ontology but belongs to a different one. In this case, all the shadowed concepts are part of the OCML base ontology.

As shown in the figure, constraints, requirements and preferences are *reified*, i.e. are represented as objects, and associated with the relevant expression by means of a `has-expression` link. Moreover, the ontology precisely defines the syntactic format of a preference or prescriptive expression by introducing the classes `prefer-expression` and `legal-prescriptive-expression`.

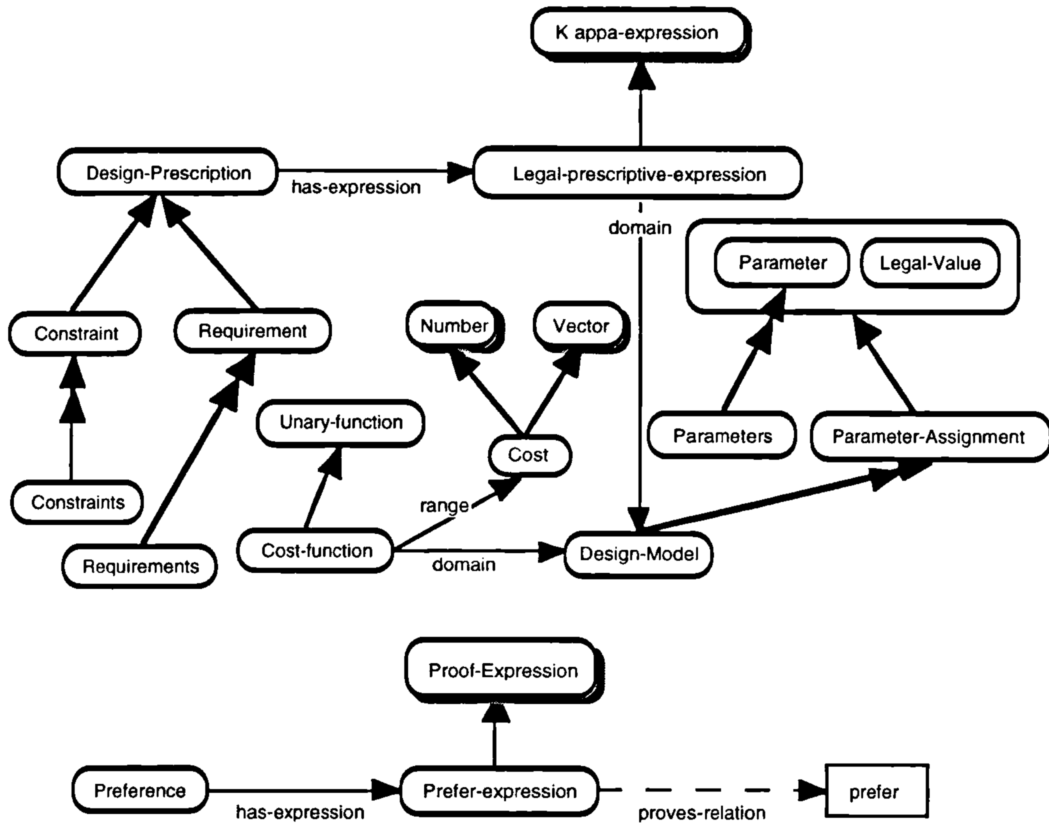


Figure 6.5. Main classes in the parametric design task ontology

Figure 6.6 shows the main functions and relations defined in the ontology. Both functions and relations are represented as rounded rectangles with a thin border and a grey fill pattern. Each relation and function is linked to the classes defining the argument types. In the case of a function I use an arrow to distinguish its range from its input arguments. In particular figure 6.6 shows that the range of function `parameter-value` is a legal value.

An important aspect of the ontology is that the value of a parameter is not an attribute of a parameter, but the mapping from a parameter to a value is mediated by a design model. This approach makes it easier to reason about multiple design models. The same approach is used to define the relations `bound-parameter` and `unbound-parameter`. Whether a parameter is bound or not depends on a particular design model.

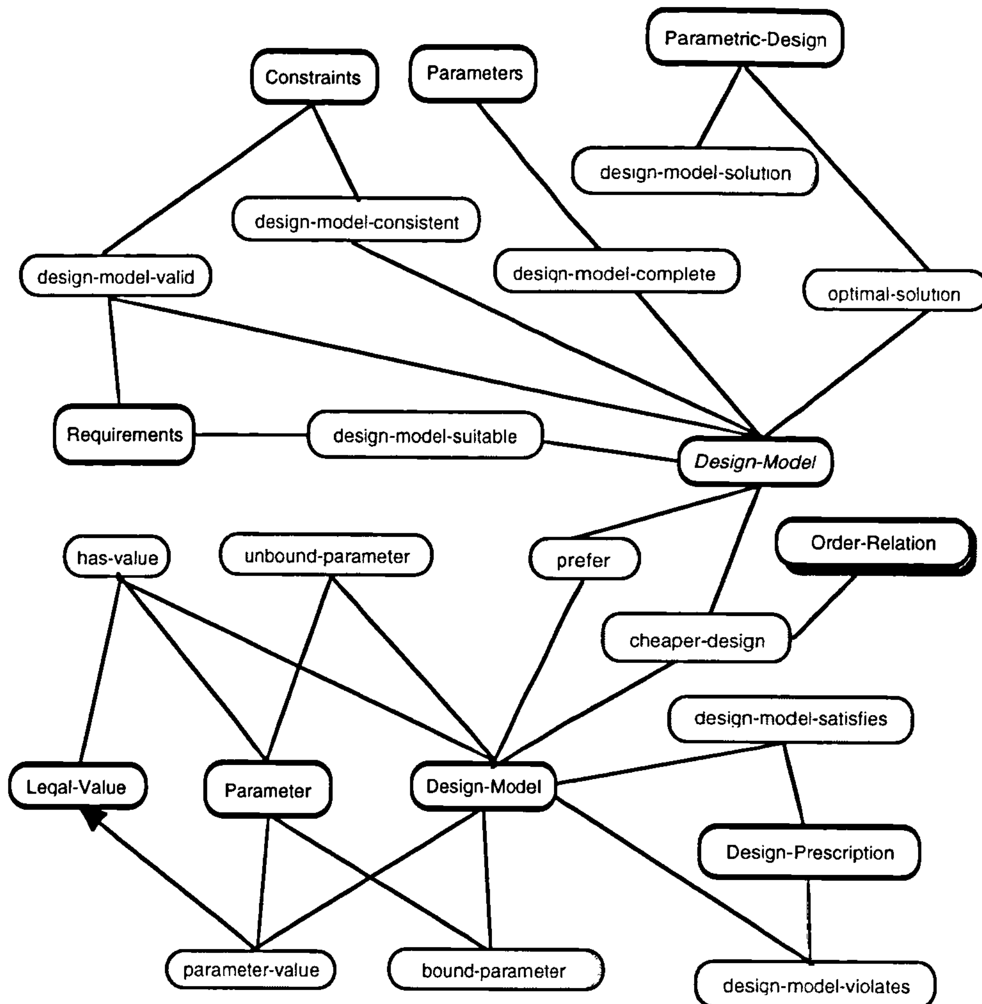


Figure 6.6. Main relations in parametric design ontology

6.4. AN OCML ONTOLOGY FOR PARAMETRIC DESIGN TASKS

In this section I illustrate the OCML model of parametric design. To avoid unnecessary prolixity, here I will only introduce a limited set of constructs, which illustrate the main modelling decisions taken when developing the ontology. The complete ontology can be found in appendix 3.

6.4.1. Modelling the notion of parametric design task

The obvious starting point of a task ontology is the definition of the generic task in question, in this case parametric design. This was already given in section 5.4.2 and is shown again below for convenience.

```
(def-task parametric-design (design-task) ?task
  ((has-input-role :value has-parameters
                   :value has-constraints
                   :value has-requirements
                   :value has-cost-function
                   :value has-cost-algebra
                   :value has-preferences)
   (has-output-role :value has-design-model :cardinality 1)
   (has-design-model :type design-model :max-cardinality 1)
   (has-parameters :type list :cardinality 1)
   (has-constraints :type list :max-cardinality 1)
   (has-requirements :type list :max-cardinality 1)
   (has-preferences :type list :max-cardinality 1)
   (has-cost-function :type cost-function :max-cardinality 1)
   (has-cost-algebra :default-value '(+ - <) :cardinality 1)

   (has-goal-expression
    :type legal-parametric-design-goal
    :default-value  (kappa (?task ?design-model)
                           (design-model-solution
                            ?design-model
                            ?task))))
```

The definition of the parametric design class given above follows straightforwardly from the discussion in section 6.2: there are only two aspects which deserve attention. The first one concerns the inclusion among the input roles of a *cost algebra*. This is a triple which specifies the functions and relation to be used to merge and subtract design costs and to compare the costs of different design models. The second one concerns the specification of the goal of a parametric design problem. As pointed out earlier the goal of a parametric design problem is to find a valid and complete design model. However, in practical design applications optimization aspects are typically very important: given a set of requirements and constraints we normally wish to find the cheapest design which does the job. Therefore the above definition specifies only a default goal for parametric design problems (to find a solution design model) and imposes a type constraint that a parametric design goal should be a sub-relation of relation `design-model-solution` - see definition of class `legal-parametric-design-goal` below. In other words, the above definition only imposes a minimal requirement on the characteristics of a solution design model. Subclasses or instances of this class are free to impose further restrictions on the space of feasible solutions.

```
(def-class LEGAL-PARAMETRIC-DESIGN-GOAL (?rel)
  :iff-def (and (binary-relation ?rel)
                (subrelation-of ?rel
                                (inverse design-model-solution))))
```

Finally, as an example, in the box below I show the specification of the Sisyphus-I office allocation task - see chapter 9 for more details on this application.

```
(def-instance sisl-pardes-task parametric-design
  ((has-cost-function compute-sisl-cost)
   (has-cost-algebra '(merge-sisl-costs
                       subtract-sisl-costs
                       cheaper-room-allocation))
   (has-parameters (setofall ?X (yqt-parameter ?X)))
   (has-constraints (setofall ?X (yqt-constraint ?X)))
   (has-requirements (setofall ?X (yqt-requirement ?X)))
   (has-preferences (setofall ?X (yqt-preference ?X)))))
```

6.4.2. Representing design models

As indicated by the graphical notation used in figure 6.5, a design model is a set of parameter assignments. Sets are represented in OCML either as lists with no duplicates or in terms of a set membership relation, which is true for any element of the set and false for any other tuple. This approach to set representation has the advantage that it makes it possible to represent sets of infinite cardinality.

In particular, the membership relation of a design model, say `?d`, is defined as a binary relation which is true for an assignment `(?p . ?v)` in `?d` if and only if `?v` is the value of `?p` in `?d`.

```
(def-class DESIGN-MODEL (set) ?d
  "A design model is defined as a functional set of
   parameter-assignments. A design model is associated with a
   binary membership function, whose first arg is a parameter, and
   the second is its value in the design model"
  ((membership-test :type binary-relation :max-cardinality 1))
  :iff-def (and (= ?pairs (setofall ?pair
                                    (element-of ?pair ?d)))
                (every ?pairs parameter-assignment)))
```

For instance the definition of a design model for the KMI office allocation problem defines the KMI design model in terms of the relation `kmi-parameter-value`. This is satisfied by tuples of the form `(?p ?v)`, where `?p` is the name of a KMI member, `?v` is an office in KMI, and `?v` has been assigned to `?p`.

```
(def-instance kmi-design-model design-model
  ((membership-test kmi-parameter-value)))
```

A parameter assignment is defined as a pair `(?p . ?v)`, where `?p` is a parameter and `?v` a legal value. The relation == is used in the definition to specify a 'strong' unification test. In OCML the default unification test, =, succeeds in cases where the first argument is an unbound variable and the second a list which does not contain the first argument. Hence, in order to ensure that the membership condition for a parameter assignment is satisfied only by pairs of the form `(?p . ?v)` the definition below uses ==, rather than =.

```
(def-class PARAMETER-ASSIGNMENT () ?pair
  "A parameter assignment is a pair (?p . ?v),
   where ?p is a parameter"
  :iff-def (and
              (== ?pair (?p . ?v))
              (parameter ?p)
              (legal-value ?v)))
```

Having defined classes `design-model` and `parameter-assignment` I can then introduce the relation `has-value`, which models the assignment of values to parameters. A parameter `?p` has value `?v` in a design model `?dm` if and only if the pair `(?p . ?v)` is an element of `?dm`.

```
(def-relation HAS-VALUE (?p ?v ?dm)
  "Parameters have values w.r.t a particular design model"
  :iff-def (and (parameter ?p)
              (design-model ?dm)
              (element-of (?p . ?v) ?dm))
  :prove-by (element-of (?p . ?v) ?dm))
```

6.4.3. Representing constraints and requirements

As already discussed, the difference between constraints and requirements is conceptual rather than formal. Therefore they can share the same representation and in the ontology I model them as distinct subclasses of a class representing a generic *design prescription* - see definitions below.

```
(def-class DESIGN-PRESCRIPTION () ?c
  "The definitions common to constraints and requirements.
   A design prescription is characterized in terms of the associated
   expression.  This is a meta kappa expression predicated over a
   design model"
  ((has-expression :cardinality 1
                   :type legal-prescriptive-expression)))
```

```
(def-class CONSTRAINT (design-prescription)
  :lisp-class-name constraint)

(def-class REQUIREMENT (design-prescription)
  "A requirement is characterized in the same way as a constraint.
   The difference here is conceptual, rather than logical"
  :lisp-class-name requirement)

(def-class LEGAL-PRESCRIPTIVE-EXPRESSION (kappa-expression)
  ?exp
  "This is an expression parametrized over one argument, which denotes
   a design model"
  :iff-def (and (kappa-expression ?exp)
                (== ?exp (kappa ?schema ?sentence))
                (= ?vars (all-free-vars-in-sentence ?sentence))
                (= (length ?vars) 1)
                (= (length ?schema) 1)
                (= (namestring (first ?schema))
                   (namestring (first ?vars)))))
```

Both constraints and requirements are reified, i.e. they are first-class objects. This approach, which is used in the Ontolingua model of the VT domain (Gruber et al., 1996), has a number of advantages. It makes it possible to reason about constraints and requirements, to attach properties to them, and to specialize them for specific classes of applications.

Each constraint and requirement is associated with an expression, whose truth status indicates whether or not the associated constraint or requirement is satisfied by a design model. The form of the expression is a *kappa-expression*. The class `legal-prescriptive-expression` specifies the type of expressions allowed as constraint or requirement expressions. In particular, it checks that the kappa expression is parametrized by only one variable, which denotes a design model. This must be the only free variable in the expression.

An example of a constraint, taken from the Sisyphus-I office allocation problem, is given below. It specifies that smokers and non-smokers should not share.

```
(def-domain-instance smoker-constraint yqt-constraint
  ((has-expression (kappa (?p ?r)
                          (not (exists (?x ?y)
                                       (and
                                        (in-room ?X ?r)
                                        (in-room ?y ?r)
                                        (<> ?x ?y)
                                        (yqt-member ?x smoker ?v)
                                        (yqt-member ?y smoker ?u)
                                        (<> ?v ?u))))))))
```

Finally, the relations `design-model-satisfies` and `design-model-violates` formalize the notions of satisfying/violating a design prescription.

```
(def-relation design-model-satisfies (?dm ?c)
   :constraint (and (design-prescription ?c))
   :iff-def (holds (the ?x
                        (has-expression ?c ?x))
                   ?dm))

(def-relation design-model-violates (?dm ?c)
   :constraint (design-prescription ?c)
   :iff-def (not (holds (the ?x
                             (has-expression ?c ?x))
                        ?dm)))
```

6.4.4. Modelling preferences

A preference is also represented as a reified object associated with a *preference expression*. This is a particular type of *proof expression*. A proof expression is best seen as a Prolog clause. The definition of class `prefer-expression` - shown below - specifies that the denotation of a prefer expression is a backward clause associated with goals of type `prefer`. In practice, this means that preferences over design models are expressed by means of the relation `prefer`. This is a binary relation which defines a partial order over design models.

```
(def-class PREFERENCE () ?p
  "A preference defines an order over two design models.  The
   difference between a preference and a constraint or requirement is
   that these distinguish good from bad models, while preferences
   distinguish between better and worse models."
  ((has-expression :cardinality 1 :type prefer-expression)))
```

```
(def-class PREFER-EXPRESSION (proof-expression) ?exp
  "A prefer expression is a backward rule clause which tries to
   prove a prefer relation instance"
  ((proves-relation :value prefer))
  :constraint (and (== ?exp (?tail if . ?rest))
                   (== ?tail (prefer ?d1 ?d2))))

(def-relation PREFER (?d1 ?d2)
  "Use this relation to express preferences between design models"
  :constraint (and (design-model ?d1)(design-model ?d2)))

;;;Prefer defines a partial order
(tell (order-relation prefer))
```

The advantage of representing preference statements as proof expressions is that this solution provides a modular way to specify conditional preferences - see discussion about modularity and backward rules in section 4.4.8.1. For instance, the definition below shows the OCML representation of the aforementioned Sisyphus-I preference which states that secretaries should be as close as possible to the head of the group. Other preferences can be added to the model by defining additional instances of class `preference`.

```
(def-instance secretary-preference yqt-preference
  ((has-expression ((prefer ?d1 ?d2)
                    if
                    (secretary ?sec)
                    (head-of-group ?head)
                    (element-of (?sec . ?sec-room1) ?d1)
                    (element-of (?head . ?head-room1) ?d1)
                    (element-of (?sec . ?sec-room2) ?d2)
                    (element-of (?head . ?head-room2) ?d2)
                    (< (compute-distance
                         ?sec-room1 ?head-room1 nil)
                       (compute-distance
                         ?sec-room2 ?head-room2 nil))))))
```

6.4.5. Modelling costs and cost functions

A cost function is simply a function which maps a design model to a cost. A cost can be a real number or a vector. This definition is 'open' in the sense that it leaves open the possibility of using alternative cost representations - e.g. qualitative values.

```
(def-class COST-FUNCTION (unary-function) ?cf
  "A cost criterion is a function which takes a design model and
   returns its cost.  The output can be either a real number or a
   vector"
  :iff-def (and (domain ?cf design-model)
                (range ?cf cost)))

(def-class COST () ?x
  "The costs I use are always real numbers of vectors.
   This definition leaves other possibilities open"
  :sufficient (or (real-number ?x)
                  (vector ?x)))
```

Earlier I pointed out that a cost function is constructed by combining preference-specific criteria. Here I formalize this requirements by introducing two axioms in the parametric design task ontology, which ensure that i) each preference-specific order relation is covered by the cost function (axiom `cost-subsumes-preferences`) and ii) that there is no contradiction between any preference and the order over design models specified by the cost function (axiom `cost-preferences-consistency`).

```
(def-axiom COST-SUBSUMES-PREFERENCES
  "This axiom states that the cost function subsumes
   each preference. That is, the cost function must be
   constructed by 'combining' preference-specific
   cost functions"
  (forall (?d1 ?d2 ?pr)
          (=>
             (and (parametric-design ?task
                                     has-preferences ?prs
                                     has-cost-function ?cf)
                  (has-cost-order-relation ?task ?rel)
                  (member ?pr ?prs)
                  (has-expression ?pr ?exp)
                  (proves ?exp `(prefer ?d1 ?d2)))
             (cheaper-design ?rel ?d1 ?d2)))

(def-axiom COST-PREFERENCES-CONSISTENCY
  "This axiom states that the cost function should not
   contradict any partial order expressed by preferences"
  (forall (?d1 ?d2)
          (=>
           (and (parametric-design ?task
                                   has-preferences ?prs
                                   has-cost-function ?cf)
                (has-cost-order-relation ?task ?rel)
                (cheaper-design ?rel ?d1 ?d2))
           (not (exists ?pr
                        (member ?pr ?prs)
                        (has-expression ?pr ?exp)
                        (proves ?exp `(prefer ?d2 ?d1)))))))
```

These definitions conclude the overview of the OCML model of parametric design tasks. In the next section I will compare and contrast this model with alternative proposals in the literature.

6.5. COMPARISON WITH OTHER APPROACHES

6.5.1. Comparison with configuration design ontology by Gruber, Olsen, and Runkel.

A number of ontologies were developed by Gruber et al. (1996), to provide a common, formal data set to all participants to the Sisyphus-II elevator design initiative (Schreiber and Birmingham, 1996). These ontologies are formalized in Ontolingua (Gruber, 1993) and include both a generic model of configuration design and a specialized ontology for representing the VT domain. Here, I will compare and contrast my ontology for parametric design with the configuration design ontology proposed by Gruber et al.

When describing their ontology, Gruber et al. list a number of design goals. These include the aim to make the ontology as much as possible independent from any domain and problem solving method, and to minimize ontological commitments - i.e. to include only the minimal distinctions that are "necessary to specify the class of configuration problems under consideration". These design goals also motivated the definition of my ontology.

The ontology by Gruber et al. looks at configuration design problems rather than just parametric design ones. For this reason it includes models of components, subparts, and *component assemblies*. I did not tackle these aspects in my ontology, which treats parameters as an unstructured collection. However, the component-related part of the ontology by Gruber et al. is defined separately and therefore it could be possible to try and integrate it with my ontology. On the other hand my ontology provides a more fine-grained framework for modelling parametric design problems. For instance the ontology by Gruber et al. does not account for essential aspects of a parametric design task specification, such as requirements and preferences.

From a general point of view, the main 'epistemological' difference between the two ontologies is that mine centres on design models, while the one by Gruber et al. centres on components. These different foci reflect my preoccupation with parametric design problems and theirs with configuration ones.

Another important distinction between the two ontologies concerns the treatment of parameters. Gruber et al. associate parameters directly with values. In my ontology this association is instead mediated by the notion of design model. I believe my approach is more flexible: it makes it easier to reason about alternative design models and consequently to support the use of methods such as A*, which store and examine multiple search states.

6.5.2. Comparison with DIDS approach

The DIDS system (Balkany et al., 1994; Runkel et al., 1992; 1994; 1996) provides domain-independent support for building design applications. The system is based on a generic model of configuration design problem solving, which defines the generic data structures and tasks, called *mechanisms* in the DIDS terminology, required for building design applications. As in the case of the ontology developed by Gruber et al. DIDS also focuses on configuration, rather than parametric design problems.

The DIDS framework distinguishes between *functions*, *parts*, *constraints*, and *preferences*. Functions and parts generalize from the notions of parameter and value range discussed here. Parts can have connections to other parts and they can be either abstract or concrete. Parts correspond to parametrized components in the ontology by Gruber et al.

The main difference between the DIDS approach and mine has to do with the philosophy underlying the two approaches. Here I am interested in building a formal framework for characterizing parametric design applications. Therefore the proposed ontology is formal, semantically rich, method-independent and specified at the knowledge level. The DIDS approach has more of an engineering flavour. Their set of concepts is neither characterized in a method-independent way nor is specified in an implementation-independent style. Thus, there is no reusable task ontology in DIDS in the sense used here.

6.5.3. Comparison with work by Wielinga, Akkermans, and Schreiber

Parametric design is formally analysed in (Wielinga et al., 1995). In this paper the authors define a parametric design task specification and refine the associated *competence theory* (Van de Velde, 1988) to derive properties of a Propose&Revise problem solving method. In chapter 3 I discussed this work extensively focusing on the aspects related to the development of a problem solving method from a task specification. In the context of this chapter, the relevant aspect of the work by Wielinga et al. is their characterization of the parametric design task and its relationship with the ontology proposed here.

The task ontology discussed by Wielinga et al. emphasizes the role that a domain theory plays in defining a parametric design task. A domain theory specifies the entities and the relationship in the problem in hand and makes it possible "to derive information about the relation between the solution, requirements and constraints". In particular a domain theory normally implies constraints and requirements which are not part of the problem specification. For instance, when giving a specification for a house, the client normally does not include 'obvious' requirements, such as water and electricity.

Wielinga et al. specify a comprehensive framework for describing design applications. It includes requirements, constraints, design structures, parameters, assignments, and *design choices* - these express preferences over preferred solutions - as well as the general notion of domain theory.

The main difference between their approach and mine is the level of granularity. Their approach is characterized at a very abstract level. The various concepts are informally illustrated at some length, but their definitions are not detailed. For instance a design structure is simply characterized as a set of tuples. The reason for this coarse granularity is that the paper is not so much concerned with modelling these notions, as much as using them to characterize formally the process of developing a problem solver for parametric design. In contrast with the work by Wielinga et al., the role of the ontology presented here is not only to provide a conceptual framework but also a practical reusable resource for modelling parametric design problems.

In a recent paper (1997) Wielinga and Schreiber have proposed a classification of types of configuration design tasks in terms of a 3x3 grid, parametrized by the properties of the design components, the way the components can be arranged, and the nature of requirements and constraints. Within this grid, they characterize parametric design as a configuration design task characterized by a fixed set of parametrized components and a fixed assembly. This definition appears to be over-restrictive. It is not necessary for the assembly to be fixed as long as the structure of the solution is known in parametrized form at the beginning of the design process. For instance, in the VT problem the assembly is strictly speaking not fixed: some components are not required by some solution designs.

Another problem with the classification proposed by Wielinga and Schreiber is that it appears to subscribe to an 'objective' view of the design task. In my view, whether a task is configuration, parametric design or assignment is not so much a property of the types of components and the types of assembly, but rather is a function of how much knowledge do we have about the solution. For instance, if I use plasticine as my technology for designing, I have of course no constraints on the types of components and on the design assembly. Bits of plasticine can be arranged together in an unconstrained way. However, the 'open-ended' nature of the technology does not necessarily imply that the task is a "full configuration design" one.

Some individual cells of the grid proposed by Wielinga and Schreiber also seem to be problematic. For instance, in their view assignment problems are characterized by a *skeletal* assembly, while parametric design ones are characterized by a *fixed* assembly. This seems counterintuitive given that assignment is a subclass of parametric design - in

other words I would expect that for each given dimension assignment always specializes parametric design.

6.6. CONCLUSIONS

In this chapter I have presented a conceptual and formal framework for representing parametric design problems. The conceptual analysis has highlighted the nature of parametric design applications and the different types of knowledge structures which need to be modelled to build models of parametric design problems. In particular I have illustrated the various types of design models and claimed that the distinction between requirements, constraints, preferences and cost function provides a rich enough framework for capturing the output of the design formulation process. Of course, this claim can be validated only empirically, by constructing parametric design applications. So far this technology has been successfully applied to a number of domains, which include elevator design (Motta et al., 1996), sliding bearing design (Horak et al., 1995), office allocation problems (Motta et al., 1994a), and truck cab design. In chapter 9 I will illustrate in detail an elevator design and two office allocation applications.

The task ontology presented here consists of a set of OCML definitions which formalize the various aspects of a parametric design problem. The ontology provides an important focus for reuse and the basis for a strong, task-driven knowledge acquisition process. In particular the ontology provides not only the constructs needed to model the conceptual structure of parametric design problems - say constraints and preferences - but it also specifies in detail the syntactic machinery required for verifying task models - e.g., the syntax of constraint and preference expressions.

Chapter 7.
A Generic Model of Parametric Design Problem Solving

This chapter describes a generic problem solving model for parametric design. This model consists of a set of generic tasks and an associated method ontology. The tasks can be seen as a set of high-level building blocks for constructing problem solving methods for parametric design. The ontology specifies the minimal commitments which need to be obeyed by any problem solving method for parametric design. In the chapter I illustrate the model in detail and compare it to alternative proposals.

7.1 INTRODUCTION

In the previous chapter I analysed the structure of parametric design applications. The resulting task ontology provides the input to this chapter, where I describe a generic framework for parametric design problem solving. This framework takes the form of a *problem solving model*[1], which decomposes the parametric design problem into a number of generic tasks and proposes default (sub-)methods for carrying them out. The model is associated with a highly generic method ontology, which expresses the minimal knowledge requirements which have to be satisfied by a PSM applicable to parametric design problems.

The claim here is that this collection of generic tasks defines the space of knowledge-intensive, decision-making activities which are carried out when solving parametric design problems. This claim will be validated in the next chapter, where it will be shown that it is possible to describe a number of problem solving methods for parametric design in terms of the proposed framework. Moreover, I will show that the differences between alternative problem solving methods can be accounted for in terms of alternative

---

[1] The term 'problem solving model', rather than 'problem solving method', is used here to emphasize that, like generic algorithmic schemas in conventional software, this model may not be fully specified - e.g. it may abstract from specific control regimes.

refinements of the common method ontology or alternative solutions to one or more of the generic tasks specified in the model.

7.2 A SEARCH-BASED MODEL OF PARAMETRIC DESIGN PROBLEM SOLVING

In this section I introduce the basic concepts which make up the search-based problem solving model for parametric design problem solving.

7.2.1 Design as search

As already mentioned in section 3.4.2 and illustrated by figure 3.5, parametric design problem solving can be characterized as the problem of navigating a *design space* efficiently. A design space is defined in terms of two components: a collection of *design states* - where each design state is uniquely defined by the associated design model, D_i - and a parametric design task. The definition below formalizes this approach.

```
(def-class DESIGN-SPACE () ?x
  "A design space is characterized in terms of a set of design states
   associated with a parametric design task"
  ((associated-with-task :type parametric-design :cardinality 1)
   (has-states :type set :cardinality 1 :default-value nil))
  :constraint (=> (member ?s (the ?set (has-states ?x ?set)))
                  (design-state ?s)))

(def-class DESIGN-STATE () ?c
  "A design state is characterized in terms of the associated
   design model"
  ((has-design-model :cardinality 1
                     :type design-model)))
```

This definition provides a *task-oriented* characterization of a design space, which is consistent with the methodological approach I have adopted to integrate task and method ontologies. This approach is based on the assumption that it is always possible to associate a design space to a parametric design task specification. This assumption is trivially true, as a design space can always be generated from a parametric design task specification by generating the powerset of all possible design models. However, no other assumptions are introduced here, either about the structure of the design space or the availability of search-control knowledge.

The advantage of introducing a task-oriented view of a design space is that this approach makes it possible to move from a task-oriented perspective to a problem solving-oriented one. Specifically, let's consider a design space about which we only know the generic structure of a node. It follows that, in the absence of additional knowledge, a problem solving agent can solve the relevant design task only through search. Thus, by

introducing the notion of design space, we obtain a problem solving framework which is completely method-independent; it only presupposes the existence of a task specification.

Given a state S_i, associated with design model D_i, I will use the notation CV_i to indicate the set of constraints violated by D_i. The notation $cf_i = cf(D_i)$ will be used to indicate the cost of D_i.

7.2.2 State transitions and design operators

7.2.2.1. The role of design operators

Nilsson (1980) characterizes a state space as a triple <S, O, G>, where S is a set of initial states, O is a set of operators, and G is a set of goal states. In contrast with Nilsson's approach, my definition of design space does not consider operators. The reason for this exclusion is that my characterization of a design space is task-oriented - i.e. it provides all the concepts required to acquire task specifications. Operators are not needed at the task level, their role is to support the search process - i.e. problem solving. Thus, the notion of design operator is introduced separately, as part of the generic method ontology for parametric design problem solving.

7.2.2.2. Representing design operators in OCML

A search step in the design space - i.e. a *state transition*, say from state S_i to state S_j - is carried out by applying a *design operator*. Informally, this can be described as an inference mechanism which generates a new design model from the one given as input. Thus, state transitions can be represented as ternary relations, which link two design states through a design operator - see definition below.

```
(def-relation STATE-TRANSITION (?s1 ?design-op ?s2)
   "A state transition associates two design states
    through a design operator.  Only meaningful transitions
    are allowed, where the target state is different from
    the source state"
  :iff-def (and (design-state ?s1 has-design-model ?d1)
                (design-state ?s2 has-design-model ?d2)
                (design-operator ?design-op body ?fun)
                (= ?d2 (call ?fun ?d1))
                (not (= ?d1 ?d2))))
```

The definition of relation `state-transition` restricts its extension only to meaningful moves in the state space, where the target design state is different from the source one. A further restriction which could be imposed here would be to narrow down state transitions to moves in the space of key design parameters. Such a restriction does not change the problem space, given that, as pointed out in the previous chapter, any parametric design problem can be transformed into one specified only in terms of key design parameters. However, here I will not impose such additional restriction, both to

avoid unnecessary complexity in the ontology and also because not all application models discussed in chapter 7, e.g. the VT application, obey such additional requirement.

The definition given below characterizes a design operator in terms of two slots, `has-assumption` and `has-body`. The former specifies a statement which is expected to hold for the application domain where the operator is meant to be used. For instance, the operator used in the Sisyphus-I problem to allocate secretaries assumes that only one head of group exists. The latter specifies a unary function which takes as input a design model, say D_i, and produces as output a design model, D_j, which is different from D_i. Thus, the body of a design operator specifies a transition step indirectly, by relating two design models.

```
(def-class DESIGN-OPERATOR ()
  "A state transition in the state space model specifies a link
   between two design states (in practice two design models).  State
   transitions are carried out by means of design operators."
  ((has-assumption :default-value (true)
                   :type relation-expression
                   :documentation
                   "This slot can be used to specify a statement that is
                    expected to hold for the application domain where
                    the operator is applied.  Assumptions are expected
                    to remain (un-)satisfied during the design process")
   (has-body :type design-operator-body)))
```

Formally, a design operator body is defined as follows.

```
(def-class DESIGN-OPERATOR-BODY (unary-function) ?fun
  "A design operator body is a unary function which takes as input
   a design model, say Di, and produces as output a design model Dj
   such that Di = Dj"
  :no-op (:constraint (and (domain ?fun design-model)
                           (range ?fun design-model)
                           (=> (= (call ?fun ?di) ?dj)
                               (not (= ?di ?dj))))))
```

At any stage of the design process a number of operators can be applicable. In order to support the modelling of meta-knowledge about design operator selection, the method ontology includes a relation, `design-operator-order`, whose definition is as follows.

```
(def-relation DESIGN-OPERATOR-ORDER (?x ?c)
  "This relation can be used to specify an application specific
   ordering of operators"
  :constraint (and (design-operator ?x)
                   (design-operator ?c)
                   (not (= ?x ?c))))

(tell (DEFINES-PARTIAL-ORDER design-operator-order))
```

In the previous chapter I associated cost with design models. However, it is useful to generalize this notion and define the cost of design operators as well - e.g. to be able to characterize the cost model used in the VT application. This can be achieved by defining the cost of an operator as the difference between the cost of the *successor state* and that of the *predecessor state*.[2]

```
(def-function OPERATOR-COST (?op ?task)
  :constraint (and (design-operator ?op)
                   (parametric-design ?task))
  :body (if (and (has-cost-difference-function ?task ?fun)
                 (state-transition ?s1 ?op ?s2))
           (call ?fun ?s2 ?s1)))
```

In the following I will use the notation cf (S_i -> S_j) to indicate the cost of an operator connecting S_i to S_j.

7.2.3 Parameter dependencies

The task ontology described in chapter 5 formalizes parameters as the elements of a design model and associates each parameter with a value range. From a task-oriented point of view this characterization is all one needs to describe parameters. However, if we adopt a problem solving perspective, it is useful to introduce the concept of *dependency* between parameters. Specifically, a parameter, say p_i, depends on a parameter p_j, where $p_i \neq p_j$, if the value of p_j can only be computed when p_i is bound. This dependency can be physical, as in the case of functional constraints such as "door operator weight = door operator engine weight + door operator header weight", or heuristic, as - for example - in the case of the procedure used in the VT domain to determine the appropriate safety beam model from the platform width.

To support the representation of parameter dependencies, the generic method ontology includes two relations, `depends-on` and `affects`. These are defined as follows.

```
(def-relation DEPENDS-ON (?p1 ?p2)
  "This relation models parameter dependencies.
   A parameter, p1, depends on  p2 if p2
   has to be bound in order to compute p1."
  :constraint (and (parameter ?p1)
                   (parameter ?p2)))

(def-relation AFFECTS (?p1 ?p2)
  "The inverse of depends-on"
  :constraint (and (parameter ?p1)
                   (parameter ?p2))
  :iff-def (depends-on ?p2 ?p1))
```

---

2 When two design states, say S_i and S_j, are linked by a design operator pointing to S_j, S_i is said to be the *predecessor* of S_j and S_j the *successor* of S_i.

Definitions such as the ones given above exemplify the basic epistemological difference between task and method ontologies. A task ontology defines the problem to be solved and does not subscribe to any particular problem solving approach. A method ontology introduces the distinctions which are relevant to the problem solving model associated with the ontology. In particular, design operators introduce dependencies between parameters. Therefore it is useful to make these dependencies explicit, by introducing the relevant modelling primitives. However, it is important to emphasize that providing modelling support for expressing dependencies does not imply that only methods which explicitly reason with dependencies can make use of such an ontology. Some methods might ignore this information, others might derive it automatically, others might require it to be asserted explicitly. Thus, formalizing the notion of 'parameter dependency' does not impose additional ontological commitments - in the sense discussed by Gruber (1995) - but rather provides modelling support for those problem solving scenarios which are implied by ontological commitments introduced elsewhere in the ontology (i.e. by the definition of design operators).

The two definitions shown below exploit the dependency network i) to decide when a parameter can be computed and ii) to retrieve all computable, unbound parameters.

```
(def-relation COMPUTABLE (?param ?dm)
  :iff-def (and (parameter ?param)
                (design-model ?dm)
                (= ?l (setofall ?x (depends-on ?param ?X)))
                (every ?l (kappa (?x)
                                 (bound-parameter ?x ?dm)))))

(def-function ALL-COMPUTABLE-PARAMETERS (?params ?dm)
  :body (setofall ?x (and (member ?x ?params)
                          (unbound-parameter ?x ?dm)
                          (computable ?x ?dm))))
```

7.3 METHODOLOGICAL ASPECTS OF PARAMETRIC DESIGN PROBLEM SOLVING

The definitions given in the previous section provide a basic method ontology to discuss parametric design problem solving. This extends the parametric design task ontology by introducing the concepts of 'design state', 'design space', 'design operator', and 'state transition'. While this ontology is obviously still very coarse-grained, it nevertheless provides a starting point for characterizing parametric design problem solving. In particular, the search paradigm and the notion of design operator introduce a problem solving viewpoint into a 'static' task ontology.

Before refining the method ontology it is useful to look at the methodological aspects of design problem solving, in particular discussing how task-level concepts are mapped into method-level concepts.

Figure 7.1 shows the main transformations which occur between task and method concepts. These are discussed in the following sub-sections.

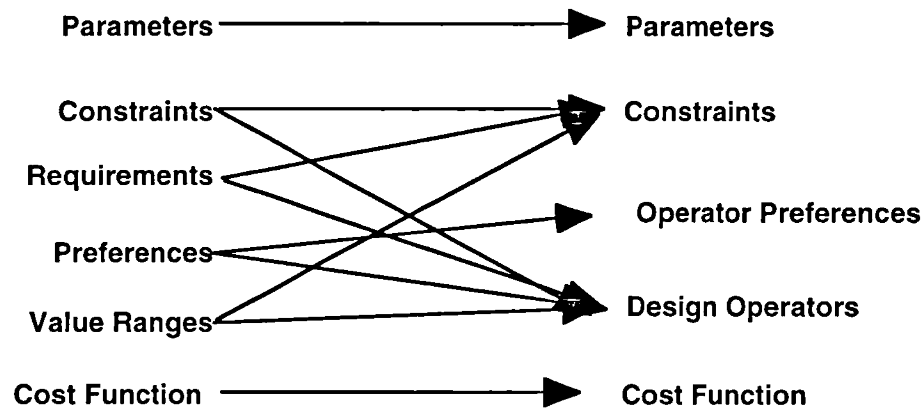


Figure 7.1. Mapping from task concepts to method concepts

7.3.1 Parameters.

The set of parameters acquired when defining a task model normally remains the same when constructing an application model. However, the method ontology introduces additional structure, by organizing parameters into a dependency network. This structure makes it possible to provide general purpose heuristics for deciding which parameter to assign or modify next.

7.3.2 Constraints.

In the previous chapter I distinguished between consistent and suitable models and defined a valid design model as one which is both suitable and consistent. Here I will remove this task-level distinction and I will treat requirements as constraints. The reason for doing this is that while such a distinction is important during the task specification and requirements engineering activities, it is less important during problem solving. When designing we are interested in finding solution models. Unless we are prepared to weaken our requirements, in which case they should be modelled as preferences, there is no operational difference between requirements and constraints.

Some authors, e.g. (Wielinga et al., 1995), point out that one important difference between constraints and requirements in design problems is that the former may or may not be applicable to a particular design solution, while the latter must always be satisfied. While this statement is true for design problems in general, it does not necessarily apply to the class of formal parametric design problems considered here, in which solution

designs are always complete. Moreover, from a modelling point of view it is quite easy to avoid this 'incompleteness of solution' problem, either by explicitly including applicability conditions in the representation of constraints or by providing default values, e.g. `:not-needed`, to those parameters/components which are not required by a solution.

Finally, the value range associated with each parameter, say Vi, becomes a restrictive constraint on the legal values of the relevant parameter in the problem solving model.

7.3.3 Design Operator

A design operator can be constructed in four possible ways. I will label these with A-D letters so that it will be easy to refer to these categories when discussing specific operators in the rest of the thesis.

Type A. If the value range, say V_i, of a parameter, p_i, is enumerable, then a design operator for p_i can be defined as a generator which, given a design state in which p_i is unbound, produces alternative design extensions where p_i is bound to a different element of V_i.

Type B. Functional constraints and requirements can be operationalized into design operators. For instance a functional constraint such as "door operator weight = door operator engine weight + door operator header weight" can be transformed into a design operator which calculates the value for parameter door operator weight.

Type C. If there is a preference, pr_i, which suggests a value for parameter p_i, say v_{ij}, then this preference can be transformed into a design operator extending the input design model with one which includes the assignment (p_i . v_{ij}). When multiple operators exist for a particular parameter, then the relation `design-operator-order` can be used to specify context dependent control knowledge. In particular, this mechanism makes it possible to transform preference ratings into control knowledge.

Type D. Heuristic, problem solving knowledge can be brought in to construct an operator. An example is the operator discussed in section 1.2.2.2, taken from the VT application, which positions the counterweight half way between the platform and the U-bracket. In relation to the VT task specification, this operator does not define a constraint or requirement. It could be possibly characterized as a preference, on the basis that locating the counterweight in a central position has some cost advantages - because less strain is put on the cables, cheaper motors can be employed. Another possibility is that the role of this operator is simply to codify experiential problem solving knowledge - i.e. a central position is a 'good default' for the counterweight. In a nutshell, the knowledge expressed by operators is not necessarily related to the task specification: operators can also be defined by means of application-specific, problem solving knowledge.

7.3.4 Cost Function

As discussed in the previous chapter a cost function provides a global criterion for ranking solutions and its definition is expected to combine the multiple criteria expressed by different preferences. A cost function should not be method-specific and therefore a task-centred cost function is 'inherited' by a problem solver from the relevant task specification. Having said so, when constructing method ontologies it is useful to introduce efficiency-related refinements of the definitions associated with cost evaluation, e.g. commitments which allow the incremental calculation of costs during the design process.

7.4 A GENERIC MODEL OF PARAMETRIC DESIGN PROBLEM SOLVING

In this section I discuss the structure of a generic model of parametric design problem solving. This model takes the form of a partially specified problem solving method which makes use of the method ontology defined earlier in this chapter. The model is described in terms of the task-method framework presented in chapter 5 (i.e. tasks, methods and roles) and is informed by the search-centred view of problem solving illustrated in chapter 1.

There are two main objectives related to this section: i) to identify the main generic tasks which characterize parametric design problem solving, and ii) to provide the 'root node' of a library of methods for parametric design.

The first objective is based on the assumption that there exists a set of generic tasks which is common to different methods for parametric design. This assumption is justified both by theoretical and empirical evidence. From a theoretical point of view the adoption of a search-centred framework constrains the number and the type of feasible problem solving activities. Empirical evidence is provided by existing surveys of design problem solvers (Balkany et al., 1993), which have uncovered generic problem solving activities which are common to different approaches. The idea is that, once an appropriate collection of generic tasks for parametric design problem solving has been abstracted, this will provide i) a set of dimensions to analyse and differentiate problem solving methods for parametric design and ii) a high-level toolkit for constructing new methods.

The second objective has to do with developing a problem solving method for parametric design which subscribes to the given task and method ontologies, exhibits enough complexity to uncover the space of parametric design subtasks but at the same time avoids unnecessary control and ontological commitments. This method, named `gen-design-psm`, provides a kind of 'method template', from which more specialized problem solving methods can be generated. In particular, in the next chapter I will illustrate a number of

methods which were constructed by refining and augmenting the generic problem solving model and which subscribe to a common, generic control structure.

7.4.1 Generic tasks in parametric design problem solving

Given the design space model, there are essentially only four actions which can be carried out: selecting a design state, selecting a design operator, applying a design operator to the selected state, and evaluating the resulting design model. The latter is needed to assess its properties, e.g. whether it provides a solution, its cost, etc. Although these four subtasks are adequate to describe the process of searching the design space, surveys of design applications show - not surprisingly! - that several dozens of different tasks exist which are carried out during design problem solving. For example, the researchers working on the DIDS project (Balkany et al., 1993) have analysed a number of configuration design systems, and classified the various *mechanisms* used by these systems into a number of generic categories: select design extension, make design extension, detect constraint violation, select fix mechanisms, make fix mechanisms, and test if-done. For example they list forty-one "make design extension" mechanisms. While there are, in my view, problems with such a bottom-up approach - see section 7.5 at the end of this chapter for a review of this and other related work - it is clear that a four-task framework is much too coarse-grained. Intermediate concepts are required, which can provide additional structure to the framework and therefore better 'conceptual handles' for representing different problem solving methods in a homogeneous way.

The notions of *design context* and *design focus* serve this purpose and 'bridge the gap' between the selection of a state and the selection of an operator, by introducing intermediate decision-making tasks. Moreover, they provide abstraction mechanisms which make it possible to generalize from different, but essentially isomorphic, method-specific behaviours. For instance, as I will show in the next chapter, the 'propose' and 'revise' phases of a Propose&Revise method essentially exhibit the same inference structure and control regime. The difference is that in the propose phase the design context is design extension, in the latter it is design revision.

Thus, a design context is an abstraction mechanism which, given a design state, provides a generic viewpoint to drive the selection of a design focus and an operator. Another way of looking at a design context is as a description of the generic goal which a problem solver decides to pursue when designing in a particular design scenario (i.e. state).

While the notion of design context makes it possible to abstract problem solving behaviour from the generic properties of a design state, a design focus provides a more fine-grained mechanism to model the design process. In particular, when analysing the behaviour of different design problem solvers, it is easy to see that, at each stage of the design process, a particular element of the design model - e.g. a part, a parameter or a

constraint violation - is selected and becomes the *focus* of the design process. The selection of a design focus depends on the given design context. For instance, a Propose&Revise problem solver focuses on parameters or constraint violations, depending on whether the context is 'propose' or 'revise'.

As in the case of design contexts, the notion of design focus makes it possible to abstract from different but essentially isomorphic problem solving behaviours. Moreover, it also provides an intermediate decision-making step, which increases the granularity of the framework and makes it easier to configure it for specific problem solving methods.

In the rest of this chapter I will illustrate the components of the proposed generic model for parametric design. For each component (task or method) I will provide an informal description and highlight the relevant modelling or problem solving issues. Moreover, I will also include in the description the OCML definitions of the most 'interesting' tasks and methods. The complete specification of the model is given in appendix 4.

7.4.2 Constructing the generic model

Given the model of design as search presented in section 7.2, I can then define a generically applicable control regime, which provides the main control structure of the generic model. This control regime introduces four subtasks, `initialise-design-space`, `select-design-state`, `design-from-state` and `reflect-design-state`. The resulting task-method structure is shown graphically in figure 7.2. The figure also shows the 'root method' of the library, `gen-design-psm`, whose OCML definition is given below. This method simply invokes task `gen-design-control`.

```
(def-class GEN-DESIGN-PSM
  (problem-solving-method-for-parametric-design
   decomposition-method)
  ((has-input-role :value has-design-operators)
   (has-output-role :value has-solution-state)
   (has-solution-state :type design-state)
   (has-design-operators :type design-operator)
   (has-output-mapping
    :value '(lambda (?psm ?state)
              (the ?dm
                (has-design-model ?state ?dm))))
   (has-body :value
             '(lambda (?psm)
                (in-environment
                 ((?s . (achieve-generic-subtask
                         ?psm gen-design-control
                         has-current-pardes-task
                         (the ?task (tackles-task ?psm ?task)))))
                 (if (design-state ?s)
                   ?s)))))
  :own-slots ((has-generic-subtasks '(gen-design-control))))
```

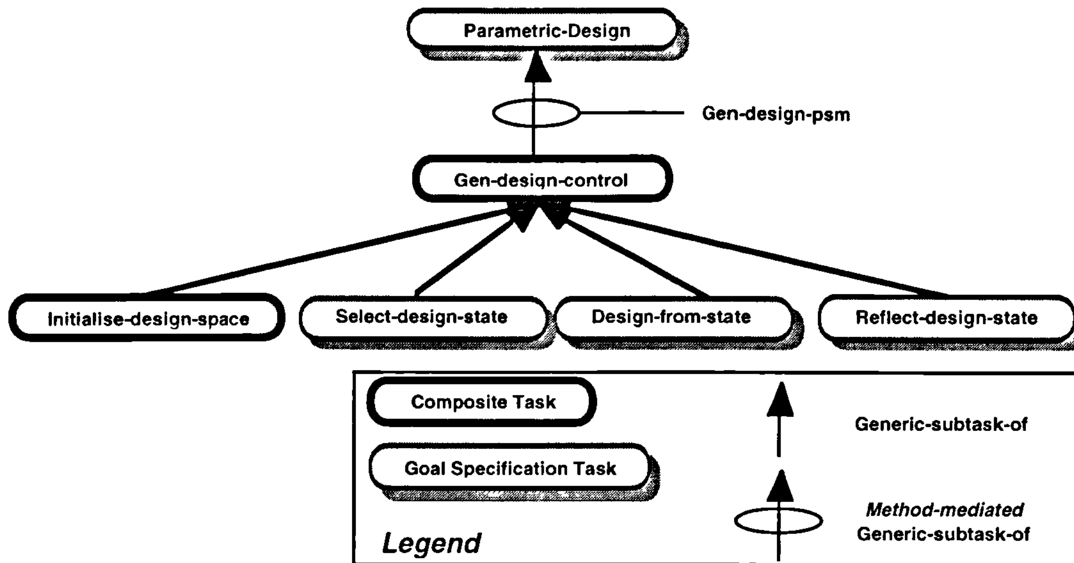


Figure 7.2. Task-Subtask decomposition introduced by generic control task.

An informal specification of task `gen-design-control` is given below. The task takes as input a set of design operators and the specific parametric design task which is being tackled. Its output is a solution design state, i.e. a design state whose associated design model satisfies the goal of the relevant parametric design task. The slot 'control' in the task description template indicates the task's control roles (in this case, Design-space).

```
Generic Task Gen-design-control
  Inputs:     Design-operators, Current-task
  Output:     Design-state
  Control:    Design-space
  Goal:       "To return a state which satisfies the goal of the current task"
  Subtasks:   Initialise-design-space, Select-design-state, Reflect-design-state,
              Design-from-state
  Body:       Initialise-design-space (Current-task) -> Design-space
              Repeat
                 Select-design-state (Design-space) -> Design-state
                 If "Select-design-state fails"
                    then Return () -> Fail
                    else
                      If "Design-state satisfies the goal of the current task"
                         then Return () -> Success
                         else
                           Do
                              Reflect-design-state (Design-state)
                              Design-from-state (Design-state)
```

As shown above, the body of `gen-design-control` first invokes task `initialise-design-space`, which returns the initial design space. This consists of a single design

state, which is associated with an empty design model. Then, tasks `select-design-state`, `reflect-design-state`, and `design-from-state` are executed cyclically, until either a solution state is reached, or state selection fails. This informal definition can be precisely specified in OCML as follows.

```
(def-class GEN-DESIGN-CONTROL (composite-task)
  ((has-input-role :value has-design-operators
                   :value has-current-task)
   (has-output-role :value has-solution-state)
   (has-solution-state :type design-state)
   (has-design-operators :type design-operator)
   (has-current-task :type parametric-design)
   (has-goal-expression
    '(kappa (?self ?result)
      (and (design-state ?result)
           (has-design-model ?result ?dm)
           (achieved (role-value ?self has-current-task) ?dm))))
   (has-body :value
             '(lambda (?self)
                (in-environment
                 ((?design-space . (achieve-generic-subtask
                                    ?psm initialise-design-space
                                    has-current-task
                                    (role-value ?self
                                                has-current-task))))
                 (REPEAT
                  (in-environment
                   ((?state . (achieve-generic-subtask
                               ?psm Select-design-state
                               has-design-space ?design-space)))
                   (if  (= ?state :nothing)
                     (RETURN :nothing)
                     (if (achieved ?self ?state)
                       (design-succeeds ?state)
                       (DO
                         (achieve-generic-subtask
                          ?psm reflect-design-state
                          has-design-state ?state)
                         (achieve-generic-subtask
                          ?psm design-from-state
                          has-design-state ?state)))))))))
  :own-slots ((has-generic-subtasks
               '(initialise-design-space
                 design-from-state
                 reflect-design-state
                 select-design-state))))
```

Task `gen-design-control` provides a very generic control loop, which is shared by all problem solving methods which are included in the library. This approach makes it possible to differentiate between different methods only on the basis of specific solutions to design subtasks, rather than in terms of the overall control regime. The advantage of this solution is that it is much easier to reason about functionally characterized behaviours than about different control regimes.

7.4.3 Subtasks of Gen-design-control

Task `initialise-design-space` initialises the design space by creating its root state, which by convention is associated with an empty design model. The creation of a design state is carried out by means of task `new-design-state`, which takes as input a design-model and produces as output a design state associated with the model. Once created, a state is evaluated to assess its properties - e.g. cost, consistency, etc. The resulting task-subtask hierarchy is described in figure 7.3.

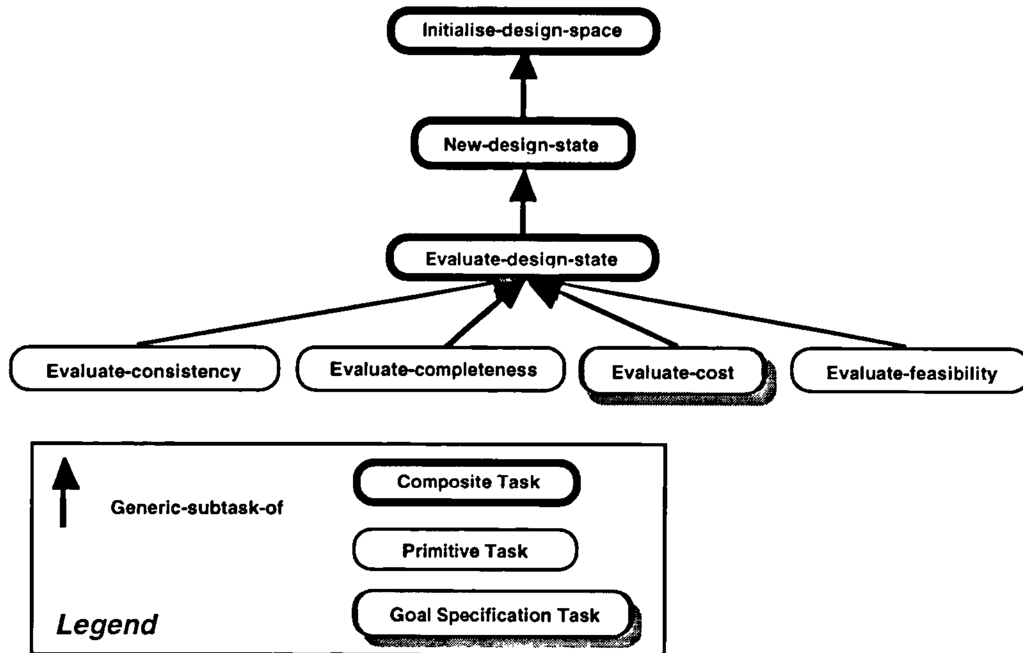


Figure 7.3. Subtasks of task `initialise-design-space`.

The goal of task `reflect-design-state` is to reflect problem solving inferences down to the domain level, by carrying out the necessary *mapping* actions. These mapping actions are not related to the nature of parametric design problem solving and therefore I will not discuss this task here. Examples of mapping knowledge will be given in chapter 9.

Tasks `evaluate-design-state`, `select-design-state`, and `design-from-state` are discussed in the next three sections.

7.4.4 Design state evaluation

The role of task `evaluate-design-state` is to assess a design state by producing the relevant problem solving information. As shown in figure 7.3, there are four main types of knowledge which are inferred when evaluating a design state: *consistency* (whether a state violates some constraints), *cost*, *completeness* (whether any parameter is unbound in the current state), and *feasibility* (i.e. whether a state can lead to a solution). This breakdown is meant to provide maximal coverage - i.e. in my experience these four

classes provide all the knowledge required to make decisions about the current design state. However not all problem solvers would require all four classes of knowledge. For example, problem solvers which are not concerned with cost issues do not need to compute it during the state evaluation process.

In the context of the formal parametric design framework introduced in chapter 6, the assumption here is that a model is defined in terms of parameters and that all the relevant constraints are part of the problem specification. In such a scenario constraint and completeness evaluation are not problematic. Checking for completeness requires finding out whether any parameter is unbound in the current design model, while constraint evaluation is carried out by applying the set of problem constraints to the current model and then returning all those which have been violated. However, it is important to emphasize that this is obviously an idealized scenario. In practice, problem constraints are often acquired incrementally, when design states which had not been anticipated are encountered, or when designers and clients realise that some problem constraints had been left unspecified. The same applies to completeness, as not all parts of a solution are in general known beforehand - in contrast with our scenario - and therefore checking for completeness might be non-trivial. An additional, important element, which is often not realized by researchers working in this area, is that even in relatively structured problems - such as, for instance, the VT elevator design problem - the set of problem constraints is normally incomplete. In the best case it would be complete with respect to the chosen problem solving approach. For example, it is very simple to generate design models which satisfy the VT set of constraints, but which are obviously wrong with respect to basic spatial geometry. The rationale for the missing constraints is that such situations would simply never arise when solving VT by means of a Propose&Revise problem solver.

Cost does not raise important decision-making issues. Having made the assumption that a cost function is provided with the model of the problem, cost evaluation reduces to applying the given cost function to the design model in question.

More complicated is to evaluate the feasibility of a design state. A number of techniques have been developed in the constraint satisfaction literature - e.g. see (Haralick and Elliott, 1980; Gaschnig, 1977) - which provide domain-independent mechanisms to assess whether the current, consistent state lies on a solution path. Although these techniques are of course relevant to parametric design applications, they are not included in the generic model of parametric design problem solving. The main reason for this exclusion is that while the model aims at being 'maximally generic' (i.e. it makes as few assumptions as possible on the available domain knowledge) the techniques developed in the constraint satisfaction literature typically make quite strong assumptions both on the representation of the domain knowledge (e.g. binary instead of n-ary constraints) and on

the problem solving process (e.g. the *backmarking* approach (Gaschnig, 1977) assumes a fixed parameter assignment order, which is not required by the model described here). Therefore these techniques are not part of the model, although of course it is easy to imagine specific refinements of the model where such techniques can be applied[3].

7.4.5 Design state selection

At any stage of the design process, a problem solver (be it human or artificial) knows about a number of design states which are relevant to the problem in hand. Typically, these are the states which have been explored during the current design process, i.e. the states included in the portion of the design space searched so far. However, human designers are of course capable of reusing past designs and the same applies to problem solvers which make use of case-based reasoning techniques when solving design applications (Zdrahal & Motta, 1996). Therefore, in general the current design space includes all the states known to the problem solver, either because they have been explored during the current design process, or because the problem solver has access to other relevant design knowledge (e.g. a case-based library of design states).

Assuming that a rational problem solver would not normally select a design state known to be unfeasible (a *dead end*), it follows that state selection is carried out in terms of the other three main criteria discussed in the previous section: contents of the design model, constraint violations, and cost. This state-centred approach to parametric design problem solving affords both analytical and engineering leverage.

On the engineering side I will show that it is possible to construct alternative (and better behaved) versions of Propose&Revise by enforcing state selection policies which are based on converging criteria. In particular I will compare different refinements of a generic Propose&Revise architecture and show that those employing cost-centred state selection policies behave better than those employing consistency-centred state selection policies.

---

[3] In addition to the aforementioned, reuse-related aspects there is also a more fundamental difference between the knowledge-centred approach I am taking here and the work in the constraint satisfaction literature. My objective here is to identify all the knowledge types and problem solving components that are relevant to parametric design problem solving, so that the resulting generic framework can be applied to (and specialized for) all possible parametric design application domains. In other words my aim is to identify and characterize the slots to be filled by application-specific knowledge. Researchers in the constraint satisfaction literature focus instead on domain-independent search-control mechanisms, which can be directly applied (i.e. with no need for application-specific knowledge) to an application domain.

On the analytical side I will show that a state-centred analysis of problem solving makes it possible to give 'semantics' to apparently idiosyncratic problem solving structures. In particular I will show that the *fix combinations* (Yost and Rothenfluh, 1996) used in the VT application are essentially search-control structures which enforce a cost-conscious search mechanism. Reformulating fix combinations in terms of a state selection policy makes explicit the search control regime adopted by the VT application.

The OCML definition of task `select-design-state`, which is shown below, simply specifies the input (a design space) and the output (a design state) of the task and states that the goal of the task is to select (in practice, return) one of the states included in the input design space.

```
(def-class SELECT-DESIGN-STATE (goal-specification-task) ?task
  ((has-input-role :value has-design-space)
   (has-output-role :value has-design-state)
   (has-goal-expression
    :value (kappa (?task ?s)
                  (and (design-state ?s)
                       (has-design-space ?task ?space)
                       (has-states ?space ?states)
                       (member ?s ?states))))
   (has-design-space :type design-space)
   (has-design-state :type design-state)))
```

The definition given in the next box shows the state selection method specified for `gen-design-psm`.

```
(def-class DEFAULT-STATE-SELECTION (primitive-method)
  ((has-body
    :value (lambda (?psm)
             (in-environment
              ((?cost-algebra . (role-value ?psm has-cost-algebra))
               (?cost-rel . (third ?cost-algebra))
               (?design-space . (role-value ?psm has-design-space)))
              (if (= ?candidates
                     (setofall ?state
                               (and (member ?state
                                            (design-space-states
                                             ?space))
                                    (not (deadend-state ?state))
                                    (not (constraint-violations
                                          ?state ?cs)))))
                (if (= ?maximal-states
                       (setofall
                        ?state
                        (and (member ?state ?candidates)
                             (= ?dm (the ?dm (has-design-model
                                              ?state ?dm)))
                             (= ?l (length ?dm))
                             (not
                              (exists
                               ?state2
                               (and (member ?state2
                                            ?candidates)
                                    (has-design-model ?state2
                                                      ?dm2)
                                    (= ?l2 (length ?dm2))
                                    (> ?l2 ?l)))))))
                  (the ?state
                    (and (member ?state ?maximal-states)
                         (state-cost ?state ?cost)
                         (not (exists
                               ?state2
                               (and (member ?state2
                                            ?maximal-states)
                                    (state-cost ?state2 ?cost2)
                                    (holds ?cost-rel
                                           ?cost2
                                           ?cost))))))))))))
   :own-slots ((tackles-task-type  select-design-state)))
```

Table 7.1 shows the criterion used by the default state selection method: the state selected is one which does not violate any constraint, maximizes the extension of the design model and minimizes the cost. These (sub-)criteria are applied sequentially. Thus, first all consistent states are collected, then all the non-maximal ones are removed, and then the cheapest one (or one of the cheapest, in case one or more states score equally on these three criteria) is returned.

Violated Constraints	Design Model	Cost
No	Max	Min

Table 7.1. Criteria for state selection in default state selection method

This selection criterion is the one typically used by problem solvers which are not concerned with cost issues (as for instance most constraint satisfaction engines) and which deal with inconsistencies by backtracking to an earlier state. These include both methods which make use of simple control regimes, such as depth-first search with chronological backtracking (Runkel et al., 1996), as well as more sophisticated regimes based on techniques such as *backjumping* (Gaschnig, 1978; Dechter, 1988). Given the framework discussed here, the differences between these methods - e.g. backjumping vs. depth-first search with chronological backtracking - are therefore explained in terms of different notions of feasibility, rather than in terms of different state selection policies. Clever backtracking techniques, such as backjumping, can propagate unfeasibility 'backwards', to nodes which precede an inconsistent state. Hence, even though different methods may use the same state selection policy, they can still exhibit different search behaviours.

7.4.6 State-based design process

The top-level control regime specified by task `gen-design-control` is method-generic; method-specific control is defined by providing the appropriate control method associated with task `design-from-state`. Such a method defines the main design strategy of a parametric design problem solver. More precisely, it specifies its strategy for navigating the design space.

The method below describes a simple control regime, which does nothing when applied to inconsistent or unfeasible states, and calls task `generate-state-successor` in an `:extend` context, when applied to incomplete states. This control regime is the one used by methods which do not use special search-control knowledge to deal with inconsistent states and do not attempt to improve solution states. One example is the method used by the DIDS researchers to solve the Sisyphus tasks (Balkany et al., 1994; Runkel et al., 1996)[4].

---

[4] To be precise, Runkel et al. (1996) actually discuss two problem solving methods for the VT task, one which uses fixes and one which does not. Only the latter makes use of the `extend-incomplete-state` control regime.

```
(def-class EXTEND-INCOMPLETE-STATE (decomposition-method)
  ((has-input-role :value has-design-state)
   (has-output-role :value generates-design-state)
   (has-design-state :type design-state)
   (generates-design-state :type design-state)
   (has-goal-expression
    :value (kappa (?task ?s)
                  (design-model-extends
                   (the ?dm (has-design-model ?s ?dm))
                   (the ?dm (has-design-model
                             (role-value
                              ?task has-design-state)
                             ?dm)))))
   (has-body
    :value
    (lambda (?psm)
      (in-environment
       ((?state . (role-value ?psm has-design-state))
        (?design-model . (the ?dm (has-design-model
                                   ?state ?dm)))
        (?constraints . (role-value ?psm has-constraints))
        (?parameters . (role-value ?psm has-parameters)))
       (if (deadend-state ?state)
         :nothing
         (if (constraint-violations ?state ?constraints)
           (tell (deadend-state ?state))
           (if (state-complete ?state ?parameters)
             (tell (solution-state ?state))
             (achieve-generic-subtask
              ?psm
              generate-state-successor
              has-design-state ?state
              has-design-context :extend))))))))

  :own-slots ((tackles-task-type  design-from-state)
              (has-generic-subtasks generate-state-successor)))
```

The reason for proposing the above method as the default way of carrying out task `design-from-state` is that this method employs minimal commitments. No knowledge roles in addition to those specified in the parametric design task ontology are used here, no actions are performed in the face of inconsistent states, and cost factors do not affect problem solving. In the next chapter I will show that it is easy to model the problem solving approach of more knowledge-intensive problem solving methods, by adding more conditions and ontological commitments to this basic control regime.

7.4.7 State generation and backtracking

Task `generate-state-successor` is a generic control task which 'mediates' between `design-from-state` and `design-from-context`. Its role is essentially to abstract a generic control pattern, which is common to all states and contexts. Specifically, this task collects the design foci relevant to the current state and context and then calls task `design-from-context` to select the appropriate focus and operator.

```
Generic Task Generate-state-successor
  Inputs:    Design-state, Design-context
  Output:    Successor-state
  Control:   Foci, Record
  Goal:      "Output is a state"
  Subtasks:  Resume-state, Design-from-context , Collect-state-foci,
             New-search-control-record
  Body:      If "Search control record exists for Design-state"
                then
                  Resume-state (Design-state, Design-context) -> Successor-state
                  If "Resume-state succeeds"
                     then Return () -> Successor-state
                     else Design-from-context (Design-state, Design-context)
                else
                  Do
                     Collect-state-foci (Design-state, Design-context) -> Foci
                     New-search-control-record (Design-state, Foci) -> Record
                     Design-from-context (Design-state, Design-context)
```

The above definition - formalized in appendix 4 - is complicated by the fact that in general task `generate-state-successor` can be given as input a state which has already been (partially) explored - i.e. we might be backtracking to this state. In such a case, the control body of the task invokes task `resume-state`.

Whether a not a state has already been explored is determined by checking whether a *search control record* exists for the state in question. A search control record is a structure associated with a design state, which records dynamic, state-related problem solving information. This includes the current design focus and those operators and foci, which are applicable to a design state, but have not yet been used to generate a successor state.

The rationale for introducing search control records as a separate construct in the model, rather than adding this information directly to the specification of a design state, is to ensure that a design state is defined 'functionally', i.e. independently of the current (control) state of the problem solver. Moreover, while different problem solving methods can add more information to this definition of a search control record, the given definition of design state remains independent of a particular problem solving method.

The box below shows the definition of a search control record and a function which retrieves the record associated with a state.

```
(def-class SEARCH-CONTROL-RECORD ()
  "This structure records the control information associated
   with a state.  It is necessary to be able to support
   generic control regimes"
  ((has-design-state :type design-state :cardinality 1)
   (has-design-focus :type design-focus  :cardinality 1)
   (has-design-operators :type list  :cardinality 1)
   (has-design-foci :type list :cardinality 1)))

(def-function THE-STATE-SEARCH-CONTROL-RECORD (?state)
  :body (the ?record (and (search-control-record ?record)
                          (has-design-state ?record ?state))))
```

Finally, the next box shows the specification of task `collect-state-foci` and that of a method, `collect-computable-parameters`, which carries out the task in a design extension context. The method uses the notion of parameter dependency, introduced earlier, to retrieve all unbound, computable parameters.

```
(def-class collect-state-foci (goal-specification-task) ?task
  ((has-input-role :value  has-design-context
                   :value  has-design-state)
   (has-output-role :value  has-design-foci)
   (has-design-foci :type list)
   (has-design-state :type design-state)
   (has-design-context :type design-context)))

(def-class collect-computable-parameters (primitive-method)
  ((has-body
    :value (lambda (?psm)
             (all-computable-parameters
              (role-value ?psm has-parameters)
              (the ?dm (has-design-model
                        (role-value ?psm has-design-state)
                        ?dm))))))
  :own-slots ((tackles-task-type collect-state-foci)
              (applicability-condition
               (kappa (?task)
                      (= (role-value ?task
                                     'has-design-context)
                         :extend)))))
```

7.4.8 Context-centred design

Task `design-from-context` is invoked in a problem solving scenario in which a state has been evaluated and selected, and a context abstracted. Given this input scenario, the task provides a generic control regime - parametrized in terms of the abstract notions of design focus and design context - which abstracts from the behaviours employed by different problem solvers in apparently different, but essentially isomorphic design scenarios. For example, in a Propose&Revise problem solver, the context could be 'revision' (and the focus a specific constraint violation) or it could be 'extension' (and the focus a specific design parameter). Thus, given this generic control regime, differences

between problem solving methods can be achieved either by instantiating the generic notions of 'context' and 'focus' by means of specific design structures, or by associating different methods to some of the subtasks of `design-from-context` - e.g. by exploiting different focus selection strategies.

The task-subtask decomposition introduced by task `design-from-context` and the data flow between the task's subtasks are shown in figure 7.4 and 7.5[5].

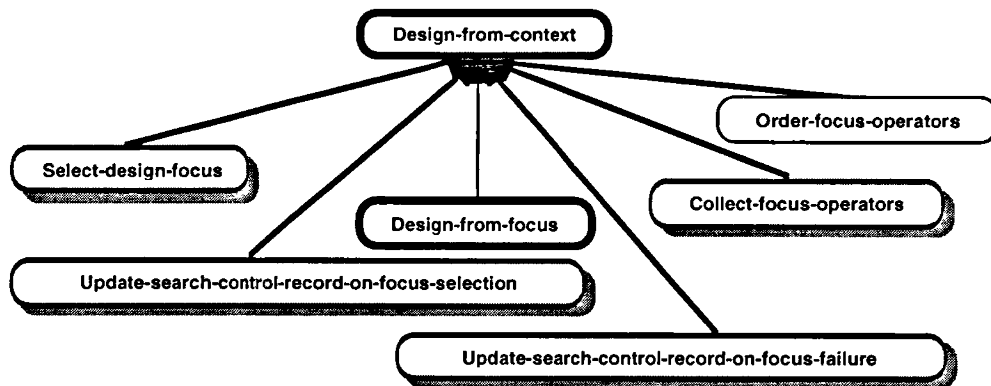


Figure 7.4. Subtasks of task `design-from-context`.

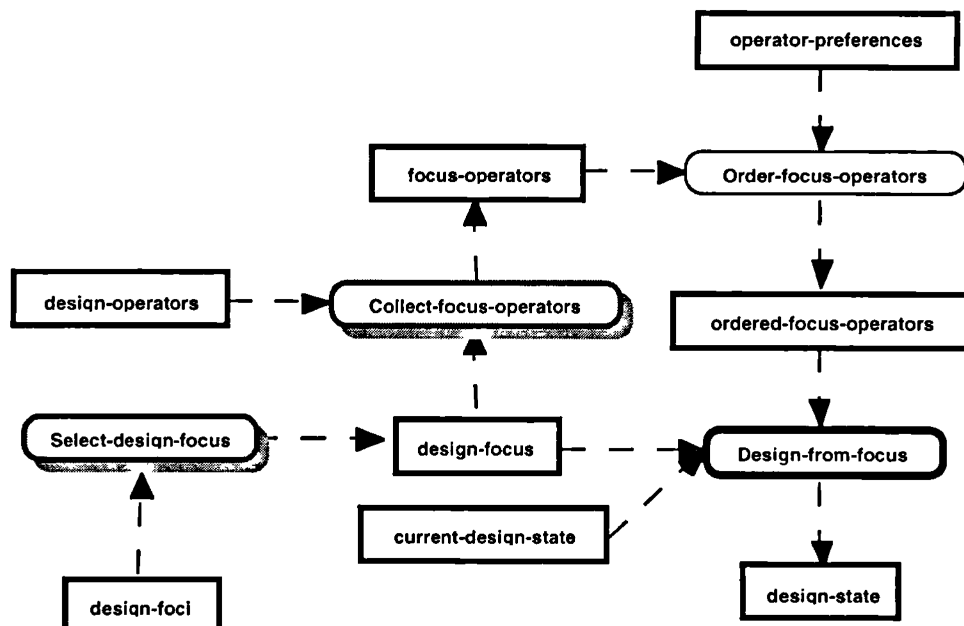


Figure 7.5. Data flow relations between subtasks of `design-from-context`

---

[5] The two subtasks which deal with updating the search control record associated with the current design state are not included in figure 7.5.

The OCML specification of task `design-from-context` is quite complicated and is given in appendix 4. A simplified[6], informal description of the task is shown in the box below.

```
Generic Task Design-from-context
  Inputs:     Design-state, Design-context, Design-foci
  Output:     Successor
  Control:    Focus, Ops1, Ops2, Operator-preferences
  Goal:       "Output is a state"
  Subtasks:   Select-design-focus, Collect-focus-operators, Sort-design-operators,
              Design-from-focus
  Body:       Repeat
                 Select-design-focus (Design-foci) -> Focus
                 If "Select-design-focus fails"
                    then Return () -> Fail
                    else
                      Do
                         Collect-focus-operators (Focus) -> Ops1
                         Sort-design-operators (Ops1, Operator-preferences) -> Ops2
                         Design-from-focus (Design-state, Focus, Ops2) -> Successor
              until "Design-from-focus succeeds"
```

As shown by the above definition, the body of task `design-from-context` consists of the following actions: i) selecting a design focus; ii) collecting all the operators relevant to the selected focus; iii) ordering them according to application-specific meta-knowledge and iv) invoking task `design-from-focus`. These actions are carried out iteratively until either i) there are no more possible design foci to try or ii) a useful result (i.e. a new design state) is returned by task `design-from-focus`.

7.4.9 Design focus selection

Selecting a design focus is the most important task in parametric design problem solving. As shown by much theoretical and empirical research on constraint satisfaction and design problem solving, selecting the right focus is crucial for efficient problem solving when using complete search methods, and for efficiency and competence when using incomplete methods. For this reason several domain-independent heuristics for variable ordering or selection have been developed in the constraint satisfaction literature (Dechter and Meiri, 1989; Sadeh and Fox, 1996; Keng and Yun, 1989; Minton et al., 1992),

---

[6] In particular, the complete definition takes care of updating the search control record associated with the input design state and also checks that subtask `collect-focus-operators` actually returns some operators.

which have been shown to perform orders of magnitude better than simple depth-first search with chronological backtracking.

These techniques can be easily integrated with the framework proposed here, to provide domain-independent focus selection techniques in a design extension context. Moreover, it is often the case that application-specific knowledge is available, which can be used for focus selection. For instance, the VT specification (Yost and Rothenfluh, 1996) states: "If more than one constraint can be processed at the same time, pick one arbitrarily. One exception to this is that if both **MACHINE GROOVE PRESSURE** and **HOIST CABLE TRACTION RATIO** constraints are violated at the same time, try to fix the **MACHINE GROOVE PRESSURE** violation first.". In terms of the framework proposed here, this statement can be interpreted as stating that if the context is 'revise' and **MACHINE GROOVE PRESSURE** and **HOIST CABLE TRACTION RATIO** are two possible design foci, then the former should be selected, rather than the latter.

In what follows, I will only discuss focus selection techniques which are relevant to a design extension context. Focus selection techniques for alternative contexts will be introduced in chapters 8 and 9.

7.4.9.1. Variable ordering heuristic and focus selection in design extension context

As I said above, much work on variable ordering which exists in the constraint satisfaction literature is relevant here. However, it is also important to emphasize that constraint satisfaction methods are almost often based on frameworks which are not as 'rich' as the one being developed here. In particular they typically consider binary constraint networks and do not tackle cost-related aspects. These restrictions imply that it is difficult to express parameter selection criteria purely in terms of constraint-related heuristics. I'll clarify this point with an example taken from the Sisyphus-I office allocation problem.

The problem specification given in (Linster, 1994) shows the following sequence of office allocation steps: head-of-group, secretaries, manager, heads of projects, researchers. It is easy to see that this sequence of assignments can be reproduced almost completely, simply by using the *dynamic search rearrangement* (DSR) heuristic (Dechter and Meiri, 1989) at each stage of the parameter selection process. In simpler terms, what happens here is that the Sisyphus-I domain expert (Siggi) always focuses on those YQT members who have the smallest number of possible rooms available. However, the DSR heuristic does not completely account for Siggi's strategy. In particular it does not have enough discriminatory power to distinguish between the manager and the head of the projects. The problem here is that from a constraint satisfaction point of view these two classes of personnel offer the same degrees of freedom - they both require single rooms. However, when deciding whom to allocate next, Siggi also applies an element of

seniority: if two elements are equally easy (or difficult) to allocate, then the most important one is allocated first. In particular, the manager is allocated before the heads of project.

This seniority element can be captured in the task specification, by representing the appropriate preferences and cost function, and in the method ontology, by specifying appropriate preferences about focus selection. It cannot be represented purely in terms of constraints or emulated by means of heuristic selection techniques which do not reason about cost aspects.

Focus selection preferences can be expressed by means of relation `design-focus-order`, whose OCML definition is shown below.

```
(def-relation DESIGN-FOCUS-ORDER (?x ?c)
  "This relation can be used to specify an application specific
   ordering of design foci"
  :constraint (and (design-focus ?x)
                   (design-focus ?c)
                   (not (= ?x ?c))))

(tell (DEFINES-PARTIAL-ORDER design-focus-order))
```

7.4.9.2. *Default parameter selection strategy for design extension context*

The default parameter selection strategy provided with the library (for a design extension context) combines the DSR strategy with focus selection preference knowledge. Its OCML definition is given below.

```
(def-class DEFAULT-PARAMETER-SELECTION (primitive-method)
  ((has-input-role
    :value has-design-focus-order-relation
    :value has-possible-values-relation)
   (has-design-focus-order-relation
    :default-value design-focus-order)
   (has-possible-values-relation
    :default-value possible-value)
   (has-body
    :value (lambda (?psm)
             (if  (= ?foci (role-value ?psm has-design-foci))
               (select-most-preferred-focus
                (collect-most-restricted-parameters
                 ?foci
                 (role-value ?psm
                             has-possible-values-relation))
                (role-value ?psm
                            has-design-focus-order-relation))))))
  :own-slots ((tackles-task-type select-design-focus)
              (applicability-condition
               (kappa (?task)
                      (every (the ?foci
                                  (has-design-foci
                                   ?task ?foci))
                             parameter)))))
```

The body of method `default-parameter-selection` first calls function `collect-most-restricted-parameters` on the set of possible foci, to collect all the parameters with the most restricted range of values in the current problem solving scenario. Then, it uses focus selection preference knowledge to discriminate between eventual ties. The definition of function `collect-most-restricted-parameters` is as follows.

```
(def-function COLLECT-MOST-RESTRICTED-PARAMETERS (?l ?rel)
  :body (in-environment
         ((?pairs . (sort (map '(lambda (?p)
                                   (list-of
                                    ?p (setofall
                                        ?v (holds ?rel ?p ?v))))
                                ?l)
                          '(kappa (?x ?y)
                             (< (length (second ?x))
                                (length (second ?y)))))))
         (map first (filter
                     ?pairs
                     '(kappa (?pair)
                        (= (second ?pair)
                           (second (first ?pairs))))))))
```

The definition above assumes that the relation `possible-value` is used to model the relation between a parameter, say ?p, and a value, say ?v, which can be assigned to ?p in the current problem solving context. Thus, the above function i) constructs a list by associating each parameter in the current pool with its effective value range, ii) sorts this list in terms of range size and then iii) returns all the parameters which are associated with the smallest number of possible values.

Finally, the definition below shows the OCML representation of function `select-most-preferred-focus`. This function takes as input a list of design foci and a focus preference relation and returns the most preferred design focus in the input list.

```
(def-function SELECT-MOST-PREFERRED-FOCUS (?l ?rel)
  "The most preferred focus is one which is in
   the input list (i.e. which is a possibility in the
   current scenario) such that there is no other focus
   which is preferred to it"
  :body (the ?focus
          (and (member ?focus ?l)
               (not (exists ?focus2
                      (and (member ?focus2 ?l)
                           (<> ?focus2 ?focus)
                           (holds ?rel ?focus2 ?focus)))))))
```

As will be demonstrated by the application examples discussed in chapter 9, this combination of a generic DSR strategy with additional, typically cost-related, preference knowledge for focus selection provides a very powerful mechanism to improve the efficiency of a problem solver and to facilitate the generation of optimal solutions.

However, it is important to emphasize that such local parameter selection knowledge is subject to *horizon effects* (Stefik, 1995). Problem solving approaches which are able to reason 'globally' about cost-related aspects, such as global hill climbing or A*, are needed if such horizon effects are to be avoided.

7.4.10 Collecting and prioritizing operators

7.4.10.1. Task sort-design-operators

Once a design focus has been selected, the relevant operators are collected and sorted. The latter task is very simple. We can use the applicable operator preference knowledge to decide which is the most 'preferred' operator. As discussed in section 7.3, this knowledge operationalizes preference knowledge acquired as part of the application specification. The definition of task `sort-design-operators` is as follows.

```
(def-class sort-design-operators (primitive-task) ?task
  ((has-input-role :value has-design-operators
                   :value has-operator-order-relation)
   (has-design-operators :type list)
   (has-operator-order-relation
    :default-value design-operator-order)
   (has-body
    :value (lambda (?task)
             (sort (role-value
                    ?task has-design-operators)
                   (role-value ?task
                               has-operator-order-relation))))))
```

Collecting design operators is also conceptually simple: we want all the design operators which are relevant to the current focus and context. For example, if the context is `:revise` and the focus a constraint violation, say `?c`, then we want to retrieve all operators which can be used to fix `?c`. If the context is `:extend`, then the current focus is a parameter which is unbound in the current design model, say `?p`. In this case we need operators which can generate a value for `?p`.

As in the earlier section on focus selection, I will confine the discussion about operator collection to the design extension context. Operator collection in contexts other than `:extend` will be discussed in chapters 8 and 9.

7.4.10.2. Design Extension Operators

The rationale for applying an operator in a design extension context is of course to assign a value to a currently unbound parameter. Given this context-specific rationale, it is clear that the definition of design operator given in section 7.2.2 is much too general. A design extension operator does not generate an arbitrary output model from an arbitrary input model. Given an input design model and a parameter, say `?p`, it generates an output model which differs from the input model only with respect to `?p`. Thus, I can specialize

the notion of design operator for a design extension context, by defining a class of *design extension operators.*

```
(def-class DESIGN-EXTENSION-OPERATOR (design-operator)

  "The body of a design extension operator is a unary function
   which takes as argument an unbound parameter, ?p, and the
   current design model, ?dm, and produces as a result a
   new value, ?z, which is taken to specify the value of
   ?p in a design model, which extends ?dm with respect to ?p.
   The values of all the other design parameters should
   not be affected by the application of the operator."

  ((applicable-to-parameters
    :default-value '(setofall ?x (parameter ?x))
    :type function-expression
    :documentation "An expression which returns the set
                    of parameters whose value can be computed
                    by means of this operator")
   (body :type design-extension-operator-body)))
```

A design extension operator is an operator, whose body takes a parameter and a design model as input and returns a new value for the parameter. The body of a design extension operator is formally defined as follows:

```
(def-class DESIGN-EXTENSION-OPERATOR-BODY (lambda-expression) ?x
  "A basic design extension operator body is a unary function which
   takes an unbound parameter, say ?p and produces a result, ?z,
   which belongs to the value range of ?p.  ?z is taken as the new
   value of ?p in the successor design state"
  :no-op (:constraint (and (nth-domain ?x 1 parameter)
                           (nth-domain ?x 2 ?dm)
                           (=> (= ?z (call ?x ?p))
                               (and (has-value-range ?p ?range)
                                    (member ?z ?range))))))
```

The slot `applicable-to-parameters` in the definition of class `design-extension-operator` makes it possible to specify the range of parameters to which a design extension operator can be applied. The value for this slot defaults to all currently defined parameters.

Having defined the class of design extension operators it is now quite simple to define a default operator collection method for a design extension context. As shown by the OCML definition below, the method collects all the design extension operators which are applicable to the current parameter.

```
(def-class DEFAULT-OPERATOR-COLLECTION (primitive-method) ?psm
  ((has-body
    :value (lambda (?psm)
             (setofall ?op
                       (and (design-operator
                             ?op
                             applicable-to-parameters ?l)
                            (member (role-value
                                     ?psm 'has-design-focus)
                                    (eval ?l)))))))
   :own-slots ((tackles-task-type collect-focus-operators)
              (applicability-condition
               (kappa
                (?task)
                (and (= :extend
                        (role-value
                         ?task 'has-design-context))
                     (parameter
                      (role-value
                       ?task 'has-design-focus)))))))
```

7.4.11 Focus-centred design

Task `design-from-focus` is similar to task `design-from-context` in the sense that, like the latter, it provides a generic control mechanism which abstracts from superficially different, but essentially isomorphic problem solving mechanisms. In particular, the purpose of this task is to define the abstract select-design-operator/apply-design-operator control regime, which is independent of the specific nature of the current focus.

As shown below, task `design-from-focus` is modelled as a composite task, whose subtasks are `try-design-operator` and `select-design-operator`. The body of the task is specified in terms of a `repeat` statement, which selects and applies a design operator, until a result other than `:nothing` is returned. The body of the task also takes care of updating the search control record associated with the current state, so that it is possible to resume this state and try a different operator application.

```
(def-class DESIGN-FROM-FOCUS (composite-task)
  ((has-input-role :value has-design-state)
   (has-output-role :value has-output-design-state)
   (has-control-role :value has-design-model
                     :value has-design-operator)
   (has-design-state :type design-state)
   (has-output-design-state :type design-state)
   (has-body
    :value
    (lambda (?task)
      (REPEAT
       (in-environment
        ((?state . (role-value ?task has-design-state))
         (?record . (the-state-search-control-record
                     ?state))
         (?focus . (the-slot-value
                    ?record 'has-design-focus))
         (?ops .  (the-slot-value
                   ?record 'has-design-operators))
         (?sub . (instantiate-generic-subtask
                  ?task select-design-operator
                  has-design-focus ?focus
                  has-design-operators ?ops))
         (?op . (solve-task ?sub)))
        (if (achieved ?sub ?op)
          (DO
            (set-slot-value
             ?record
             has-design-operators
             (remove ?op ?ops))
            (in-environment
             ((?sub2 . (instantiate-generic-subtask
                        ?task try-design-operator
                        has-design-operator ?op
                        has-design-focus ?focus
                        has-design-model (the-slot-value
                                          ?state
                                          'has-design-model)))
              (?result . (solve-task ?sub2)))
             (if (achieved ?sub2 ?result)
               (RETURN ?result))))
         (RETURN :nothing)))))))
  :own-slots ((has-generic-subtasks '(select-design-operator
                                      try-design-operator))))
```

7.4.12 Design operator selection

Given that the operators applicable to the current focus have been ordered in terms of the operator preference knowledge, design operator selection normally consists of picking the next operator on the ordered list. Therefore the default operator selection method is very simple - see definition below.

```
(def-class SELECT-DESIGN-OPERATOR (goal-specification-task)
  ((has-input-role :value has-design-operators
                   :value has-design-focus)
   (has-output-role :value has-selected-operator)
   (has-design-operators :type list)
   (has-selected-operator :type design-operator)
   (has-design-focus :type design-focus)))

(def-class DEFAULT-OPERATOR-SELECTION (primitive-method) ?psm
  ((has-body
    :value (lambda (?psm)
             (first (role-value ?psm
                                'has-design-operators)))))
   :own-slots ((tackles-task-type  select-design-operator)))
```

Because parametric design is often concerned with finding optimal or sub-optimal solutions, preference knowledge is used to drive the selection of a design operator. This type of knowledge is often called *local* preference knowledge (Poeck and Puppe, 1992) and is defined as knowledge which can be used to make locally optimal decisions. The use of local preference knowledge leads to greedy algorithms, such as hill-climbing, which can get stuck in local maxima. In the next section I will discuss a problem solving method called *Propose&Improve*, which tries to avoid getting stuck in local maxima by applying a 'global hill-climbing' kind of approach.

Finally, it is important to highlight that, in contrast with the focus selection case, the techniques developed in the constraint satisfaction literature, i.e. *value ordering heuristics* (Dechter and Pearl, 1988), are of only limited use here. These techniques try to find at each stage of the variable assignment process the least constraining values - i.e. the values which are less likely to cause backtracking at a later stage of the constraint solving process. Unfortunately these heuristics are only practical in binary constraint networks with fixed variable ordering. As discussed by Sadeh and Fox (1996), these heuristics do not perform very well in the presence of dynamic variable ordering, which is the scenario assumed here. Moreover, Sadeh and Fox also demonstrate that these heuristics do badly in tightly connected constraint networks.

7.4.13 Applying a design operator

The last task which is left to discuss is the application of a design operator. This is a simple task which does not require any decision making. The box below shows how this task is carried out in the case of design extension operators. Basically, the chosen operator is applied to the current focus (i.e. parameter) and design model. If the value is something other than `:nothing`, then a new state is created and returned.

```
(def-class TRY-DESIGN-OPERATOR (goal-specification-task) ?task
  ((has-input-role :value has-design-operator
                   :value has-design-focus
                   :value has-design-model)
   (has-output-role :value generates-design-state)
   (has-design-focus :type design-focus)
   (has-design-operator :type design-operator)
   (has-design-model :type design-model)
   (generates-design-state :type design-state)
   (has-goal-expression
    :value (kappa (?task ?s)
                  (and (design-state ?s)
                       (generates-design-state ?task ?s))))))
```

```
(def-class TRY-DESIGN-EXTENSION-OPERATOR (primitive-method)
  ((has-body
    :value
    (lambda (?psm)
      (in-environment
       ((?dm . (role-value ?psm 'has-design-model))
        (?focus . (role-value ?psm 'has-design-focus))
        (?value . (apply-design-extension-operator
                   ?focus ?dm (role-value ?psm
                                          'has-design-operator))))
       (if (not (= ?value :nothing))
         (achieve-generic-subtask
          ?psm new-design-state
          has-design-model (cons
                            (cons ?focus ?value)
                            ?dm)))))))
  :own-slots ((tackles-task-type try-design-operator)
              (applicability-condition
               (kappa
                (?task)
                (design-extension-operator
                 (role-value
                  ?task has-design-operator))))))
```

The method `try-design-extension-operator` makes use of the function `apply-design-extension-operator`, which is defined as follows.

```
(def-function apply-design-extension-operator (?param ?dm ?op)
  :constraint (and (parameter ?param)
                   (design-model ?dm)
                   (design-extension-operator ?op))
  :body (call (the ?body (has-body ?op ?body)) ?param ?dm))
```

With this definition I have concluded the description of a generic model for parametric design problem solving. This model identifies the main subtasks and knowledge roles which characterize parametric design problem solving and proposes default methods for carrying out these tasks. In the next section I will summarize the main aspects of the proposed model and I will then conclude the chapter by comparing it to alternative proposals.

7.4.14 Main aspects of the generic model for parametric design problem solving

In this section I will highlight the main aspects of the model discussed in this chapter. These are: methodological framework; knowledge types and generic tasks.

7.4.14.1. Methodological framework.

The model presented in this chapter builds on the task ontology discussed in chapter 6 and on the view of problem solving as search discussed in chapter 1. Thus, it instantiates the generic notions of search space, search state and state transition in the context of the parametric design ontology, generating the concepts of design space, design state, and design operator. It also introduces the notions of design context and design focus to decompose the problem of selecting a design operator into a number of intermediate subproblems. In addition, I have also discussed how task knowledge is operationalized during the problem solving process, by illustrating the mapping between task and method concepts.

7.4.14.2. Knowledge Roles

Table 7.2 shows the main classes of problem solving knowledge associated with the generic parametric design model. The classes shown in bold indicate the main domain roles associated with the framework. These roles can be filled by means of the appropriate application-specific knowledge, much as in the role-limiting method approach. However, the framework has been designed so that only design operators and mapping knowledge are required in order to instantiate the problem solver in a domain. All the other types of domain knowledge, which are shown in bold italics, denote optional roles, which are useful to improve the efficiency of the problem solving process, but are not essential. Finally, the roles shown in plain text indicate intermediate knowledge structures generated during problem solving.

Knowledge Classes	Description
Design Operator	Knowledge for modifying design models
Design Extension Operator	Knowledge for extending design models
Design Space	The space of all design models considered by a problem solver
Design State	An element of the design space
Success State	A state which denotes a solution model
Deadend State	A state which does not lead to a solution state
Incomplete State	A state which is associated with an incomplete design model
Inconsistent State	A state in which some constraints are violated.
Search Control Record	Problem solving control knowledge associated with a design state
Mapping Knowledge	Knowledge which relates the problem solving notion of design model to domain-specific structures (e.g. to room allocations)
Design Context	Abstract label associated with a design state which can be used to decide the next problem solving step.
Design Focus	Abstract notion which denotes the main design element driving the selection of a design operator.
Focus Selection Knowledge	Knowledge used to select a design focus
Operator Selection Knowledge	Knowledge used to select a design operator
Available Parameter Values	Knowledge which supports the generation of the values available for an unbound design parameter.

Table 7.2. Problem solving knowledge for parametric design

7.4.14.3. Generic tasks

Figure 7.6. below shows the overall task-subtask structure of the generic design model. This consists of 23 tasks, which divide into 11 goal specification tasks, 5 primitive tasks, and 7 composite tasks. For each goal specification task I discussed one or more default methods, thus providing a complete problem solver, `gen-design-psm`. This problem

solver implements a simple depth-first control with chronological backtracking and uses both application-specific knowledge and domain-independent heuristics to guide focus (i.e. parameter) and operator selection.

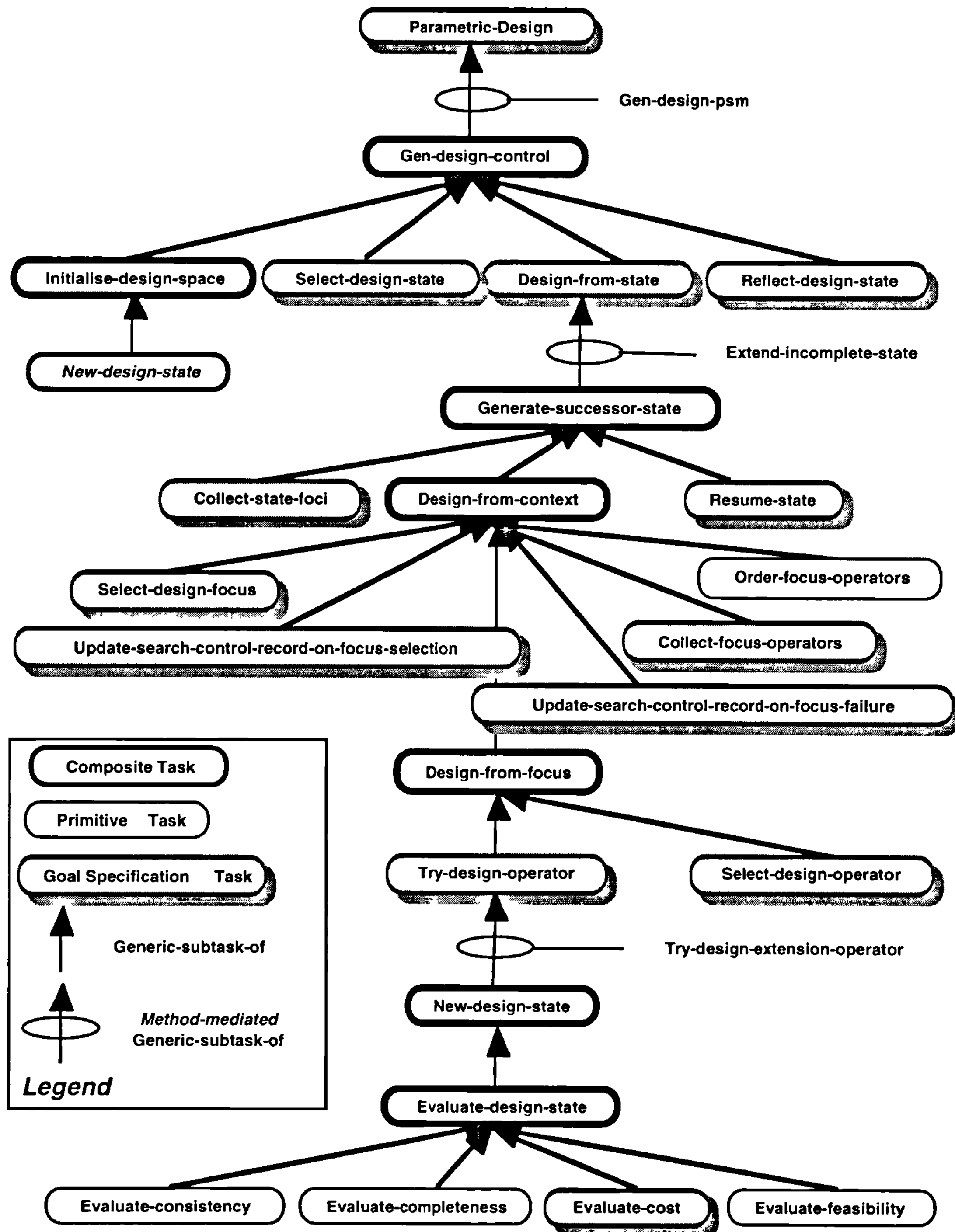


Figure 7.6. Overall task-method hierarchy in generic design model.

Ultimately, reusable components can be validated in one of two ways: either by showing that the proposed components subsume and therefore account for existing work in the literature (i.e. the new components provide analytical leverage), or by showing that it is possible to effectively reuse these components in different applications (engineering

leverage). In the next two chapters I will provide empirical evidence that the components developed in this work fulfil both the analytical and engineering roles, by showing that i) it is possible to model a number of existing problem solving methods as refinements of the proposed generic model; ii) it is possible to characterize task models of well known applications in terms of the parametric design ontology given in chapter 5 and iii) that the problem solving components developed in this work can be effectively reused to build parametric design applications.

7.5 COMPARISON WITH OTHER APPROACHES

7.5.1 Comparison with DIDS toolkit

The task-centred aspects of the DIDS framework have already been discussed in section 6.5.2, where I compared the parametric design task ontology presented in the previous chapter with the framework underlying the DIDS approach. In this section I will look at the method-related aspects of the DIDS model and, in particular, I will compare the problem solving model presented in this chapter with the library of DIDS mechanisms and the overall 'design philosophy' characterizing the DIDS model.

The work presented here has a number of similarities with the DIDS approach. Both frameworks are based on a view of design as search through a design space, and both the DIDS researchers and I share the goal of generating a set of reusable components for design applications. The set of knowledge types provided in the DIDS library is also consistent with the one discussed here. It includes *ordering knowledge* which is a subset of what I call focus selection knowledge, and preference knowledge which corresponds to operator selection knowledge. In addition, the DIDS library includes *connection* and *arrangement knowledge*, which is not included in my model.

A difference between the model presented here and the DIDS library of mechanisms is that the latter aims to support full configuration design problem solving, while here I only focus on parametric design problems. On the other hand I believe that both the approach and the model presented here provide a number of advantages compared with the DIDS approach and library of mechanisms. Specifically, the differences are as follows.

- The generic design model presented here is meant i) to subsume specific problem solving methods for parametric design, and ii) to provide the basis for implementing a shell supporting rapid development of applications through reuse. The same claim cannot be made for the DIDS framework, which plays essentially an engineering role. It is easy to see the difference by looking at the integration of fixes in the DIDS framework and in the one presented here. Fixes can be integrated in DIDS by adding a suitable mechanism - add-part-using-fixes (Runkel et al., 1996). The problem with this solution is that the proposed mechanism

carries out several functions: it adds a value to a parameter and checks whether a constraint is violated; if this is the case, then it applies fixes to remove the inconsistency. This approach of course raises questions about the granularity of the mechanisms, and about the design principles underlying the DIDS library. More importantly, it makes it harder - for example - to experiment with different variants of propose and revise. For instance, we might want to fix the constraint violations only after the model has been completed - CMR architecture (Motta et al., 1996). In order to do this we would need to change the DIDS mechanism, even if it is only a change to the control regime: the basic fix application procedure remains the same. In contrast with the DIDS approach, my framework makes it possible to characterize a fix as a specialization of a design operator, which is invoked when designing from an inconsistent state. Thus, adding fixes to the model only requires specializing the class of design operators and specifying the conditions under which fixes should be applied. As a result, it becomes easier to experiment with alternative control regimes. Moreover, the fix mechanism is provided a clear computational semantics in terms of the search paradigm.

- The framework proposed here is 'rich' in the sense that it is formulated through a process of ontological engineering. This 'richness' affords several advantages: the framework can support verification of application knowledge and provides a formal basis for reuse.

- A third important difference between my approach and the DIDS one is that we seem to subscribe to alternative views of what constitutes reuse. For the DIDS researchers reuse consists of providing a very general problem solving model. However, the price for such generality is inefficiency - see solution #1 to VT problem (Runkel et al., 1996). In contrast with the DIDS approach, I believe that supporting reuse consists of providing a rich set of reusable mechanisms, which can be used in different problem solving scenarios to develop efficient problem solvers. This set is not meant to be minimal. On the contrary, it is meant to be maximal and provide adequate leverage for developing efficient reasoners. For this reason my framework proposes a much more fine-grained breakdown of parametric design tasks than afforded by the DIDS tools. Moreover, as I will discuss in the next chapter, I have expanded the set of reusable problem solving components presented here by generating additional components as required by more specific problem solving methods.

- Finally, although the DIDS researchers claim to have based their framework on a model of design as search, it is not clear what are the principles which underline the DIDS approach to identify reusable mechanisms. As discussed above in the case of add-part-with-fixes, some of these mechanisms seem to be quite coarse-

grained. Moreover, while the DIDS researchers give the impression that mechanisms are essentially functionally defined components, the DIDS library includes mechanisms such as "chronological-backtrack", which provide simple control. In contrast with DIDS, here I have followed a top-down approach, which started with the task ontology and the search paradigm, and was then refined by introducing the notions of design context and design focus.

7.5.2 Comparison with Chandrasekaran

In a very influential paper, Chandrasekaran (1990) carried out a task analysis of design problem solving in which he discusses a generic Propose-Verify-Critique-Modify class of methods (PVCM) and constructs a task-method hierarchy, which associates a number of methods to each subtask of the generic PVCM class. For instance he discusses simulation methods for design verification and the application of case-based and constraint satisfaction techniques to design extension. For each method, Chandrasekaran briefly describes its knowledge requirements and computational properties.

The main difference between Chandrasekaran's analysis and mine is of course one of granularity. His analysis is carried out at a much higher level of abstraction than the one adopted here. While my goal is to provide a fine-grained, formally specified reusable problem solving model for parametric design, Chandrasekaran's analysis is concerned with characterizing the topmost level of the task-method structure of design problem solving. Nevertheless, the basic philosophy which underlines Chandrasekaran's work also applies to the work presented here. Specifically, this can be seen as an attempt to instantiate some of Chandrasekaran's ideas - e.g., the use of task analysis to study problem solving, the knowledge-intensive nature of design problem solving - in a precisely defined subset of the space of design applications.

7.5.3 Comparison with constraint satisfaction approaches

Design problems in general and parametric design problem in particular can be viewed as constraint satisfaction problems and solved by means of constraint satisfaction techniques (Flemming et al., 1992).

The main difference between the approach formulated here and constraint satisfaction techniques is that, in contrast with the latter, the former subscribes to a knowledge-based view of problem solving, where knowledge is brought in to tackle complexity (Feigenbaum, 1977). Thus, the main goal of the proposed model is to identify the knowledge-intensive tasks and types of application-specific knowledge, which can be exploited during parametric design problem solving. In contrast with this approach, researchers in constraint satisfaction develop efficient algorithms, which can be directly applied to solve problems formulated as networks of constraints. Thus, there is a

fundamental distinction in the goals driving research in knowledge modelling and research in constraint satisfaction. Of course, like other researchers who subscribe to the knowledge-centred paradigm, I believe that "the combinatorics of complex problems can best be handled through the use of domain-specific knowledge" (Wielinga and Schreiber, 1997).

Nevertheless, as demonstrated throughout the chapter, it is possible to integrate the results from the constraint satisfaction literature within a knowledge-intensive framework. For instance, I have shown that - assuming that the relevant knowledge about possible values for unbound parameters is available - techniques such as DSR, which have been developed in the constraint satisfaction literature, can be used to support domain-independent, least commitment strategies.

Chapter 8.
Problem Solving Methods for Parametric Design

In this chapter I discuss a number of problem solving methods for parametric design, constructed by specializing the generic problem solving model presented in the previous chapter. Thus, the proposed model is proven adequate for characterizing a number of approaches to parametric design problem solving. Moreover, the resulting, uniform view of parametric design provides i) a generic framework suitable for comparing and contrasting different methods, ii) an organizational schema providing the overall structure of a library of reusable problem solving components and iii) a search-centred interpretation model which can be used to understand the problem solving role played by the mechanisms and knowledge structures employed by problem solving methods - e.g. the fix mechanism in Propose&Revise.

8.1 INTRODUCTION

In the previous chapter I illustrated a model of parametric design problem solving, which is informed by the parametric design task ontology and by the view of design as search. This model comprises a number of generic tasks, which provide useful building blocks for re-engineering existing problem solving methods for parametric design and for constructing new ones. Moreover, the proposed model also fulfils a practical role, as it can be configured to provide a particular problem solving method for parametric design, *Gen-design-psm*[1].

In this chapter I will substantiate the claim concerning the analytical leverage provided by the generic problem solving model, by constructing a number of problem solving methods for parametric design as specializations of the model. In particular the goal here

---

[1] In what follows I will use the term 'Gen-design-psm' both to refer to the 'generic problem solving model' (i.e. the set of generic tasks for parametric design introduced in the previous chapter) and to refer to the problem solving method which can be constructed by solving the goal specification tasks included in the generic model by means of the (specific) methods discussed in the previous chapter. In general, it should be apparent from the context of the discussion whether the term is being used to refer to the model or the problem solving method.

is to show that this uniform view of problem solving methods provides a number of advantages, including: i) a generic framework suitable for comparing and contrasting different methods, ii) an organizational schema providing the overall structure of a library of reusable problem solving components and iii) a search-centred interpretation model which can be used to understand the problem solving role played by the mechanisms and knowledge structures employed by problem solving methods - e.g. the fix mechanism in Propose&Revise.

8.2 CHARACTERIZING PROBLEM SOLVING METHODS

In order to facilitate the analysis and comparison of alternative problem solving methods, I will make use of a method description framework, which is based on the model presented in the previous chapter. The framework highlights the main types of application-specific knowledge required by a problem solving method, say M, as well as the strategies used by M to carry out the main knowledge-intensive tasks presented in the previous chapter. Specifically, the framework consists of the following thirteen fields.

- **Problem Solving Knowledge.** The generic classes of application-specific knowledge required by a method. For the sake of brevity this field does not include the knowledge roles associated with the specification of the parametric design generic task - e.g. constraints. That is, I will assume that each method subscribes to the parametric design task ontology presented in chapter 6. Moreover, when describing this field I will not consider either mapping knowledge or design operators. The reason for excluding mapping knowledge is that this knowledge is not brought in to tackle problem solving complexity, but is rather a consequence of the domain-independence of the method specifications. Because all methods are characterized in a domain-independent style, they all require mapping knowledge. Thus, this type of knowledge does not help distinguishing between different methods and, in any case, plays a methodological rather than problem solving role. Design operators are also, in a sense, 'special', given that all methods require them. Thus, they will be treated in a separate field - see below - which indicates the kinds of design operators used by a method. If a method does not require some knowledge role introduced in the parametric design task ontology, e.g. some methods do not require a cost function, this aspect will be explicitly highlighted in the description of this field.
- **Constraint Types.** The types of constraints identified by a method. For instance, Propose&Revise distinguishes between *fixable* and *non-fixable* constraints.

- **Additional Subtasks.** The subtasks introduced by a method which are not present in Gen-design-psm.
- **State-Based Control.** The control regime used to carry out task `design-from-state`.
- **Contexts.** The contexts considered by a method.
- **Focus Types.** The types of design foci considered by a method.
- **Focus Selection Policy.** The strategy used to select a design focus.
- **Design Operator Types.** The types of design operators considered by a method.
- **Design Operator Order Policy.** The strategy used to decide the order in which the design operators applicable to the current focus are tried.
- **Available Design Space.** This field indicates what subset of the design space is used as an input to the state selection task. For example, the design space considered by a design method based on a 'strict' hill climbing strategy (i.e. one admitting no backtracking) can be characterized as comprising all and only the maximally extensive design states. Other methods, for instance A*, consider the total search space explored so far, with no additional filtering. When characterizing the design space considered by a method I will make an *assumption of rationality*, i.e. I will implicitly assume that all the states marked as 'dead ends' are not part of the input space.
- **State Selection Policy.** The strategy used to decide which state to expand, at each cycle of the select/expand/evaluate control regime.
- **Completeness.** Is the method complete? That is, will it guarantee to find a solution if this exists?
- **Optimality.** Does it take into account cost-related aspects?

Table 8.1 shows the result of applying the method description template to Gen-design-psm. This problem solving method makes use of focus and operator selection knowledge, as well as knowledge which allows a problem solver to determine dynamically, for an unbound parameter, the set of feasible assignments. It is interesting to note that, as a consequence of the state-based control regime used by Gen-design-psm, this method does not require a global cost function. At each cycle of the design process, the state selection mechanism always returns at most one consistent and maximal state. Therefore, any further, cost-based discrimination is unnecessary.

Method Class	Gen-design-psm
Problem Solving Knowledge	Focus Selection Knowledge Operator Selection Knowledge Available Parameter Values (required by DSR strategy) (Cost Function not needed)
Constraint Types	Constraint
Additional Subtasks	None
State-Based Control	Extend Incomplete State
Contexts	Extend
Focus Types	Parameter
Focus Selection Policy	DSR strategy + Focus Selection Knowledge
Design Operator Types	Design Extension Operator
Design Operator Order Policy	Operator Selection Knowledge
Available Design Space	All feasible states generated by the depth-first search algorithm.
State Selection Policy	1) Violated Constraints: No 2) Design Model: Max
Completeness	Yes
Optimality	Local

Table 8.1. Synoptic description of Gen-design-psm

8.3 PROPOSE&BACKTRACK

Propose&Backtrack is the method used by Runkel et al. (1996), to solve the VT problem without resorting to the use of fixes. This method implements a simple, depth-first control regime in which, at each step of the design process, unassigned parts are selected and assigned. The assignment is carried out by selecting a value from the value range of the selected part. If the assignment results in an inconsistency, a different value for the part is tried. If there are no values left, chronological backtracking is used to go back to a consistent state. When deciding which value to assign to a part, Propose&Backtrack assumes the existence of local preference knowledge, which is used to rank the available

parameter values. In the case of the VT application, this knowledge is based on the costs assigned to the relevant procedures and fixes.

Method Class	**Propose&Backtrack**
Problem Solving Knowledge	Value Range Preference Knowledge (Cost Function not needed)
Constraint Types	Constraint
Additional Subtasks	None
State-Based Control	Extend Incomplete State
Contexts	Extend
Focus Types	Parameter (Part)
Focus Selection Policy	?? (not enough information provided in the literature)
Design Operator Types	Design Extension Operator
Design Operator Order Policy	Use Value Range Preference Knowledge
Available Design Space	All feasible states generated by the depth-first search algorithm.
State Selection Policy	1) Violated Constraints: No 2) Design Model: Max
Completeness	Yes
Optimality	Local

Table 8.2. Synoptic description of Propose&Backtrack

Table 8.2 shows the main features of Propose&Backtrack in terms of the proposed framework. These are discussed below.

Problem Solving Knowledge. Propose&Backtrack makes use of preference knowledge to prioritize the values available for a given parameter. This knowledge can be seen as a special case of the operator preference knowledge discussed in section 7.2.2.2.

Constraint Types. Propose&Backtrack uses constraints only to check whether an assignment is consistent. The definition of class `constraint` included in the parametric design task ontology is sufficient to guarantee this functionality.

Additional Subtasks. None required.

State-Based Control. Propose&Backtrack uses the same control regime as Gen-design-psm. If the current state is incomplete, then a successor state is generated by means of a design extension operator. If a constraint violation is encountered, then the method backtracks to the previous state.

Contexts. Only one context is considered by Propose&Backtrack, `:extend`.

Focus Types. Only one type of foci is needed by Propose&Backtrack: parameters.

Focus Selection Policy. The available descriptions do not provide enough information on this aspect of the method. Given the constraint satisfaction viewpoint adopted by the DIDS researchers it is expected that techniques like DSR are exploited to choose which part to assign next.

Design Operator Types. Only design extension operators are required by Propose&Backtrack. In practice there is a one-to-one mapping between operators and parameter values.

Design Operator Order Policy. Operators map to parameter values and these are prioritized by means of value range preference knowledge.

Available Design Space. This comprises all the feasible states generated by means of the depth-first search control regime.

State Selection Policy. At each cycle of the design process Propose&Backtrack selects the maximally extensive consistent state in the design space. Because of the state-based control regime used by the method, only one consistent and feasible state can exist at each moment, for a given size of design model. Therefore there is no need for a cost-based state selection mechanism.

Completeness. Propose&Backtrack employs a complete, depth-first search method, which will eventually find a solution, if this exists.

Optimality. Propose&Backtrack is a greedy algorithm which uses local preference knowledge about parameter values to make locally optimal steps. Of course, this local optimality is subject to horizon effects.

In summary Propose&Backtrack is a simple refinement of Gen-design-psm. Like the latter, Propose&Backtrack relies on local preference knowledge, to perform 'good' design extensions, and on chronological backtracking, to explore alternative paths, in case an inconsistency or a dead end is encountered. Therefore, its performance relies on two crucial aspects of the problem: i) that the available local preference knowledge is effective in guiding the search process and ii) that the problem exhibits only a *weakly connected* (Sadeh and Fox, 1996) constraint network. These are pretty strong assumptions, which are rarely jointly satisfied, except in relatively simple parametric

design problems (like, for instance, the Sisyphus-I office allocation problem). For example, the control regime used by Propose&Backtrack, chronological backtracking, is too weak to tackle VT efficiently (Runkel et al., 1996). Moreover, experimental results from the KMI office allocation problem - see next chapter - suggest that the available local control knowledge (e.g. focus and operator selection knowledge) does not provide enough guidance to achieve good solutions by means of a Propose&Backtrack approach.

8.4 HILL-CLIMBING

A straightforward refinement of Gen-design-psm can be constructed by replacing its state-based control regime (`extend-incomplete-state`) with one based on a *hill climbing* approach - see definition of method `hc-control` in the box below.

```
(def-class HC-CONTROL (decomposition-method)
  ((has-input-role :value has-design-state
                   :value has-design-context)
   (has-output-role :value has-successors)
   (has-design-state :type design-state)
   (has-successors :type list
                   :default-value nil)
   (has-design-context :type design-context
                       :default-value :extend)
   (has-goal-expression
    :value (kappa (?task ?s)
                  (state-fully-expanded
                   (role-value ?task has-design-state))))
   (has-body
    :value
    (lambda (?psm)
      (in-environment
       ((?state . (role-value ?psm has-design-state))
        (?design-model . (the ?dm (has-design-model
                                   ?state ?dm)))
        (?constraints . (role-value ?psm has-constraints))
        (?parameters . (role-value ?psm has-parameters)))
       (if (deadend-state ?state)
         :nothing
         (if (constraint-violations ?state ?constraints)
           (tell (deadend-state ?state))
           (if (state-complete ?state ?parameters)
             (tell (solution-state ?state))
             (achieve-generic-subtask
                ?psm generate-all-successors
                has-design-state (role-value ?psm has-design-state)
                has-design-context (role-value
                                    ?psm has-design-context))))))))

  :own-slots ((tackles-task-type design-from-state)
              (has-generic-subtasks '(generate-all-successors))))
```

As shown by the above definition, the difference between the state control required to model a hill climbing approach and the one used by Gen-design-psm is simply that the former generates all successors of the current state, while the latter generates just one. In

particular, given the control regime defined by method `Hc-control`, the 'hill climbing-type' behaviour can be achieved by using cost evaluation as the final criterion to choose from all the states generated at a particular cycle of the design process - see 'state selection policy' field in table 8.3.

Hence, it is easy to see that both Gen-design-psm and hill climbing follow a similar philosophy: a 'good' (if not optimal) solution can be in some cases achieved by repeatedly making locally optimal steps. The difference between Gen-design-psm and hill climbing concerns the approach taken to make these locally optimal steps. The former relies on application-specific knowledge and domain-independent heuristics; the latter on a more expansive (and expensive) search policy. Specifically, Gen-design-psm provides slots for specifying focus and operator selection knowledge. If such knowledge is assumed to be perfect, then the next best state will be selected. Hill climbing achieves the same result by evaluating all possible successor states and then choosing the best one according to the given evaluation function. Thus, it can be seen as an alternative to Gen-design-psm for those application domains where local preference knowledge is not available.

Table 8.3 summarizes the main features of the proposed hill climbing adaptation of the generic parametric design model. In what follows, the resulting problem solving method will be referred to as *Hc-design.*

As shown in the table (and already pointed out), the main difference between Hc-design and Gen-design-psm is that the former does not require local preference knowledge, replacing it with a more expensive state-based control regime. On average, this control regime increases the size of the search space by a factor $((k - 1) / 2)^d$, where d is the depth of the search tree and k is the average branching factor. On the one hand this situation is paradigmatic of the trade-off between knowledge and complexity; on the other hand, this comparison does not take into account the time spent deciding on the appropriate focus and operator selection. If these tasks require a complex domain analysis, or if such knowledge is not available, then hill-climbing becomes a feasible alternative.

The close connection between Gen-design-psm and Hc-design can also be seen by noting that the two methods essentially subscribe to the same state selection policy. The main difference is that, in the case of Gen-design-psm, the third step (minimizing cost) is redundant as the first two are guaranteed to provide enough discriminatory power. Nevertheless, if we assume that the local preference knowledge used by Gen-design-psm always selects the next best state, then the two methods generate exactly the same search space and explore it in the same sequence.

Finally, it is easy to see that Hc-design can be transformed into a *best-first* search algorithm (Rich and Knight, 1991) simply by swapping steps (2) and (3) in the state selection policy.

Method Class	**Hc-design**
Problem Solving Knowledge	None (but uses Cost Function for evaluating state)
Constraint Types	Constraint
Additional Subtasks	HC-Control
State-Based Control	HC-Control
Contexts	Extend
Focus Types	Parameter
Focus Selection Policy	None (not needed)
Design Operator Types	Design Extension Operator
Design Operator Order Policy	None (not needed)
Available Design Space	All feasible nodes generated so far.
State Selection Policy	1) Violated Constraints: No 2) Design Model: Max 3) Cost: Min
Completeness	Yes
Optimality	Local

Table 8.3. Synoptic description of Hc-design

8.5 A*-BASED DESIGN

The A* algorithm (Rich and Knight, 1991) is another search strategy which can be easily integrated into the given framework, to produce a problem solving method for parametric design. This algorithm substitutes the state selection strategy used by hill climbing with one attempting to achieve global optimality. Specifically, the A* algorithm makes use of a heuristic cost function to estimate the distance between the current state and a solution. The result is a generalized notion of state cost, which adds this heuristic estimate to the result obtained by means of the 'ordinary' cost function. Thus, it is possible to avoid the horizon effects typical of locally optimal search approaches, such as Gen-design-psm and hill climbing.

The box below shows the cost evaluation method used by an A*-style design method (*A-star-design*). This definition assumes the existence of an input role, `has-cost-estimate-function`, defining the heuristic state evaluation function. The method simply computes the current state cost and the estimated distance to a goal node and adds these two measures by means of the relevant *cost merging function*. This is defined as part of a parametric design task specification and is retrieved from the current parametric design problem by means of the relation `has-cost-sum-function`.

```
(def-class A*-COST-EVALUATION (primitive-method) ?psm
  ((has-input-role :value has-cost-estimate-function)
   (has-cost-estimate-function :type cost-function)
   (has-body
    :value (lambda (?psm)
             (in-environment
              ((?task . (role-value ?psm has-current-task))
               (?state . (role-value ?psm has-design-state))
               (?design-model . (the ?dm (has-design-model
                                          ?state ?dm)))
               (?cost-fun . (role-value ?psm has-cost-function))
               (?h-cost-fun . (role-value
                               ?psm has-cost-estimate-function))
               (?add-fun . (the ?rel (has-cost-sum-function
                                      ?task ?rel)))
               (?cost . (call ?add-fun
                              (call ?cost-fun ?design-model)
                              (call ?h-cost-fun ?design-model))))

              (do
                (tell (state-cost ?state ?cost))
                ?cost)))))
  :own-slots ((tackles-task-type  evaluate-cost)))
```

As shown in table 8.4, there are two main differences between A-star-design and Hc-design, which are associated with cost evaluation and state selection. In both cases A-star-design uses a global approach, while Hc-design uses a local one. In particular the state selection criterion used by A* gives priority to cost over size of design model, in contrast with hill climbing. This approach is of course much more expensive in principle and therefore A* relies on the heuristic cost function - normally called the *h* function - to try and focus the search process. The main requirement imposed on the *h* function is that this should be 'optimistic' - i.e. it should never overestimate the distance to a goal node. A heuristic function for which this property holds is called *admissible*. Gaschnig (1979) analysed the behaviour of A* with various degrees of errors in the heuristic function and found that, unless very precise heuristics are used, the complexity of the search space becomes exponential in the worst case. The problem with A* is that such precise heuristics are in practice very difficult to achieve, unless much is known about the structure of the search space. As a result, it is very difficult to apply A* effectively to a parametric design problem. However, as in the case of hill climbing it is possible to make use of A*-style strategies to solve specific sub-problems, for which a good

heuristic function can be defined. For example, in (Zdrahal and Motta, 1995) we discuss a modification of a Propose&Revise problem solver, which applies A* to the design revision process.

Method Class	**A-star-design**
Problem Solving Knowledge	Heuristic Cost Function
Constraint Types	Constraint
Additional Subtasks	HC-Control
State-Based Control	HC-Control
Contexts	Extend
Focus Types	Parameter
Focus Selection Policy	None (not needed)
Design Operator Types	Design Extension Operator
Design Operator Order Policy	None (not needed)
Available Design Space	All feasible nodes generated so far.
State Selection Policy	1) Violated Constraints: No 2) Cost: Min 3) Design Model: Max
Completeness	Yes
Optimality	Global

Table 8.4. Synoptic description of A-star-design

8.6 BEYOND UNIFORM APPROACHES TO PARAMETRIC DESIGN

The methods described in the previous sections (with the possible exception of Gen-design-psm) only require limited amounts of application-specific knowledge. While this property has the advantage of facilitating the knowledge acquisition process and the applicability of the method in question, it often leads to poor performance - see e.g. applications discussed in next chapter and also discussion by Runkel et al. (1996) on the performance of Propose&Backtrack. Moreover, in the case of A-star-design, such 'limited knowledge' takes the form of a heuristic evaluation function, which is in practice very difficult to define for a problem of some complexity. As pointed out when discussing the role of search in problem solving, search is inevitable if we don't know

how to reach a solution node directly. But typically, the reason for being unable to reach a solution node directly is because our knowledge about the problem space is limited. In these cases it is unlikely that we would be able to meet the requirements imposed by A*.

A second problem with the three methods discussed earlier concerns the reliance on a uniform problem solving approach. As Stefik (1995) points out, "Seldom does a single search method provide an adequate problem-solving framework for a complex task." In particular a uniform problem solving approach inevitably restricts the types of problem solving knowledge which can be applied to the problem. For this reason researchers have developed problem solving methods which distinguish between multiple phases and introduce a richer variety of knowledge structures. A famous example of such a method (more precisely class of methods) in the design area is Propose&Revise (Marcus and McDermott, 1989), which differentiates between design extension and revision and introduces the appropriate knowledge roles for both phases.

In the next sections I will discuss two methods which are part of the parametric design library, and which rely on a dual-context problem solving strategy. One of them is the aforementioned Propose&Revise, the other is called *Propose&Improve* and can be used for parametric design problems which require to achieve, or at least approximate, an optimal solution. In order to model these methods I will introduce a second type of design operators, called *design modification operators*. These are operators (such as fixes) which modify, rather than extend, design models - i.e. which replace the values of already bound parameters.

8.7 DESIGN MODIFICATION OPERATORS

Although it is possible to represent all kinds of operators in a homogeneous way, as functions which map an input design model to an output one, it is useful to distinguish between operators which play different roles in a problem solving model. For example, in section 7.4.10.2 I defined a class of operators which model design extension steps. Here, I will introduce a second class of operators, `design-modification-operator`, which is meant to be used to 'correct' or 'improve' the values of bound parameters, in cases where these parameters lead to inconsistencies or sub-optimal solutions.

The box below shows the basic type of design modification operator. Its body is defined as a function which takes as inputs a design model, say `?dm`, and a parameter, say `?p`, which is bound to some value, say `?v`, in `?dm`. The output is a design model, in which `?p` is bound to a value other than `?v`.

```
(def-class DESIGN-MODIFICATION-OPERATOR (design-operator)
  "The body of a basic design modification operator is a
   lambda expression which takes two arguments, a parameter
   and a design model.
   The output is a design model which differs from the input model
   at least with respect to the input parameter"
  ((applicable-to-parameters
    :default-value '(setofall ?x (parameter ?x))
    :type function-expression
    :documentation "An expression which returns the set
                    of parameters which can be modified
                    by means of this operator")
   (body :type design-modification-operator-body)))
```

```
(def-class DESIGN-MODIFICATION-OPERATOR-BODY
  (lambda-expression) ?x
  :no-op (:constraint
          (and (nth-domain ?x 1 parameter)
               (nth-domain ?x 2 design-model)
               (=> (= ?z (call ?x ?p ?d))
                   (and  (design-model ?z)
                         (has-value ?p ?v ?d)
                         (not (has-value ?p ?v ?d)))))))
```

In order to integrate this class of operators with the problem solving model discussed in the previous chapter we need a new method for task `try-design-operator`, applicable when the selected operator is a design modification one. This method is defined in the next box.

```
(def-class TRY-DESIGN-MODIFICATION-OPERATOR (primitive-method)
  ((has-body
    :value
    (lambda (?psm)
      (in-environment
       ((?dm . (role-value ?psm 'has-design-model))
        (?focus . (role-value ?psm 'has-design-focus))
        (?dm-new . (apply-design-modification-operator
                    ?focus ?dm (role-value ?psm
                                     has-design-operator))))
       (if (not (= ?value :nothing))
         (achieve-generic-subtask
          ?psm new-design-state
          has-design-model ?dm-new))))))

  :own-slots ((tackles-task-type try-design-operator)
              (applicability-condition
               (kappa (?task)
                      (design-modification-operator
                       (role-value
                        ?task has-design-operator))))))

(def-function apply-design-modification-operator (?param ?dm ?op)
  :body (apply-design-extension-operator ?param ?dm ?op))
```

8.8 PROPOSE&IMPROVE

The basic idea underlying the Propose&Improve method is that global optimality can be achieved or approximated by dividing the problem solving process into two phases: a first one, which is concerned with finding a solution, and a second one, which attempts to improve it. The method described here is similar 'in spirit' to approaches such as *genetic algorithms* (Goldberg, 1989) or *simulated annealing* (Kirkpatrick et al., 1983), which first generate (or attempt to generate) a solution quickly and then modify it to produce a better one (or to add to the pool of current solutions in the case of genetic algorithms).

Here I will instantiate this generic (and admittedly vague) idea of dividing problem solving into a 'propose' and an 'improve' phase in the context of the proposed problem solving framework for parametric design. In particular, in contrast with approaches such as simulated annealing, which base the solution modification process on a random perturbation of the solution parameters, the method proposed here grounds this process on a detailed cost analysis of the current best solution. More precisely, it identifies the solution components (i.e. parameters) which are currently most expensive, and then uses specific improvement operators to modify their values.

8.8.1 Modelling Propose&Improve

As shown by the definition given in the following box, the Propose&Improve class of problem solving methods refines Gen-design-psm in two ways: it introduces a new role - `has-parameter-cost-fun` - and specializes the goal of a generic parametric design task (which is to find a complete and valid design model), by introducing a criterion of *p&i-optimality*. This criterion provides a method-specific, operational interpretation of the notion of optimal design model, which is expressed in non-operational terms in the task ontology. In particular a solution state is said to be *p&i-optimal* if the method is unable to find a better solution with limited computational resources. This criterion is represented by stating that, if the current solution state cannot be improved, i.e. if this state has been fully expanded, then the current solution is p&i-optimal.

```
(def-class p&i-psm (gen-design-psm)
  "The goal of a p&i method is to find a solution
   which is 'p&i-optimal'. By this we mean that
   the solution cannot be further improved by means of
   the p&i method"
  ((has-input-role :value has-parameter-cost-fun)
   (has-parameter-cost-fun :type parameter-cost-fun)
   (has-goal-expression
    :value (kappa (?psm ?state)
                  (and (tackles-task ?psm ?task)
                       (p&i-optimal ?state ?task))))))
```

```
(def-relation p&i-optimal (?state ?task)
  "We define a p&i-optimal state as a solution state which
   has been fully expanded.  If this is the case then it means we
   have tried to improve it and failed"
  :iff-def (and (has-design-model ?state ?dm)
                (achieved ?task ?dm)
                (state-fully-expanded ?state)))
```

Role `has-parameter-cost-fun`, in the definition of class `p&i-psm`, refers to application-specific knowledge which can be used to identify the parameter (or set of parameters) which is expensive in the current solution. That is, this definition states the requirement for a *detailed cost model*, expressed in terms of the cost of each component. A problem solver subscribing to this requirement is then able to use the detailed cost model to identify which parts of the design model to try and improve.

The class of parameter cost functions is formally defined as follows.

```
(def-class parameter-cost-fun (binary-function)
  "A parameter cost function computes the cost of
   a parameter in a design model"
  :constraint (and
                (nthdomain ?fun 1 ?parameter)
                (nthdomain ?fun 2 ?design-model)
                (range ?fun ?cost)))
```

8.8.2 Task-method structure of Propose&Improve

The control method `p&i-control`, shown below, defines the state-based control of the Propose&Improve method. This control regime is very similar to the one used by Gen-design-psm. The only difference is that, when faced with a complete and consistent state, P&i-design will try to improve on it, rather than simply declaring it a 'success' state. To improve the current state, `p&i-control` simply calls the generic task `generate-state-successor` in an `:improve`, rather than `:extend` context. Thus, both the design extension and the design improvement phases are carried out by means of the same sequence of subtasks; the differentiation between these two phases is achieved by parametrizing a uniform control regime in terms of different design contexts. The advantage of this approach is that it allows us to specify different focus and operator

selection and collection mechanisms in a modular style, while keeping the same control structure in both phases.

```
(def-class  P&I-CONTROL (decomposition-method)
  ((has-input-role :value has-design-state)
   (has-output-role :value generates-design-state)
   (has-design-state :type design-state)
   (generates-design-state :type design-state)
   (has-body
     :value
     (lambda (?psm)
      (in-environment
       ((?state . (role-value ?psm has-design-state))
        (?design-model . (the ?dm (has-design-model
                                    ?state ?dm)))
        (?constraints . (role-value ?psm has-constraints))
        (?parameters . (role-value ?psm has-parameters)))
       (if (deadend-state ?state)
         :nothing
         (if (constraint-violations ?state ?constraints)
           (tell (deadend-state ?state))
           (if (state-complete ?state ?parameters)

             ;;current model is complete, let's improve it
             (achieve-generic-subtask
               ?psm
               generate-state-successor
               has-design-state ?state
               has-design-context :improve)

             ;;current model is incomplete, let's extend it
             (achieve-generic-subtask
               ?psm
               generate-state-successor
               has-design-state ?state
               has-design-context :extend))))))))

   :own-slots ((tackles-task-type  design-from-state)
               (has-generic-subtasks generate-state-successor)))
```

In order to complete the task structure of the Propose&Improve design method it is necessary to specify the mechanisms for carrying out focus and operator collection and selection, which are relevant to an `:improve` context. These are described in the next sections.

8.8.2.1 Focus collection in `:improve` context

All parameters are potentially foci for improvement, and therefore the default method for foci collection simply returns all design parameters.

```
(def-class collect-all-parameters  (primitive-method)
  ((has-body
    :value (lambda (?psm)
             (role-value ?psm has-parameters))

   :own-slots ((tackles-task-type collect-state-foci)
               (applicability-condition
                (kappa (?task)
                       (= (role-value ?task 'has-design-context)
                          :improve)))))))
```

8.8.2.2 *Focus selection in* `:improve` *context*

In general, application specific knowledge is needed to infer which part of the design to try and improve, without increasing the overall cost of the design. If this knowledge is not available, a good default is to select the currently most expensive part.

```
(def-class SELECT-MOST-EXPENSIVE-PARAMETER (primitive-method)
  ((has-body
    :value (lambda (?psm)
             (the-most-expensive-parameter
              (role-value ?psm has-design-foci)
              (the ?dm (has-design-model
                         (role-value ?psm has-design-state)
                         ?dm))
              (role-value ?psm has-parameter-cost-fun))))
   :own-slots ((tackles-task-type select-design-focus)
               (applicability-condition
                (kappa (?task)
                       (= (role-value ?task has-design-context)
                          :improve)))))
```

8.8.2.3 *Operator collection and selection in* `:improve` *context*

Given a specific parameter, say p, the operators which are potentially useful are the design modification operators which are applicable to p.

```
(def-class Collect-design-modification-operators (primitive-method)
  ((has-body
    :value
    (lambda (?psm)
      (setofall ?op
                (and (design-modification-operator
                      ?op
                      applicable-to-parameters ?l)
                     (member (role-value ?psm 'has-design-focus)
                             (eval ?l)))))))

  :own-slots ((tackles-task-type collect-focus-operators)
              (applicability-condition
               (kappa
                (?task)
                (and (= :improve
                        (role-value
                         ?task 'has-design-context))
                     (parameter
                      (role-value
                       ?task 'has-design-focus)))))))
```

As far as operator selection is concerned, there is no need to provide a specialized method for an `:improve` context. We can assume that application-specific, operator preference knowledge is available both for design extension and design modification operators.

8.8.3 Analysis of Propose&Improve

As shown in table 8.5, the design-centred adaptation of Propose&Improve presented here provides a greater range of knowledge roles than any of the methods described in the previous sections. It divides the design process in two phases. The first phase consists of a straightforward extend-and-backtrack process, which is modelled on Gen-design-psm. This process guarantees that a solution is found, if it exists, and uses local preference knowledge to try and approximate an optimal solution. Once a solution is reached, a (global) hill climbing-type approach is adopted, which tries to improve the solution until one is generated which cannot be bettered by means of the available operators. However, it is important to note that, in contrast with Hc-design, the strategy used by P&i-design does not backtrack during the design improvement phase. If it is not possible to improve on the current best solution, then the method stops and returns it. The reason for this strategy is that to allow backtracking during the improvement phase is essentially the same as carrying out a complete search over the space of possible improvements. Of course, this would be normally very inefficient, given the large space of parameters and the small probability that improvements will be found by modifying all but a small subset of expensive parameters. For this reason, the specification proposed here does away with backtracking. However, it is possible also to envisage alternative solutions, which allow limited backtracking, by introducing tighter foci collection policies

(e.g. considering only the parameters over a certain threshold) and improvement cut-off limits (e.g. rejecting all improvements which fall below a certain threshold).

Propose&Improve is particularly suitable for parametric design problems in which optimality is an important solution criterion and which are characterized by a *dynamic cost function* - i.e. a cost function in which the cost of an assignment can only be fully evaluated once a number of other assignments have been completed. This situation often arises in resource assignment problems, such as timetabling and office allocation (Poeck and Puppe, 1992). In the next section I will illustrate this point by discussing the application of P&i-design to the KMI office allocation problem, an application characterized by a dynamic cost function and by a strong optimality criterion.

Method Class	Propose&Improve
Problem Solving Knowledge	Focus Selection Knowledge Operator Selection Knowledge Available Parameter Values Detailed Cost Function
Constraint Types	Constraint
Additional Subtasks	Improve-design
State-Based Control	Extend-design + Improve-design
Contexts	Extend, Improve
Focus Types	Parameter
Focus Selection Policy	Open (Extend) Most expensive (Improve)
Design Operator Types	Design Extension Operator Design Modification Operator
Design Operator Order Policy	Operator Preference Knowledge
Available Design Space	All feasible nodes generated so far (Propose) Currently best state (Improve)
State Selection Policy	1) Violated Constraints: No 2) Design Model: Max 3) Cost: Min
Completeness	Yes (solution) No (optimal solution)
Optimality	Both local and global (not guaranteed)

Table 8.5. Synoptic description of Propose&Improve

8.9 PROPOSE&REVISE

8.9.1 Introduction

Propose&Revise is one of the role-limiting methods included in the library developed at the end of the eighties by McDermott's group (Marcus, 1988). This method informed the architecture of the SALT knowledge acquisition tool (Marcus and McDermott 1989) and

was used for solving i) the VT elevator design application (Marcus et al., 1988; Yost and Rothenfluh, 1996) and ii) a flow-shop scheduling problem (Stout et al., 1988).

In this section I will carry out a 'rational reconstruction' of Propose&Revise, characterizing it in terms of the generic model of parametric design problem solving presented in the previous chapter. However, in contrast with other analyses of Propose&Revise which exist in the literature (Fensel, 1995b; Wielinga et al., 1995; Motta et al., 1994b), my aim here is not just to produce a knowledge-level description of Marcus' role-limiting method, but rather to tease out and formalize the essential elements of a generic Propose&Revise problem solver. Hence, rather than producing the specification of a particular method, the analysis conducted here will characterize a class of methods, the *P&R-class*, whose elements share the essential properties of a Propose&Revise model of problem solving and are distinguished on the basis of finer-grained properties. The advantage of this approach is twofold: i) it helps clarifying the nature of Propose&Revise problem solving, in particular abstracting from the sometimes idiosyncratic design decisions which were included in Marcus' original method and ii) provides a generic template which can be instantiated in several different ways, to produce alternative *P&R-type* methods. As a result this analysis affords a better understanding of the space of Propose&Revise methods. For instance it highlights the reasons for choosing alternative modelling solutions and the trade-offs between different approaches.

To avoid possible terminological confusion, in what follows I will use the term Propose&Revise as a generic name for a method in the P&R-class and *P&R-Marcus* to refer to the original role-limiting method.

8.9.2 Differentiating Propose&Revise

The basic feature of a Propose&Revise method is that it divides the design process into two main subtasks, *propose* and *revise*. The former is carried out in order to extend incomplete, but consistent designs; the latter to restore consistency (i.e. to remove the relevant constraint violations). Obviously, given that all the methods discussed previously also include a design extension activity (i.e., a propose task), it is clear that the main difference between Propose&Revise and other classes of methods is related to design revision. Specifically, all the methods presented in sections 8.3-8.8 follow the same *consistency-centred* approach to problem solving: when an inconsistent state is encountered, this is marked as *nogood* and an alternative, consistent state is selected. In contrast with these methods, a Propose&Revise problem solver operates on inconsistent states directly, by means of special-purpose design modification operators, usually called *fixes*. Thus, at a coarse-grained level of description, we can differentiate the P&R-class from other classes of methods in terms of three different viewpoints.

- **Knowledge Viewpoint.** From a knowledge-centred point of view the main difference between Propose&Revise and other methods is that this introduces a class of design modification operators which operate on inconsistent states and whose purpose is to remove constraint violations. All the other methods discussed so far simply discard inconsistent states.
- **Control Viewpoint.** From a control viewpoint Propose&Revise can be seen as performing *knowledge-based* (Marcus et al., 1988), rather than *general-purpose backtracking.*
- **State-centred Viewpoint.** All the approaches discussed in the previous sections assume that only consistent states can be found on a *solution path* (i.e. a path from an initial to a goal state). Propose&Revise does away with this assumption, by providing mechanisms which support state transitions from inconsistent to consistent states.

Of the three viewpoints presented above, the state-centred one is a particularly interesting one, as it highlights the fact that a Propose&Revise problem solver does away with the consistency-centred approach used by all the other methods. This aspect is important also from a cognitive point of view given that consistency-first approaches are at odds with much literature on design problem solving, which characterizes this process as one in which partially correct designs are iteratively revised (Chandrasekaran, 1990; Cross et al., 1993).

In summary a preliminary, coarse-grained analysis of Propose&Revise shows that, by introducing fixes and a design revision task, this class of methods introduces the important novel principle that inconsistent states can also be the object of the design process. This principle of *constraint violation tolerance* opens up a number of possible strategies for design problem solving. For instance this principle can be instantiated in case-based design by relaxing the constraint that only consistent design models need to be stored in a library of cases. In such a scenario, a case-based design problem solver could select the design model in the library which most closely match the current specification (Zdrahal and Motta, 1996), regardless of consistency issues, and then repair eventual inconsistencies by means of *repair methods* (Minton et al., 1992).

In the next section I will introduce the basic modelling constructs required to define a Propose&Revise problem solver as a refinement of the Gen-design-psm model.

8.9.3 Introducing fixes.

The box below shows ontologically minimal definitions of fixes and fixable constraints: a fix is simply a design modification operator associated with one or more constraint violations; a fixable constraint is a constraint for which a fix has been defined. This

definition specifies that the only important aspect of a fix is its association to a constraint and does not introduce any further modelling commitments.

```
(def-class DESIGN-FIX (design-modification-operator)
  ((applicable-to-constraints
    :type function-expression
    :documentation "An expression which returns the set
                    of constraints which can be solved by
                    this fix")))

(def-class FIXABLE-CONSTRAINT (constraint) ?x
  "A fixable constraint is a constraint which
   can be fixed by some relevant design fix"
  :iff-def (exists
            ?f
            (and (design-fix ?f)
                 (member ?x (the ?l
                            (applicable-to-constraints ?f ?l))))))
```

8.9.4 Task-method structure of Propose&Revise

8.9.4.1 EMR vs. CMR architectures

The control regime used by P&R-Marcus attempts to fix inconsistencies as soon as these are encountered. In particular, any transition between two inconsistent states is considered 'nogood', unless it monotonically reduces the number of constraint violations.[2] A diagrammatic representation of the behaviour of P&R-Marcus is shown in figure 8.1. Here I use white-filled circles, to indicate consistent design states, and black-filled ones, to indicate inconsistent ones. The diameter of a black-filled node gives an indication of the number of constraints violated by the associated state. As shown in the figure, fix applications which do not decrease the number of constraint violations are rejected - these are marked with a square. Thus, in a sense, the P&R-Marcus approach only 'timidly' moves away from consistency-centred approaches; while inconsistent states can be part of a solution path, the method tries to minimize their occurrence in a solution path.

---

[2] This is only an approximation of the behaviour of Marcus' EMR architecture, which rejects any fix application which either fails to resolve the currently selected constraint violation or introduces a new violation (Yost and Rothenfluh, 1996). Thus, it is possible in principle to construct scenarios in which EMR rejects fix applications which monotonically reduce the number of constraint violations. For instance let's consider a situation in which two constraint violations, cv_1 and cv_2, are violated in state s_i and the application of a fix produces a new state, s_j, where cv_1 and cv_2 are satisfied but a third constraint, cv_3, is violated. In such a case EMR will discard s_j despite the fact that the state transition has monotonically decreased the number of constraint violations.

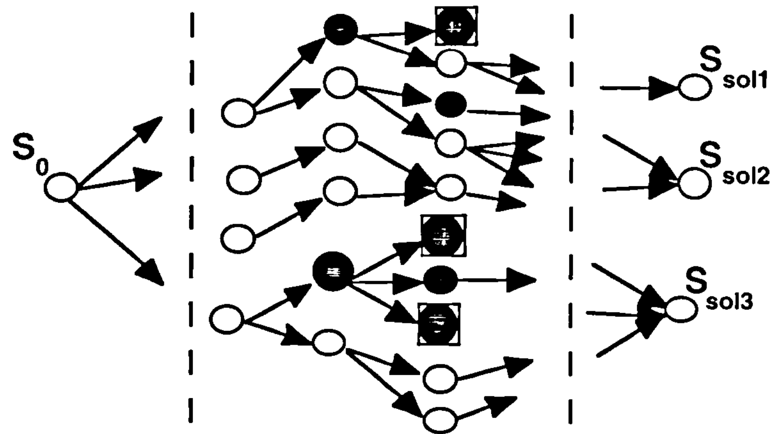


Figure 8.1. Design space in P&R-Marcus problem solving.

In (Motta et al., 1994b) we dubbed this control regime *Extend-Model-then-Revise* (EMR). An alternative to EMR is the *Complete-Model-then-Revise* (*CMR*) approach, in which revision only takes place once the design model has been completed. An advantage of CMR is that, because all constraint violations are tackled after the completion of the design extension process, it is therefore possible to reason about the relations between constraints, parameters and fixes, and about the fix application process itself. For instance it is possible in a CMR approach to make use of techniques such as the *min-conflict heuristic*, which improve the efficiency of the constraint satisfaction process in the average case (Minton et al., 1992). In particular we used a CMR approach when developing a solution to the Sisyphus-II VT elevator design problem (Motta et al., 1994b; 1996).

8.9.4.2 Modelling Propose&Revise control regimes

The EMR and CMR control regimes can be modelled by introducing a new subtask, `revise-design`, and modifying the control regime used by `gen-design-psm`, so that this new subtask is invoked when constraint violations occur. Here I will only give the definition of `emr-control`; `cmr-control` can be defined trivially by swapping the order in which the completeness and constraint violations checks are carried out.

```
(def-class emr-control (decomposition-method)
  ((has-input-role :value has-design-state)
   (has-output-role :value generates-design-state)
   (has-design-state :type design-state)
   (generates-design-state :type design-state)
   (has-body
    :value
    (lambda (?psm)
      (in-environment
       ((?state . (role-value ?psm has-design-state))
        (?design-model . (the ?dm (has-design-model
                                   ?state ?dm)))
        (?constraints . (role-value ?psm has-constraints))
        (?parameters . (role-value ?psm has-parameters)))
       (if (deadend-state ?state)
         :nothing
         (if (constraint-violations ?state ?constraints)

           ;;if constraints are violated we invoke task revise-
             design
           (achieve-generic-subtask
            ?psm
            revise-design
            has-design-state ?state)

           (if (state-complete ?state ?parameters)
             (tell (solution-state ?state))

           ;;if no constraints are violated and the design is
           ;;incomplete, we extend it.
             (achieve-generic-subtask
              ?psm
              generate-state-successor
              has-design-state ?state
              has-design-context :extend))))))))

  :own-slots ((tackles-task-type  design-from-state)
              (has-generic-subtasks generate-state-successor
                                    revise-design)))
```

Design revision is about moving from a consistent to an inconsistent state: the definition below captures this goal-centred specification of the task, which is independent of specific design revision strategies.

```
(def-class REVISE-DESIGN (goal-specification-task) ?task
  "The goal of the task revise-design is to
   move to an output state which is consistent"
  ((has-input-role :value has-design-state)
   (has-output-role :value has-output-state)
   (has-output-state :type design-state)
   (has-design-state :type design-state)
   (has-goal-expression
    :value (kappa (?task ?s)
                  (and (design-state ?s)
                       (not (constraint-violations ?s ?any)))))))
```

8.9.5 Methods for design revision

In general there are a number of approaches which can be taken when carrying out design revisions. The mechanisms presented in this section provide i) different ways of searching a *revision space* and ii) alternative success criteria. By the term 'revision space' I indicate the sub-design space defined by the application of fixes. Obviously it could be possible to define this library of design revision methods in a functional style, as inferences which do not introduce local search control. In this case the search control mechanism defined by the state selection policy used by a problem solver would apply to the design revision phase as well. However, it is advantageous to decouple the search policy used during design revision from that used for the overall problem solving method. This approach provides for a more flexible framework for Propose&Revise problem solving, which makes it possible to mix and match revision strategies and generic search methods - e.g. to mix an A*-type approach for the overall problem solver with different methods for design revisions.

8.9.5.1 One-step revision

This approach consists of applying all the relevant fixes until a consistent successor state is generated. That is, the method succeeds only if the inconsistency can be solved with one move over the state space. In general this method is appropriate only if there is just one constraint violation associated with the current state. If multiple constraint violations occur, then a number of state transitions are usually needed to restore consistency.

Hence, this method is quite primitive: normally sequences of inconsistent states are acceptable if, for instance, the trend is positive - i.e. the number of constraint violations decreases with each state transition. This method can be defined as follows.

```
(def-class ONE-STEP-REVISION  (primitive-method)
  ((has-body
    :value (lambda (?psm)
             (REPEAT
              (in-environment
               ((?input . (role-value ?psm has-design-state))
                (?output . (achieve-generic-subtask
                            ?psm
                            generate-state-successor
                            has-design-state ?input
                            has-design-context :revise)))
               (if  (achieved ?psm ?output)
                 (RETURN ?output)))))))
   :own-slots ((has-generic-subtasks '(generate-state-successor))
               (tackles-task-type  revise-design)
               (applicability-condition
                ;;this method is only useful if there is only one
                ;;constraint violation in the current design state

                (kappa (?task)
                       (in-environment
                        ((?input . (role-value
                                    ?task has-design-state)))
                        (= (cardinality
                            (the ?cs (constraint-violations
                                      ?state ?cs)))
                           1))))))
```

8.9.5.2 Focus-centred revision

The previous method is applicable to scenarios where there is only one constraint violation, say ?c, and succeeds only if there exists a fix, whose application removes ?c without causing new constraint violations. This method can be generalized to situations in which multiple constraints are violated, by replacing its halting condition with one which stops the fix application process as soon as the current focus is solved. This new method, called `try-fix-focus`, can therefore be used in situations in which there are multiple constraint violations. The idea is that as long as we can fix one of these, then a state transition in the revision space is acceptable. The method is defined as follows.

```
(def-class TRY-FIX-FOCUS (primitive-method)
  ((has-body
    :value (lambda (?psm)
             (REPEAT
                  (in-environment
                   ((?input . (role-value ?psm has-design-state))
                    (?output . (achieve-generic-subtask
                                ?psm
                                generate-state-successor
                                has-design-state ?input
                                has-design-context :revise)))
                   (if (design-state ?output)
                     (in-environment
                      ((?record . (the-state-search-control-record
                                   ?state))
                       (?focus . (the-slot-value
                                  ?record 'has-design-focus))
                       (?dm . (the-slot-value
                               ?output has-design-model)))
                      (if (design-model-satisfies ?dm ?focus)
                        (RETURN ?output))))))))))
  :own-slots ((has-generic-subtasks '(generate-state-successor))
              (tackles-task-type  revise-design)))
```

8.9.5.3 Fix-monotonically

The problem with the focus-centred approach is that in some cases we might not be prepared to accept state transitions which fix the current focus but, for instance, introduce two new constraint violations. For instance P&R-Marcus rejects any fix application which either fails to fix the current focus or introduce new constraint violations. The rationale for this approach is that unless some convergence criterion is introduced on the constraint fixing process then the search process becomes open to potential combinatorial explosion.

The criterion used by P&R-Marcus can be generalized by defining a method which considers successful any fix application which decreases the number of constraint violations. The OCML definition of this method, which is called `fix-monotonically`, is given below.

```
(def-class FIX-MONOTONICALLY (primitive-method)
  ((has-body
    :value (lambda (?psm)
             (REPEAT
              (in-environment
               ((?input . (role-value ?psm has-design-state))
                (?output . (achieve-generic-subtask
                             ?psm
                             generate-state-successor
                             has-design-state ?input
                             has-design-context :revise)))
               (if (design-state ?output)
                 (DO
                   (achieve-generic-subtask
                    ?psm
                    evaluate-consistency
                    has-design-state ?output
                    has-design-context :revise)
                   (if (< (cardinality
                            (the ?cs (constraint-violations
                                      ?output ?cs)))
                          (cardinality
                            (the ?cs (constraint-violations
                                      ?input ?cs))))
                       (RETURN ?output)))))))))
  :own-slots ((has-generic-subtasks '(generate-state-successor
                                      evaluate-consistency ))
             (tackles-task-type  revise-design)))
```

Thus, the approach followed by P&R-Marcus combines and strengthens the criteria used by `fix-monotonically` and `try-fix-focus`: it requires the current focus to be fixed and also rejects any move which produces new constraint violations.

As in the case of Propose&Improve, in order to complete the task structure of the Propose&Revise class of methods it is necessary to specify the relevant mechanisms for carrying out focus and operator collection and selection. These are described in the next sections.

8.9.5.4 Focus collection in `:revise` *context*

The possible foci in a `:revise` context are all the fixable constraint violations.

```
(def-class collect-all-cvs (primitive-method)
  ((has-body
    :value (lambda (?psm)
             (setofall
              ?cv
              (and (fixable-constraint ?cv)
                   (member ?cv
                           (the ?cs (constraint-violations
                                     (role-value
                                      ?psm has-design-state)
                                     ?cs))))))))
  :own-slots ((tackles-task-type collect-state-foci)
              (applicability-condition
               (kappa (?task)
                      (= (role-value ?task
                                     'has-design-context)
                         :revise)))))
```

8.9.5.5 Focus selection in `:revise` context

A good heuristic for focus selection in a `:revise` context is the min-conflict heuristic proposed by Minton et al. (1992). This suggests to try and modify the value of the parameter which is associated with the maximum conflict set (i.e. which is associated with the highest number of violated constraints). While this is a simple heuristic to specify, in order to model it in the current ontology I need to introduce a number of additional ontological commitments, concerning the ability of retrieving the parameters which contribute to a particular constraint. For the sake of brevity I will therefore skip the definition of this part of the Propose&Revise model.

8.9.5.6 Operator collection and selection in `:revise` context

The fixes which are potentially useful to a focus in a `:revise` context are simply those associated with the focus in question.

```
(def-class Collect-applicable-fixes (primitive-method)
  ((has-body
    :value
    (lambda (?psm)
      (setofall ?op
                (and (design-modification-operator
                      ?op
                      applicable-to-constraints ?l)
                     (member (role-value ?psm 'has-design-focus)
                             (eval ?l)))))))

  :own-slots ((tackles-task-type collect-focus-operators)
              (applicability-condition
               (kappa
                (?task)
                (and (= :revise
                        (role-value
                         ?task 'has-design-context))
                     (fixable-constraint
                      (role-value
                       ?task 'has-design-focus)))))))
```

Finally, as in the case of Propose&Improve, there is no need to define a specialized operator selection method for a `:revise` context. We can assume that application-specific operator preference knowledge is available as in the case of `:extend` and `:improve` contexts.

8.9.6 Characterizing the P&R-class of problem solving methods

Table 8.6 shows the main features of the P&R-class of problem solving methods defined as a refinement of Gen-Design-psm. This characterization is independent of the actual revision strategy used by a specific instantiation of this class.

Method Class	Propose&Revise
Problem Solving Knowledge	Focus Selection Knowledge Operator Selection Knowledge Available Parameter Values
Constraint Types	Fixable Constraint Constraint
Additional Subtasks	Revise-design
State-Based Control	Extend-design + Revise-design (EMR or CMR)
Contexts	Extend, Revise
Focus Types	Parameter (Extend) Constraint (Revise)
Focus Selection Policy	DSR strategy + Focus Selection Knowledge + Min-conflict heuristic
Design Operator Types	Design Extension Operator Fix (Design Modification Operator)
Design Operator Order Policy	Operator Selection Knowledge
Available Design Space	All feasible states (Propose) Revision Space (Revise)
State Selection Policy	1) Design Model: Max 2) Violated Constraints: Min 3) Cost: Min
Completeness	Yes
Optimality	Local

Table 8.6 Synoptic description of P&R-type methods.

In particular there are two aspects which are worth emphasizing here.

The first one concerns the integration of knowledge-based and chronological backtracking. In contrast with the control regime used by P&R-Marcus, knowledge-based backtracking in the P&R-class does not replace chronological backtracking but is rather added 'on top' of it. Thus when a constraint violation is encountered, the revision space is explored according to the chosen design revision strategy and, if this fails, then the behaviour 'gracefully degrades' to chronological backtracking. This approach has

two advantages: it avoids the brittleness exhibited by the application of P&R-Marcus to VT (Motta & Zdrahal, 1996; Zdrahal & Motta, 1996; Chapter 9) and combines the advantages in terms of reuse and modularity associated with Propose&Backtrack (Runkel et al., 1996) with the efficiency provided by the local search through a space of fixes.

Another important aspect of the P&R-class concerns its unique state selection policy, which gives priority to the size of design models over cost and consistency. Intuitively, the idea of a Propose&Revise approach is that backtracking needs to be avoided: the currently most complete model should be operated on, even if it is inconsistent. It is interesting to note that this philosophy lends itself naturally to a CMR-type approach. That is, given a no-backtracking philosophy, then an approach like CMR, which privileges quick model completion over constraint fixing, is the most natural for this class of problem solving methods.

8.9.7 P&R-Marcus

Having defined a generic P&R-class, it is straightforward to characterize P&R-Marcus as an instantiation of this class - see table 8.7. In particular P&R-Marcus subscribes to the same state selection policy defined for the generic P&R-class. However, P&R-Marcus does not allow any form of backtracking outside the revision space and therefore adopts an incomplete search policy. As already said, P&R-Marcus makes use of a EMR-style control regime and therefore carries out revisions as soon as inconsistencies occur. The revision space is searched in a cost-conscious style: cheaper fixes are tried before more expensive ones. Finally it uses a focus-centred, monotonic design revision policy.

Method Class	P&R(Marcus)
Problem Solving Knowledge	Operator Cost Operator Selection Knowledge (cost-centred operator selection)
Constraint Types	Fixable Constraint Constraint
Additional Subtasks	Revise-design
State-Based Control	Extend-design + Revise-design (EMR control)
Contexts	Extend, Revise
Focus Types	Parameter (Extend) Constraint (Revise)
Focus Selection Policy	Open
Design Operator Types	Procedure (Design Extension Operator) Fix (Design Modification Operator)
Design Operator Order Policy	Select cheapest
Available Design Space	Maximal Design Model (Propose) Revision Space (Revise)
State Selection Policy	1) Design Model: Max 2) Violated Constraints: Min 3) Cost: Min
Completeness	No
Optimality	Local

Table 8.7 Synoptic description of Marcus' Propose&Revise.

8.10 CONCLUSIONS

The characterization of the Propose&Revise class of problem solving methods concludes the description of the current version of the OCML library of problem solving methods for parametric design. In the remaining of this chapter I will restate the main points concerning the design of the library while in the next chapter I will illustrate a number of applications developed by reusing and configuring the problem solving methods presented here.

8.10.1 Classifying problem solving methods

The methods examined so far can be classified into three groups - see table 8.8 - depending on the state selection policy they use.

- **Consistency-centred approaches.** These methods emphasize consistency-related aspects. They only operate on consistent nodes and attempt to construct a consistent solution path. Either local preference knowledge or complete state expansion followed by successor evaluation is used to select next best state.
- **Cost-centred approaches.** These methods (i.e. A*-design) emphasize cost-related aspects. They sacrifice quick convergence criteria (choosing the maximal design state) for cost minimization.
- **Completion-centred approaches.** The main criterion used by these methods is to minimize backtracking. Like the consistency-centred approach, they use local preference knowledge to select the next best state.

Approach	Consistency-centred	Cost-centred	Completion-centred
Methods	Gen-design-psm Propose&Backtrack Hc-design Propose&Improve	A*-design	P&R-class
State selection policy	1) Constraints: Min 2) Design Model: Max 3) Cost: Min	1) Constraints: Min 2) Cost: Min 3) Design Model: Max	1) Design Model: Max 2) Constraints: Min 3) Cost: Min

Table 8.8 Method characterization by state selection policy.

This classification provides a coarse-grained selection criterion to guide an initial method selection. For instance, if a task model does not specify too large a problem space and cost minimization is the paramount criterion, then it makes sense to attempt to configure an A*-type problem solver. Vice versa, if the problem space is large and the constraint network tightly connected, then a Propose&Revise approach may be appropriate. Of course, these generic selection criteria are only meant to provide heuristic rules for initial method selection. Nevertheless, they provide a useful starting point from which to begin the method configuration process.

8.10.2 Uniform view of problem solving methods.

An important feature of the approach taken to develop the library is the fact that this is grounded on a uniform, search-centred view of problem solving. As already pointed out, the adoption of this view is important to provide a foundation both to the overall

modelling framework and to the library. In addition, this uniform view of problem solving (which is based on a restricted number of concepts, such as states and operators) makes it easier to compare and differentiate methods. For instance, while the Propose&Backtrack and Hc-design methods superficially appear to be very different, the synoptic descriptions of these methods show that they basically follow the same philosophy and would normally exhibit similar competence. The difference between them is that Propose&Backtrack relies on local preference knowledge to select the next best state, while Hc-design relies on state expansion and evaluation to achieve the same result.

Another example of the advantages deriving from adopting a search-centred paradigm is given by the analysis of Propose&Revise presented in section 8.9. A number of researchers have published knowledge-level descriptions of Propose&Revise (Fensel, 1995b; Wielinga et al., 1995; Motta et al., 1994b), which try to clarify the role of fixes and describe the revision strategy used by P&R-Marcus. A problem which is common to all these approaches is that (in one way or another) they all get somehow 'stuck' with the details of the non-monotonic revision strategy used by Marcus et al. In contrast with these approaches, the description of P&R-Marcus given here emphasizes that i) fixes are simply a kind of design modification operators, ii) various ways of searching the revision space are possible and iii) P&R-Marcus employs a cost-centred criterion, which combines a monotonic approach to constraint fixing with a focus-centred approach. This is all we need to know for a knowledge-level analysis of P&R-Marcus: at this level the details of the (rather idiosyncratic) fix combination mechanism are irrelevant, as long as the knowledge-level description is functionally equivalent to the revision strategy used by P&R-Marcus. In particular, as I will discuss in the next chapter when illustrating the VT problem, it is possible to characterize fix combinations in different ways, either as a search strategy or as composite design modification operators.

8.10.3 Modularity (Plug and Play)

Another important feature of the approach followed here is that, because the different methods are specified out of a common set of building blocks, it is possible to construct 'hybrid' problem solvers which integrate different features from different methods. For example, it is quite easy to define a Propose-Revise-Improve problem solver, simply by i) defining a control method which combines the control regimes used by P&R-class and P&I-psm, and ii) selecting the state selection policy associated with the P&R-class. No other customization is necessary.

A control method integrating an EMR-style approach with Propose&Improve is defined in the following box.

```
(def-class emr+improve-control (decomposition-method)
  ((has-input-role :value has-design-state)
   (has-output-role :value generates-design-state)
   (has-design-state :type design-state)
   (generates-design-state :type design-state)
   (has-body
    :value
    (lambda (?psm)
      (in-environment
       ((?state . (role-value ?psm has-design-state))
        (?design-model . (the ?dm (has-design-model
                                   ?state ?dm)))
        (?constraints . (role-value ?psm has-constraints))
        (?parameters . (role-value ?psm has-parameters)))
       (if (deadend-state ?state)
         :nothing
         (if (constraint-violations ?state ?constraints)

           ;; some constraint is violated, we revise
           (achieve-generic-subtask
            ?psm
            revise-design
            has-design-state ?state)
           (if (state-complete ?state ?parameters)

           ;; solution state reached, we try improving it
             (achieve-generic-subtask
               ?psm
               generate-state-successor
               has-design-state ?state
               has-design-context :improve)

           ;; consistent, incomplete state, we try extending it
             (achieve-generic-subtask
              ?psm
              generate-state-successor
              has-design-state ?state
              has-design-context :extend))))))))

  :own-slots ((tackles-task-type  design-from-state)
              (has-generic-subtasks generate-state-successor
                                    revise-design)))
```

Thus, the approach described here attempts to combine the advantages gained from the availability of a set of complete problem solving methods (*method generality*) with those deriving from constructing the methods out of a common set of generic building blocks (*component modularity*). Method generality is required to avoid the problems encountered by Orsvärn (1996), when trying to reuse task-method libraries. Component modularity is essential in order to make problem solving methods more flexible (Poeck and Gappa, 1993).

8.10.4 Task-independent approaches

In recent years there has been renewed interest in task-independent specifications of PSMs, on the ground that the "task specific formulation of PSMs unnecessarily limits the

applicability of PSMs" (van Heijst and Anjewierden, 1996). While it is not at all obvious that task-independent approaches provide any benefits when tackling the knowledge acquisition problem, it is clear that they are appealing from a reuse point of view.

The approach to library and application development presented here enjoys a dual foundation: i) a task-specific one, defined by a task ontology and a task-centred instantiation of a search model of problem solving; and ii) a task-independent one, defined by the use of search to underpin the problem solving components in the library. Thus, the approach integrates the results associated with task-independent libraries of search algorithms (Stefik, 1995) with the task-oriented approaches to library construction, which are common to 'modern' knowledge engineering libraries (Breuker and van de Velde, 1994; Benjamins, 1993; O'Hara, 1995). Hence, the methods described here can be understood both in terms of their knowledge requirements and in terms of their search behaviour.

An advantage of the dual foundation enjoyed by the library is that it provides a natural starting point for pursuing task-independent specifications of problem solving methods. In particular, it is possible 'to strip' the methods of their task-specific aspects and characterize them exclusively in terms of their search behaviours and task-independent commitments. Such an exercise was carried out in a recent paper, written in collaboration with a number of colleagues (Fensel et al., 1997), where we provide a task-independent characterization of a Propose&Revise method. Specifically, we characterize Propose&Revise as a search algorithm which makes use of two types of state transition operators: the first type is applied to *correct* and *incomplete* states, the second one is applied to *incorrect* ones.

Such task-independent specification of Propose&Revise puts into question the 'traditional' dichotomy of *strong* vs. *weak* methods. In particular, let's consider the view expressed by McDermott (1988), who points out that "a weak method is more open with respect to control than a role-limiting method can be; a weak method does not put any limits on the nature or complexity of the task-specific control knowledge it uses". Our description of Propose&Revise shows that this view is problematic: a task-independent characterization of Propose&Revise is not more "open to control" than a task-specific one. Moreover, a (so-called) weak method, such as A*, makes precise assumptions about the existence of heuristic knowledge, which allow the method to converge to an optimal solution. Therefore it is not more "open to control" than a (so-called) strong method, such as Propose&Revise. On the contrary it actually imposes stronger requirements on the availability of domain knowledge than Propose&Revise.

Thus, all methods can in principle be specified in a task-independent (i.e. weak) or task-dependent (i.e. strong) fashion. Hence, the weak vs. strong dichotomy does not provide

much leverage with respect to method characterization. What is important is to characterize the functionalities provided and the requirements imposed by a method. By describing these requirements and functionalities in a common framework (which can be task specific or task independent) we can then compare, contrast and combine them, to build knowledge-based applications.

Chapter 9.
Application Development by Reuse

In this chapter I illustrate a number of application models constructed by applying the library of problem solving components described in the previous chapters to three application domains. These are the Sisyphus-I and KMI office allocation problems, and the VT elevator design problem. The purpose of this exercise is to validate the various technologies presented in the previous chapters - e.g. the library, the OCML modelling language, the various ontologies, the TMDA framework - by showing that they provide effective support for KBS development by reuse.

9.1. INTRODUCTION

The purpose of this application-centred chapter is to validate empirically the various epistemological and conceptual structures presented in the previous chapters, by showing that they provide excellent leverage for building applications by reuse. Specifically, my aim here is to validate the following components:

- The **overall application development framework**. By showing that it provides the appropriate distinctions required to support KBS development by reuse.
- The **parametric design task ontology**. By showing that it provides the appropriate conceptual distinctions for characterizing parametric design tasks.
- The **problem solving methods** described in chapter 8. By showing that they can be effectively used to construct parametric design applications.

In particular I will discuss three application domains: the two sample problems tackled in the first two Sisyphus initiatives (Linster, 1994; Schreiber and Birmingham, 1996) and the KMI office allocation problem. The former are well-known benchmarking problems and their inclusion is especially useful as it makes it possible to compare my approach/framework to alternative ones on an application-specific basis. The latter is a

real-world office allocation problem which our institute faced when moving to a new building[1].

The application models described here were partially coded using a simple, task-specific shell, which implements in an object-oriented style the generic tasks for parametric design included in the Gen-design-psm problem solving model. This shell is briefly described in the next section.

9.2. A SHELL FOR PARAMETRIC DESIGN PROBLEM SOLVING

Although OCML is a fully operational language, its main purpose is not to support the efficient implementation of KBS but rather the development of knowledge models. In other words, its operational capabilities are meant to support the verification and validation of knowledge models, rather than the implementation of efficient artefacts. For this reason, in parallel with the development of the OCML library of parametric design methods, I have implemented a rather basic object-oriented shell, which mirrors the structure of the generic model of parametric design problem solving described in chapter 7 and supports rapid prototyping of parametric design models.

This task-specific shell is not meant to replace an OCML application model completely, but rather to improve its efficiency by providing symbol-level mechanisms corresponding to the generic tasks used by a problem solving method. Thus, task, method and domain ontologies remain typically unchanged, unless efficiency reasons make it necessary to replace ontological components defined in OCML with symbol level analogues.

9.2.1. Integrating knowledge-level and symbol-level constructs

Like OCML, the parametric design shell is implemented in the Common Lisp Object System (Kiczales et al., 1991). As discussed in chapter 4, the OCML system provides a number of mechanisms for integrating OCML models with Lisp modules. In particular, in the implementation of the parametric design shell I have taken advantage of three mechanisms which support the integration of OCML models and Lisp programs. These mechanisms are discussed in the next three sections.

---

[1] In addition to these domains, the library has been tested on a number of real-world application domains in the context of the CEC-funded Encode project on configuration design (Copernicus Programme, Project 0149). These domains include sliding bearing design, simple mechanics problems (Hatala, 1997), initial vehicle (truck) design, design and selection of casting technology and sheet metal forming technology for manufacturing mechanical parts (Valasek & Zdrahal, 1997).

9.2.1.1. Integration through procedural attachments.

These are specified through the `:lisp-fun` keyword. This mechanism makes it possible to augment an ontological definition by attaching to it a piece of Lisp code. This is used to verify whether a statement is satisfied (in the case of relations), or to compute the value of a function or control expression (in the case of procedures and functions).

9.2.1.2. Integration through classes.

An OCML class is implemented as a CLOS object and the `def-class` primitive for defining OCML classes also allows the user to specify additional information about the CLOS object associated with an OCML class. For instance it is possible to specify additional class slots which are not part of the definition but only serve a symbol-level purpose. An example of the use of these additional class slots is given by the definition below, which augments the OCML class `parameter` with two additional, symbol-level slots which are used by the shell to cache the links between parameters and design operators and between parameters and constraints. These links support efficient operator and constraint collection mechanisms.

```
(def-class PARAMETER () ?p
  "This modifies the definition given in the parametric
   design task ontology. It adds symbol-level slots to cache
   the links between parameters and the relevant operators and
   constraints"
  ((has-value-range :type set)
  :iff-def (exists ?task (and (parametric-design ?task)
                              (has-parameters ?task ?l)
                              (member ?p ?l)))
  :lisp-slots ((relevant-operators :accessor relevant-operators
                                   :initform nil)
               (relevant-constraints :accessor relevant-constraints
                                   :initform nil))
  :lisp-class-name parameter)
```

The keyword `:lisp-class-name` is also a symbol level option, which makes it possible to specify the name of the CLOS object associated with an OCML class. This option is used, for instance, to allow the user to specify CLOS methods parametrized over an OCML class. For example, the following definition shows a CLOS method parametrized over class `parameter`. The method returns the operators applicable to the parameter which is the current focus.

```
(defmethod FILTER-OPERATORS-APPLICABLE-TO-FOCUS ((method
                                                   design-method)
                                                  (state design-state)
                                                  (focus parameter))
  (intersection (state-operators state)
                (relevant-operators focus)))
```

9.2.1.3. Integration through functional interface.

The OCML system provides a functional interface which makes it possible to create, access and modify OCML definitions from other Lisp modules. This interface includes entity-creating Lisp macros, such as `def-class`, `def-relation` and `def-function`; support for a *tell-ask interface* - e.g. `findall`, `findany`, `setofall`, `ask`, `tell`; and macros to evaluate control and functional expressions - i.e. `ocml-eval` and `procedure-eval`.

9.2.2. Symbol-level support for parametric design

The parametric design shell essentially provides two classes of efficiency-enhancing definitions: it i) replaces the task-method structure with a set of CLOS methods and ii) provides a simple caching mechanism which constructs a network of relations between parameters, constraints and operators, to speed up the selection, filtering and evaluation processes. These two types of mechanisms are discussed in the next two sections.

9.2.2.1. Replacing OCML tasks and methods with CLOS methods

As discussed in the previous chapters, the problem solving methods in the library are modelled in terms of tasks and methods. For example, in chapter 8 I showed four control methods for task `design-from-state`, each of these specifying a different approach to state-based design. When multiple methods are applicable to the same task, then the most specific one is chosen, where 'specificity' is interpreted in accordance with the rules given in chapter 5.

A similar mechanism is provided at the symbol level, where generic tasks and methods are mapped to *generic functions* in CLOS. For example, task `design-from-state` is mapped to a generic function of the same name, which is defined as follows:

```
(defgeneric design-from-state (application method state)
  (:documentation "A generic function which implements the
                   task-method structure associated with task
                   design-from-state"))
```

Having defined the appropriate generic function, individual control methods for task `design-from-state` can be defined by means of `defmethod` statements. For instance, the definition below specifies the control regime used by Propose&Backtrack.

```
(defmethod DESIGN-FROM-STATE (appl
                              (method propose&backtrack)
                              (state design-state))
  (extend-evaluate-select-control appl method state))

(defun extend-evaluate-select-control (appl method state)
  (if (violated-constraints state)
      (state-fails state)
      (if (state-feasible? state)
        (if (state-complete? state)
          (design-succeeds state)
          (extend-design appl method state))
        (state-fails state))))
```

As shown by the above definition, the CLOS method `design-from-state` takes three arguments: an application, a class of problem solving methods and a type of design state. The first argument, `application`, provides a placeholder to define application-specific method customization. The second and third argument make it possible to specialize a generic function for a class of problem solving methods. Thus, an alternative approach to state-based design, for instance the one used by HC-design, can be implemented as follows.

```
(defmethod DESIGN-FROM-STATE (appl
                              (method  HC-DESIGN-METHOD)
                              (state hc-design-state))
  (if (violated-constraints state)
    (state-fails state)
    (if (state-feasible? state)
      (if (state-complete? state)
        (design-succeeds state)
        (expand-state appl method state))
      (state-fails state))))
```

As shown by the definitions presented here, symbol-level specifications of problem solving methods can be associated with different types of design states. This is in contrast with the knowledge level models, where a state is specified in terms of the associated design model and task, and a search control record is defined for each state. For efficiency reasons I have merged the notions of design state and search control record in the parametric design shell. As a result, symbol-level design states tend to be large structures, comprising several, efficiency-related types of slots. Moreover, these structures are customized for different classes of problem solving methods.

9.2.2.2. Optimizing knowledge-level models for symbol-level efficiency

The other main difference between the structure of the shell and the model of parametric design discussed in chapter 7 concerns the use of symbol-level optimization mechanisms, such as caching. In particular, before a design application is executed, the shell creates a network of links between the main components of a design specification (i.e. parameters,

constraints and design operators). This network of links makes it possible to speed up significantly the tasks of collecting, filtering and selecting appropriate design foci and operators.

In conclusion, the shell provides a set of Lisp structures, which support the efficient execution of application models for parametric design. The set of generic functions implemented in the shell mirrors the generic problem solving model for parametric design presented in chapter 7. This approach to KBS design, based on the idea of enforcing a structural consistency between knowledge level and symbol level architectures, is often called *structure-preserving design* (Schreiber, 1992). However, given the operational nature of the knowledge level model and the prototypical nature of the shell, the latter does not entirely replace a knowledge-level model; only the problem-solving-intensive components (generic tasks).

In the rest of this chapter I will illustrate how the shell was used to benchmark various problem solving methods on three application domains.

9.3. THE SISYPHUS-I OFFICE ALLOCATION PROBLEM

9.3.1. Description of the Sisyphus-I problem

The Sisyphus-I office allocation problem (Linster, 1994) was the first of a series of test applications which have been used by the knowledge acquisition community to compare different approaches to building knowledge-based applications. The problem consists of allocating the members of the YQT research group to rooms in a new building. The problem specification consists of a four-page document, which describes the layout of the building, provides the relevant data about the fifteen members of the group and includes a protocol describing the steps taken by a (virtual) domain expert, Siggi, when solving the problem. A limitation of the problem description is that there is no requirement specification as such. The constraints and requirements applicable to the problem have to be elicited indirectly from the protocol, which is shown in table 9.1.

1) Thomas in C5-117	a) The head needs a central office, so that he is close to all members of the group. This should be a large office. b) This assignment is defined first, as the location of the office of the head restricts the possibilities of the subsequent assignments.
2) Monika and Ulrike in C5-119	a) The secretaries' office should be located close to the office of the head. Both secretaries should work together in one large office. This assignment is executed as soon as possible, as its possible choices are extremely constrained.
3) Eva in C5-116	a) The manager must have maximum access to the head and the secretariat. At the same time she should have a centrally located office. A small office will do. b) This is the earliest point where this decision can be taken.
4) Joachim in C5-115	There is no reason for the sequence of assignments of Joachim, Hans and Katharina a) The heads of large projects should be close
5) Hans in C5-114	a) The heads of large projects should be close to the head and the secretariat.
6) Katharina in C5-113	a) The heads of large projects should be close to the head and the secretariat.
7) Andy and Uwe in C5-120	a) Both smoke. To avoid conflicts with non-smokers they share an office. Neither of them is eligible for a single office. This is the first twin-room assignment, as the smoker/non- smoker conflict is a severe one.
8) Werner and Jürgen in C5-123	a) The are both implementing systems, both non-smokers. They do not work in the same project, but they work on related subjects. Members of the same project should not share offices. Sharing with members of other projects enhances synergy effects. b) There really are no criteria for the sequence of these twin room assignments.
9) Marc and Angi in C5-122	a) Marc is implementing systems, Angi isn't. This should not be a problem. Putting them together would ensure good cooperation between the RESPECT and the KRITON projects.
10) Harry and Michael in C5-121	a) They are both implementing systems. Harry develops object systems, Michael uses them. This creates synergy.

Table 9.1. Problem solving protocol for Sisyphus-I office allocation problem.

The data about the YQT members are given in a tabular format, part of which is shown in table 9.2 - see (Linster, 1994) for the complete description.

Name	Role	Project	Smoker	Hacker	Works with
Werner	researcher	RESPECT	No	Yes	Angi, Marc
Marc	researcher	KRITON	No	Yes	Angi, Werner
Andy	researcher	TUTOR	Yes	No	-
Harry	researcher	BABYLON	No	Yes	Jurgen, Thomas
Thomas	researcher	EULISP	No	No	Jurgen, Harry
Ulrike	secretary	-	No	No	Thomas, Monika, Eva
Eva	manager	-	No	No	Thomas, Ulrike, Monika
Monika	secretary	-	No	No	Thomas, Ulrike, Eva

Table 9.2. Data about YQT members

In addition, we also know that Thomas is the head of the group and that Hans, Katharina and Joachim are heads of large projects. As shown by the protocol given in table 9.1, this information is used by Siggi during the allocation process. Finally, figure 9.1 shows the layout of the YQT building - shaded offices cannot be used for the room allocation process; rooms c5-113, c5-114, c5-115 and c5-116 are single offices, the others are double ones.

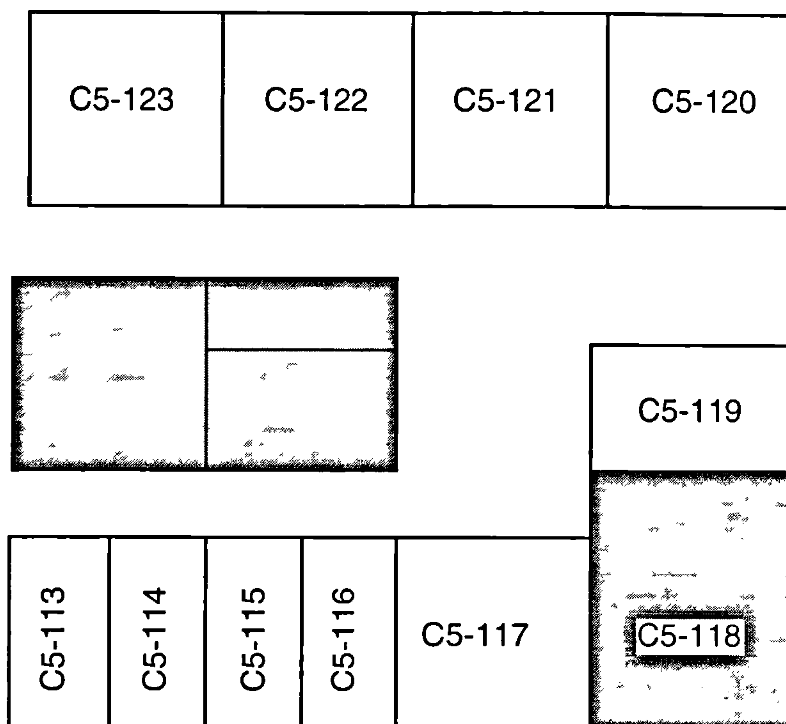


Figure 9.1. Layout of the YQT building

In what follows, I will develop an application model for the Sisyphus-I problem in accordance with the modelling framework presented in chapter 3. Thus, I will define a task model of the problem, select and configure a domain-independent problem solver, construct a domain model and then link domain and problem solving components by means of mapping and application-specific knowledge. Of course, the artificial nature of the problem and the 'behavioural' style of the protocol impose strong limits on the knowledge acquisition process. The development of a task model for a problem is in reality a dialectic, multi-stage process during which all the relevant stakeholders negotiate a common view of the problem. In this case, there is no stakeholder to negotiate with and the goal is therefore to interpret Siggi's protocol in terms of the components of the parametric design task ontology described in chapter 6.

9.3.2. Constructing a task model for the Sisyphus-I problem.

The parametric design task ontology characterizes parametric design problems in terms of parameters (with their value ranges), constraints, requirements, preferences and cost function.

9.3.2.1. Parameters

The problem is one of assigning rooms to YQT members. Thus, it can be modelled as a parametric design problem in which the set of parameters correspond to the set of YQT members. The value of a parameter is given by the corresponding room. Note that the inverse approach - i.e. characterizing rooms as parameters and YQT members as

parameter values - does not work, given that a room can have multiple values in an allocation model. Hence, the problem consists of finding a valid and complete set of assignments for the fifteen design parameters.

9.3.2.2. Value ranges

Given the homogeneous nature of the parameter set, one could just assign the same value range to all parameters: the set of all usable rooms. However, in order to be able to make effective use of heuristics such as DSR, it is useful to try and restrict the value range of a parameter as precisely as possible, so that this information can help with the focus selection process. Therefore, each parameter is assigned a value range that reflects quite closely the problem's set of requirements and constraints.

In particular, it is clear from Siggi's protocol that the problem's requirements and constraints are associated with classes of YQT members, rather than with specific individuals. Thus, I can generate the following set of value ranges for the various classes of YQT members.

Type of YQT member	Value Range	Justification
head of group	all large, central offices	"The head needs a central office...this should be a large office"
secretary	all large offices	"..secretaries should work together in one large office"
manager	a centrally located office	"..should have a centrally located office"
head-of-project	a single office	Siggi allocates them in single offices.
researcher	any office	Siggi does not indicate any kind of constraints on the type of rooms which can be given to researchers

Table 9.3. Value ranges for classes of parameters in Sisyphus-I

The value ranges shown in table 9.3. were derived by means of a two-stage process: i) identifying the constraints and requirements (indirectly) indicated by Siggi's utterances and ii) abstracting from them static restrictions on the possible values associated with a class of YQT members. The first step relies to a large extent on judgement. Given that I have no access to a real Siggi, any decision on whether a statement in a protocol is meant to be interpreted as a constraint, a requirement, or a preference is essentially a guess - although see section 9.3.2.4 for a discussion on the criteria adopted to distinguish between requirements, constraints and preferences. The second step reflects the role of value ranges in the task ontology: these are not meant to express dynamic constraints, but

merely a set of values which are feasible for a particular parameter, or class of parameters. Therefore, when specifying the value ranges for the Sisyphus-I parameters, I did not consider dynamic constraints such as "secretaries should be close to the office of the head".

9.3.2.3. Constraints and Requirements

The following set of requirements and constraints were identified from the protocol.

Requirements	Constraints
R1. head of group in large, central office; **R2.** the secretaries' office has to be close to the office of the head; **R3.** manager, head of group, and heads of project do not share; **R4.** secretaries share the same room; **R5.** manager goes into a central office.	**C1.** do not exceed room size; **C2.** smokers cannot share with non-smokers.

Table 9.4. Requirements and constraints in Sisyphus-I

As already pointed out, this characterization of requirements and constraints is somewhat subjective, given that certain requirements, e.g. "manager goes into a central office", could easily be regarded as preferences. On the other hand, given the artificial nature of the problem, to ask whether or not this is the right model is not a meaningful question. The purpose of the exercise is to show that the proposed task ontology for parametric design is adequate to capture the relevant conceptual distinctions, regardless of whether these effectively mirror those held by the original domain expert.

It is important to note that the distinction between constraints and requirements shown in table 9.4 reflect the conceptual distinction discussed in section 6.2.2. Requirements specify properties of the solution design, while constraints limit the space of solutions.

9.3.2.4. Preferences

Table 9.5 lists the preferences identified from the protocol and provides a justification for each of them. Not having access to real domain experts or clients, I used two general criteria for identifying preferences in the protocol. The first criterion was that any requirement violated by Siggi in fact denotes a preference, rather than a requirement. This criterion was used to identify P3 and P4. The second criterion identifies as a preference any requirement which can be more or less satisfied by a design model, rather than definitely satisfied or definitely violated. This is the case for preference P2. Having

"maximum access" to somebody can be interpreted as being as close as possible to that person: the closer the better. Thus, I can model this informal requirement as a preference stating that different design models should be ranked in terms of the distance between the manager and the head and secretaries. Finally, I also added P1 to complete the set of 'closeness' preferences expressed by Siggi. This preference is not redundant, nor it is inconsistent with R2 The latter specifies that secretaries should be close to the head; P1 gives higher ranking to those models which minimize this distance.

Preferences	Comments
P1. head as close as possible to secretaries;	The requirement specifies that they should be close. However, it makes sense to also add a preference so that solutions where the distance between the head and secretariat is minimized are given a higher ranking.
P2. manager as close as possible to head and secretaries;	Siggi talks about having maximum access to the head and the secretariat. I model this as a preference stating that models which minimize the distance between the manager and the head of group and secretaries are 'better'.
P3. heads of large projects as close as possible to head and secretaries;	Siggi actually talks about "heads of projects being close". However, his solution does not satisfy his own requirement. Therefore I model this as a preference, rather than as a requirement.
P4. members of the same project should not share.	Siggi states that members of the same project should not share. However, his solution does not fully enforce this requirement (Harry and Michael are allocated together despite the fact that they work in the same project). Therefore I model this as a preference,

Table 9.5. Preferences in Sisyphus-I

9.3.2.5. Cost function

As discussed in chapter 5, a cost function provides a global ordering criterion which is used to distinguish better from worse solutions. This criterion typically subsumes all preferences, although in those cases where the preferences are not mutually consistent some decision making is needed to decide how to define a global criterion out of mutually inconsistent, local criteria.

In this case the preferences are mutually consistent and therefore the only issue concerns how to harmonize them. The solution I have taken is to characterize the output of the cost function as a four-dimensional vector, $<n_1, n_2, n_3, n_4>$, where n_1 measures the distance between the room of the head of the group and that of the secretaries; n_2 the distance between the manager's room and the rooms of the head of the group and the secretaries; n_3 the distance between the heads of projects and the head of group and the secretaries; and n_4 provides a measure of the 'project synergy' afforded by a solution. The latter is computed as 1 minus the ratio between all shared assignments which maximize synergy and all shared assignments[2].

The use of a vector of partial costs to represent the global cost of a design model has two main advantages: it avoids the problem of trying to harmonize measures which are not obviously commensurable (in particular n_4 is not the same kind of measure as the others) and it makes it possible to define a cost order relation which ranks the four preferences, from the most important (P1) to the least important (P4). The rationale for this ranking is that preferences associated with senior members of an organizational hierarchy tend to have priority over those associated with junior members. More formally, the ordering relation over design models in the Sisyphus-I problem is defined as follows.

Definition 9.1. A design model in the Sisyphus-I domain, d1, with cost $<n_{11}, n_{12}, n_{13}, n_{14}>$, is cheaper than a design model d2, with cost $<n_{21}, n_{22}, n_{23}, n_{24}>$ if and only if one or more of the following conditions are satisfied:

i) $n_{11} < n_{21}$;

ii) $n_{11} = n_{21}$ and $n_{12} < n_{22}$;

iii) $n_{11} = n_{21}$ and $n_{12} = n_{22}$ and $n_{13} < n_{23}$.

iv) $n_{11} = n_{21}$ and $n_{12} = n_{22}$ and $n_{13} = n_{23}$ and $n_{14} < n_{24}$.

9.3.3. Domain modelling

According to the framework presented in chapter 3, a domain model consists of a multi-functional knowledge base. In this case, there is no pre-existing knowledge base and

---

2 If there are no shared assignments, then n_4 is 0.

therefore the domain model consists of the domain knowledge provided by the Sisyphus-I documentation. This body of knowledge covers the layout of the building and provides some information about the members of the YQT research group.

Figure 9.2 shows the main classes used to model the Sisyphus-I domain. This organization is quite straightforward and reflects the information provided with the domain documentation.

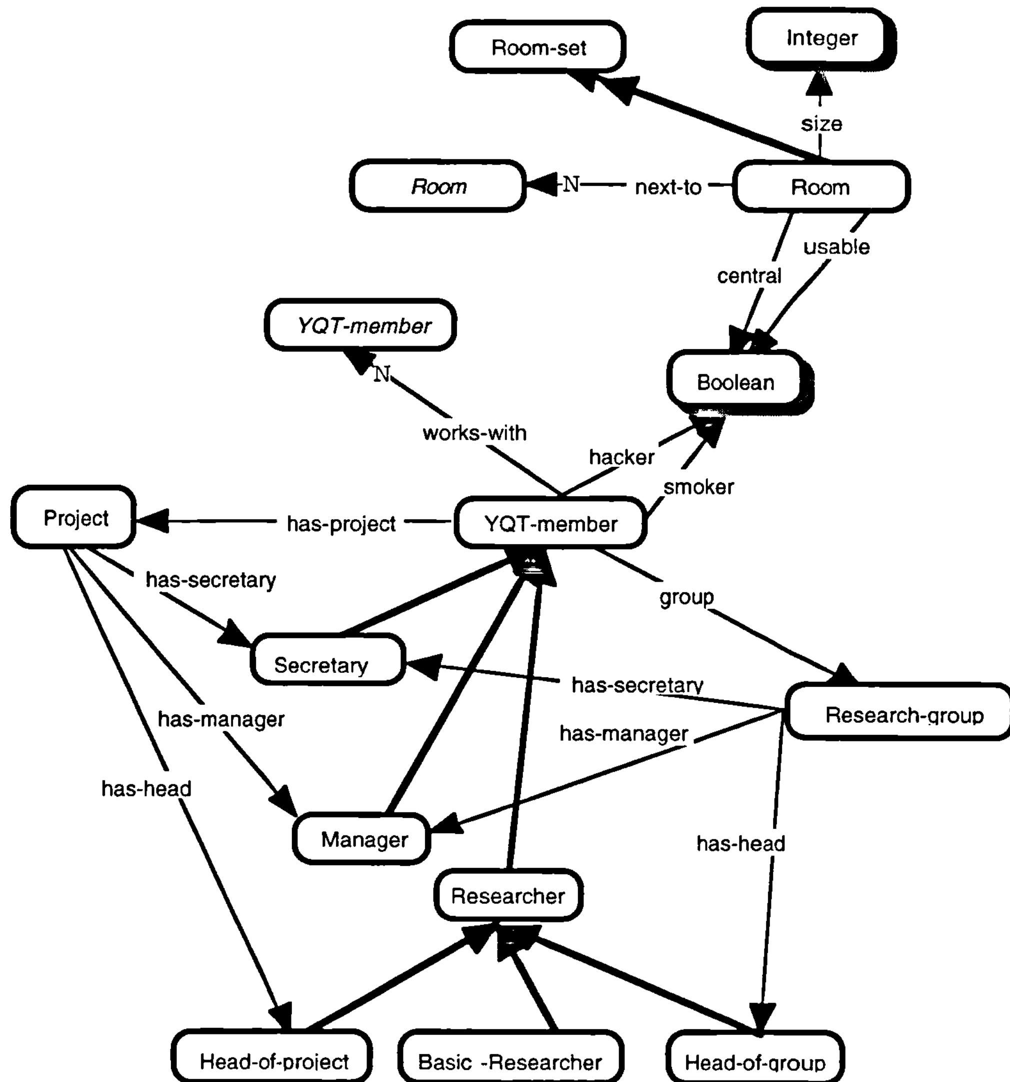


Figure 9.2. Main classes in Sisyphus-I domain

Figure 9.3. shows the main relations and functions defined in the Sisyphus-I domain. Much of the action in the Sisyphus-I application consists of reasoning about the relative distance between rooms in the YQT building. Therefore the domain model comprises a number of functions and relations which can be used to reason about the layout. In particular, the function `compute-distance` computes the distance between two rooms; the function `compute-distance*` generalizes the notion of distance to list of rooms: it computes the distance between a room and a list of rooms; relation `in-room` associates a

YQT member to the office he or she occupies; relation `closest-to` is satisfied by a triple `<room, room-set, room-list>` if `room` is the closest room in `room-set` to the rooms in `room-list`. Finally, relation `closer-than` is satisfied by a triple <room$_1$, room$_2$, room-list> if and only if room$_1$ is closer than room$_2$ to the rooms listed in `room-list`.

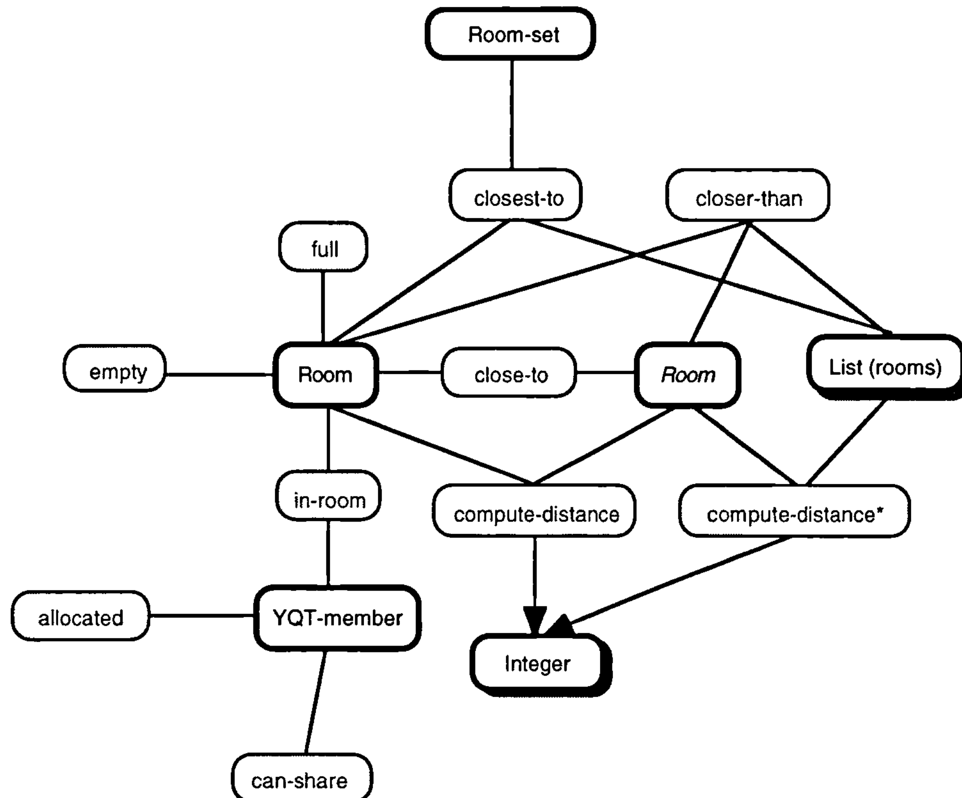


Figure 9.3. Main relations and functions in Sisyphus-I domain

9.3.4. From task to problem solving: specifying design operators

As shown in figure 7.1, moving from a task to a method dimension involves defining design operators and operator preferences from the set of constraints, requirements and preferences defining a task specification. In this section I discuss the design operators defined for the Sisyphus-I application. These operators specify part of the application-configuration component of the application model. Because the parameters in the Sisyphus-I application are characterized in terms of their type (e.g. head of group, manager), design operators are associated with classes, rather than with individual parameters. Thus, I will structure the discussion in terms of the types of YQT members.

9.3.4.1. Multiple design extension operators

The operators used for the Sisyphus-I application are instances of a class of design operators, which is called `multiple-design-extension-operator`. The instances of this class are design extension operators which can be called repeatedly, each time

generating a different assignment. That is, they collapse several operators into one. This class of operators is defined as follows.

```
(def-class MULTIPLE-DESIGN-EXTENSION-OPERATOR (design-extension-
operator multiple-operator)
  "The body of a  multiple design extension operator is a binary
   function which takes as argument an unbound  parameter, ?p, and a
   set of values, ?values, and produces as a result a new value, ?z,
   which is taken to specify the value of ?p in the next design
   state.  The values of all the other design parameters should not
   be affected by the application of the operator.  A multiple
   design operator can be invoked repeatedly to spawn alternative
   successor states"
  ((has-body :type multiple-design-extension-operator-body))
  :lisp-class-name multiple-design-extension-operator)

(def-class multiple-design-extension-operator-body
  (lambda-expression) ?x
  "A multiple design extension operator body is a binary function
   which takes a parameter, say ?p, and a list of values, say
   ?values, and produces a result, ?z, which belongs to the value
   range of ?p but is not a member of the list ?values.  ?z is taken
   as the new value of ?p in the successor design state"
  :no-op (:constraint (and (nth-domain ?x 1 parameter)
                           (nth-domain ?x 2 ?y)
                           (=> (= ?z (call ?x ?p ?values))
                               (and (has-value-range ?p ?range)
                                    (forall ?v
                                            (=> (member ?v ?values)
                                                (member ?v ?range)))
                                    (member ?z ?range)
                                    (not (member ?z ?values)))))))
```

The main feature of a multiple design extension operator is that its body takes two arguments: a parameter and a list of values. The latter can be used to pass the operator the list of values which have been tried and failed. This mechanism makes it possible to backtrack to an already tried operator and generate a different value.

I can now discuss the specific operators defined in the Sisyphus-I application.

9.3.4.2. Head of group

The main requirement on the head of the group is that this should have a large, central office. We also know that his office should be close to that of the secretaries. Thus, I define two operators which deal with the allocation of the head of group in the two cases in which some secretary has or has not been allocated. The first one, `assign-head-of-group1`, deals with the case in which a secretary has been allocated.

```
(def-instance assign-head-of-group1 yqt-design-operator
  "If one or more secretaries have been allocated, put
   the head in a large, central room, as close as possible
   to the secretaries"
  ((applicable-to-parameters '(map meta-reference
                              (setofall ?x (head-of-group ?x))))
   (has-body (lambda (?x ?rooms)
               (if (and (secretary ?y)
                        (in-room ?y ?sec-room))
                 (the ?r1
                   (and (room ?r1 size 2 central yes usable yes)
                        (not (member ?r1 ?rooms))
                        (empty ?r1)
                        (not (exists
                              ?r2
                              (and (room ?r2 size 2 central yes
                                                   usable yes)
                                   (<> ?r2 ?r1)
                                   (not (member ?r2 ?rooms))
                                   (empty ?r2)
                                   (closer-than
                                    ?r2 ?r1 (?sec-room)))))))))))
```

This operator selects an empty, central room, of size 2, which is the closest to that of the secretaries. If no secretary has been allocated, then a different operator is applicable, which is shown in the following box.

```
(def-instance assign-head-of-group2 yqt-design-operator
  "If there is no secretary allocated, then put the head
   in a large, central room"
  ((applicable-to-parameters '(map meta-reference
                              (setofall ?x (head-of-group ?x))))
   (has-body (lambda (?x ?rooms)
               (if (not (exists ?y
                                (and (secretary ?y)
                                     (in-room ?y ?sec-room))))
                 (the ?room
                   (and (room ?room size 2 central yes usable yes)
                        (empty ?room)
                        (not (member ?room ?rooms))))))))
```

This operator simply returns a room suitable for the head of group.

9.3.4.3. Secretaries

Secretaries should be allocated together in a large room, which should be close to the head of the group. This requirement is operationalized by means of two design operators. The first one deals with the case in which the head of the group has already been allocated and there are at least two more secretaries to allocate; the second one is used in those cases in which the previous one is either not applicable or fails. These operators are shown in the following box. Note the use of the `design-operator-order` statement to indicate that the first operator has to be tried before the second one.

```
(def-instance assign-secretaries1 yqt-design-operator
  "If at least 2 secretaries need to be allocated, we want
   to put them in a large room, as close as possible to the
   head of the group. The assumption here is that there is only
   one head of group"
  ((assumption (< (cardinality (setofall ?y (head-of-group ?y)))
                  2))
   (applicable-to-parameters '(map meta-reference
                              (setofall ?x (secretary ?x))))
   (has-body (lambda (?x ?rooms)
               (if (and (head-of-group ?y)
                        (in-room ?y ?head-room)
                        (> (cardinality
                             (setofall
                              ?sec
                              (and (secretary ?sec)
                                   (not (in-room
                                         ?sec ?any)))))
                           1))
                 (the ?r1
                   (and (room ?r1 size 2 usable yes)
                        (not (member ?r1 ?rooms))
                        (empty ?r1)
                        (not (exists
                               ?r2
                               (and (room ?r2 size 2 usable yes)
                                    (not (member ?r2 ?rooms))
                                    (<> ?r2 ?r1)
                                    (empty ?r2)
                                    (closer-than
                                     ?r2 ?r1 (?head-room)))))))))))
```

```
(def-instance  assign-secretaries2 yqt-design-operator
  "This operator simply puts a secretary in a room partially
   occupied by another secretary"
  ((applicable-to-parameters '(map meta-reference
                              (setofall ?x
                                        (secretary ?x))))
   (has-body (lambda (?x ?rooms)
               (if (and (secretary ?y)
                        (in-room ?y ?room)
                        (not (full ?room))
                        (not (member ?room ?rooms)))
                 ?room))))

(tell (design-operator-order assign-secretaries2
                             assign-secretaries1))
```

9.3.4.4. Manager

The manager should have her own central office, as close as possible to the head and the secretaries. The operator below operationalizes this requirement.

```
(def-instance  assign-manager-central yqt-design-operator
  "Manager to have her own office as close as possible to
   secs and head of group.  Small office ok for manager"
  ((applicable-to-parameters '(map meta-reference
                                (setofall ?x (manager ?x))))
   (has-body (lambda (?x ?rooms)
               (if (and (not
                         (exists
                          ?sec
                          (and (secretary ?sec)
                               (not (in-room
                                     ?sec ?some)))))
                        (head-of-group ?head)
                        (in-room ?head ?head-room))
                 (the ?r1
                   (and (= ?s-rooms
                           (setofall ?sec-room
                                     (and (secretary ?y)
                                          (in-room ?y ?sec-room))))
                        (room ?r1 usable yes central yes)
                        (not (member ?r1 ?rooms))
                        (empty ?r1)
                        (not (exists ?r2
                                     (and (room
                                           ?r2
                                           usable yes
                                           central yes)
                                          (not (member ?r2 ?rooms))
                                          (<> ?r2 ?r1)
                                          (empty ?r2)
                                          (closer-than
                                           ?r2 ?r1
                                           (cons ?head-room
                                                 ?s-rooms)))))))))))
```

9.3.4.5. Head of project

Heads of projects should have their own office, as close as possible to the head of the group and the secretaries. The following operator implements this requirement.

```
(def-instance  assign-head-of-project yqt-design-operator
  "Heads of projects should have their own office as close
   as possible to the head of the group and the secretaries.
   This office does not have to be central"
  ((applicable-to-parameters '(map meta-reference
                              (setofall ?x (head-of-project ?x))))
   (has-body (lambda (?x ?rooms)
              (if (and (not
                         (exists ?sec
                                 (and (secretary ?sec)
                                      (not (in-room
                                            ?sec ?some)))))
                        (head-of-group ?head)
                        (in-room ?head ?head-room))
                (the ?rl
                  (and (= ?s-rooms
                          (setofall ?sec-room
                                    (and (secretary ?y)
                                         (in-room ?y ?sec-room))))
                        (room ?rl usable yes)
                        (not (member ?rl ?rooms))
                        (empty ?rl)
                        (not (exists ?r2
                                    (and (room ?r2 usable yes)
                                         (not (member ?r2 ?rooms))
                                         (<> ?r2 ?rl)
                                         (empty ?r2)
                                         (closer-than
                                          ?r2 ?rl
                                          (cons ?head-room
                                                ?s-rooms))))))))))))
```

9.3.4.6. *Researchers*

No requirements apply to researchers. Siggi prefers to maximize project synergy. These conditions are met by the combination of the next two operators and the operator ranking specified by the `design-operator-order` statement.

```
(def-instance  assign-researcher1 yqt-design-operator
  "maximize synergy between projects"
  ((applicable-to-parameters '(map meta-reference
                              (setofall ?x
                                        (basic-researcher ?x))))
   (has-body (lambda (?x ?rooms)
              (the ?a-room
                (and
                 (room ?a-room)
                 (not (member ?a-room ?rooms))
                 (in-room ?a ?a-room)
                 (can-share ?a)
                 (has-project ?a ?p1)
                 (has-project (domain-reference ?x) ?p2)
                 (<> ?p1 ?p2)
                 (not (full ?a-room)))))))
```

```
(def-instance assign-researcher2 yqt-design-operator
  "any empty room will do"
  ((applicable-to-parameters '(map meta-reference
                                (setofall ?x
                                          (basic-researcher ?x))))
   (has-body (lambda (?x ?rooms)
               (the ?rl
                 (and (room ?rl usable yes)
                      (not (member ?rl ?rooms))
                      (empty ?rl))))))))
```

```
(tell (design-operator-order assign-researcher1
                              assign-researcher2))
```

9.3.5. Modelling constraints and requirements

As discussed in section 7.3.2, the distinction between requirements and constraints is enforced only when constructing the task specification. During problem solving they are treated in the same way, as constructs specifying design prescriptions.

In order to improve the efficiency of the constraint checking process, when building an application model I make use of a more 'operational' representation of constraints, which links these to the relevant parameters and parametrizes the constraint expression with respect to an assignment, rather than a design model. Thus, at each cycle of the design process only the constraints affected by the changes to the previous model need to be checked. This 'parameter-oriented' class of constraints is called `parametric-constraint`. Its definition is shown in the box below.

```
(def-class PARAMETRIC-CONSTRAINT (constraint)
  "A parametric constraint provides a more 'problem solving
   oriented' definition of constraints.  Both the precondition and
   the expression are binary kappa expressions, which take as
   argument a parameter and a value (i.e. an assignment).  The idea
   here is that this assignment-oriented approach makes possible to
   limit the number of constraints to be tested at each stage of the
   design process to those relevant to the modified assignments"
  ((applicable-to-parameters
    :type function-expression
    :documentation "An expression which returns the set
                    of parameters to which this constraint
                    is applicable")
   (has-expression
    :cardinality 1
    :type legal-parametric-constraint-expression)
   (has-precondition
    :default-value '(kappa (?p ?d)
                     (true))
    :type kappa-expression
    :documentation "This expression can be used to determine whether
                    a constraint makes sense for a given parameter
                    assignment"))

   ;;symbol-level definitions
  :lisp-class-name parametric-constraint
  :Lisp-slots ((effective-parameter-list
                :initform nil
                :accessor effective-parameter-list)))
```

As shown by the above definition, a parametric constraint is described in terms of three slots: `applicable-to-parameters`, `has-expression`, and `has-precondition`. The latter specifies an expression which checks whether a parameter is relevant to the current assignment. Slot `has-expression` specifies the constraint expression associated with the constraint instance. This expression is parametrized in terms of a parameter assignment. Finally, slot `applicable-to-parameters` defines an expression indicating the set of parameters to which the constraint is applicable.

In order to clarify the purpose of the slots defining a parametric constraint I show below the definition of the constraint `smoker-constraint` from the Sisyphus-I application model. Slot `applicable-to-parameters` specifies that this constraint is applicable to all the parameters denoting YQT members who can share; slot `has-precondition` specifies that this constraint has to be checked for all shared allocations and slot `has-expression` checks that the assignment passed as input satisfies the constraint.

```
(def-domain-instance smoker-constraint yqt-constraint
  ((applicable-to-parameters '(map meta-reference
                               (setofall
                                ?x (and (yqt-member ?x)
                                        (can-share ?x)))))
   (has-precondition (kappa (?p ?r)
                            (> (length
                                (setofall
                                 ?x (in-room ?x ?r)))
                               1)))
   (has-expression (kappa (?p ?r)
                          (not (exists (?x ?y)
                                       (and
                                        (in-room ?x ?r)
                                        (in-room ?y ?r)
                                        (<> ?x ?y)
                                        (yqt-member ?x smoker ?v)
                                        (yqt-member ?y smoker ?u)
                                        (<> ?v ?u)))))))
```

9.3.6. Mapping Knowledge

In the previous sections I have discussed the representations of design operators and constraints in the Sisyphus-I model. These constructs are not part of the Sisyphus-I domain model but are instead part of the application configuration component. In other words these constructs specify application-specific knowledge and are modelled in a method-oriented style.

Hence, there are only two mappings which need to be performed to integrate a domain-independent design problem solver with the Sisyphus-I domain model. The first one associates the concept of parameter at the problem solving level with the concept of YQT member at the domain level. The second one maps the relation used by the problem solver to represent the current design model, called `current-design-model`, to the relation used to represent office assignments at the domain level, `in-room`.

The mapping between parameters and YQT members is specified by means of the macro `def-upward-class-mapping` - see definition below. This macro takes two classes, one defined at the domain level, the other at the problem solving level, and associates each instance of the domain class to a newly-created instance of the method class. The functions `meta-reference` and `domain-reference` can then be used to retrieve the corresponding method-level instance from a domain-level one and vice versa. This form of mapping is useful when only a simple association between problem solving and domain concepts is needed.

```
(def-class yqt-parameter (parameter)
   "The class of YQT specific parameters")

(def-upward-class-mapping yqt-member yqt-parameter)
```

The mapping between the relation `current-design-model` at the problem solving level and `in-room` at the domain level is more complicated, because it is necessary to create a bi-directional mapping to ensure the consistency between the view at the problem solving level and that at the domain level. This mapping can be realized by means of the set of definitions shown in the box below. The first one provides a way to lift at the problem solving level the set of `in-room` statements asserted in the domain knowledge base. The other definitions are used to reflect down to the domain level assertions and deletions of type `current-design-model`.

```
(def-relation-mapping current-design-model :up
   ((current-design-model ?dm)
    if
    (= ?dm (setofall (?p . ?v)
                     (and (in-room ?X ?v)
                          (maps-to ?p ?x))))))

(def-relation-mapping current-design-model (:down :add)
   (lambda (?x)
     (loop for ?pair in ?x
           do
           (if (maps-to (first ?pair) ?z)
             (tell (in-room ?z (rest ?pair)))))))

(def-relation-mapping current-design-model (:down :remove)
   (lambda (?x)
     (unassert (in-room ?x ?y))))
```

With this discussion of the relevant mapping knowledge I have concluded the description of the main aspects of an application model for the Sisyphus-I problem. In the next section I will discuss the results obtained by trying out different problem solving methods on this application domain.

9.3.7. Solving the Sisyphus-I office allocation problem

9.3.7.1. Solving by Gen-design-psm

The first method I applied to solving the Sisyphus-I problem was a configuration of the Gen-design-psm method described in chapters 7 and 8. This method enhances a simple depth-first control regime by the use of focus selection and design operator knowledge, to decide which state transition to perform. In particular, the method was configured as follows.

- **Focus selection**. This task was carried out by employing a DSR strategy. This analyses i) the value ranges associated with each parameter and ii) the current design model, to select, at each stage, the most constrained parameter. Although the method ontology associated with Gen-design-psm also provides a mechanism for stating application-specific, focus selection knowledge (by means of the appropriate `design-focus-order` statements), no such knowledge was required.

- **Operator selection.** Operator preference knowledge was specified by means of the `design-operator-order` statements shown in the earlier sections.

Thus, hardly any method configuration was necessary to apply Gen-design-psm to Sisyphus-I.

Table 9.6 shows the result of applying Gen-design-psm to Sisyphus-I. The column labelled 'efficiency' gives an indication of the performance of the method. The score is computed as the ratio between the size of the minimal search space required to solve the task and that of the search space effectively navigated to reach a solution. Methods which go 'straight to the solution', without performing any backtracking, get a 100% efficiency score.

Solution	Cost	Efficiency
THOMAS_D in C5-117 ULRIKE_U in C5-119 MONIKA_X in C5-119 EVA_I in C5-116 KATHARINA_N in C5-115 HANS_W in C5-114 JOACHIM_I in C5-113 WERNER_L in C5-120 MARC_M in C5-120 ANDY_L in C5-121 HARRY_C in C5-122 JURGEN_L in C5-122 ANGY_W in C5-123 MICHAEL_T in C5-123 UWE_T in C5-121	(4 10 106 0)	78%

Table 9.6. Siggi's solution by Gen-design-psm

The solution shown in table 9.6 is isomorphic to the one generated by Siggi. This is interesting because it shows that Siggi basically applies a simple DSR heuristic to decide the order of the assignments. That is, the problem solver does not require any additional knowledge about focus ordering, in order to emulate Siggi. Moreover, the resulting performance is very good. The design system only backtracks four times, in each case because of assignments which violate the smokers-related constraint. This backtracking

could be easily avoided by modifying the operator `assign-researcher1`, to ensure that it takes into account the smokers-related constraint when generating an assignment.

However, the solution given in table 9.6 is not optimal; an alternative, optimal solution is shown in table 9.7.

Solution	Cost	Efficiency
THOMAS_D in C5-119	(1 25 187 0)	78%
ULRIKE_U in C5-120		
MONIKA_X in C5-120		
EVA_I in C5-116		
KATHARINA_N in C5-115		
HANS_W in C5-114		
JOACHIM_I in C5-113		
WERNER_L in C5-117		
MARC_M in C5-117		
ANDY_L in C5-121		
HARRY_C in C5-122		
JURGEN_L in C5-122		
ANGY_W in C5-123		
MICHAEL_T in C5-123		
UWE_T in C5-121		

Table 9.7. Optimal solution by Gen-design-psm

While Gen-design-psm can in principle find the optimal solution shown in table 9.7, it is not necessarily able to generate it. The reason is shown in figure 9.4: operator `assign-head-of-group2` can equally generate any of two possible design extensions: <thomas_d, c5-117> or <thomas_d, c5-119>. If the second one is chosen then the problem solver will reach the optimal solution. Otherwise it will reach the same solution as Siggi's.

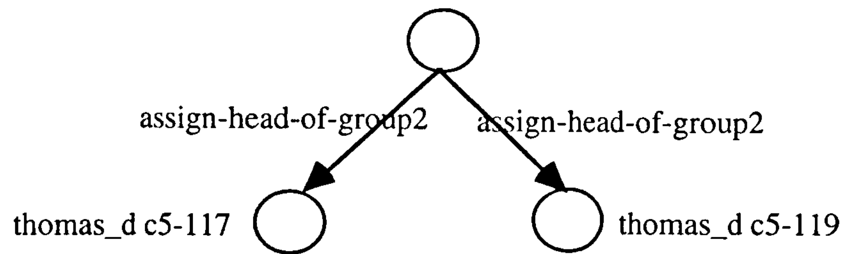


Figure 9.4. Alternative state transitions by means of `assign-head-of-group2`

9.3.7.2. Solving by HC-design

Another method I tried on the Sisyphus-I domain was HC-design. Like Gen-design-psm this method was able to generate both solutions shown in tables 9.6 and 9.7. Like Gen-design-psm, it could not guarantee to derive one rather than the other. The reason is shown in figure 9.5: the two possible expansions of the root state have the same cost. Therefore, HC-design does not have enough information to select the optimal solution path.

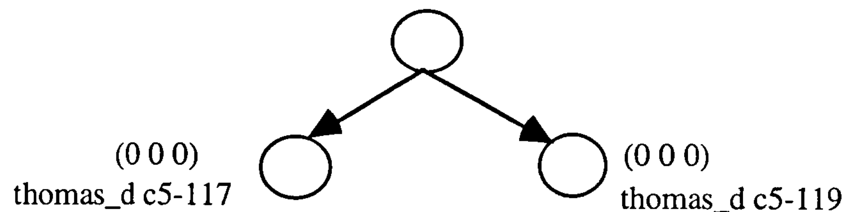


Figure 9.5. State space after expansion of root node in Sisyphus-I by HC-design

Thus, HC-design and Gen-design-psm exhibit the same competence. However, Gen-design-psm is much more efficient than HC-design, which has only a 10% efficiency.

9.3.7.3. Solving by A-design*

9.3.7.3.1. Defining the heuristic function

In order to configure A*-design for the Sisyphus-I domain, I defined a heuristic function which, for a given state - say s_i, computes the estimated cost required for reaching a solution state from s_i. This heuristic function is conceptually quite simple. The estimated cost is a quadruple, $<hc_{i1}, hc_{i2}, hc_{i3}, hc_{i4}>$, which is defined in terms of the following rules.

i) $hc_{i1} = 0$ if both head of group and secretaries are allocated in s_i;

ii) $hc_{i1} = h_1 (d_i)$ if head of group or some secretary is not allocated in s_i;

iii) $hc_{i2} = 0$ if head of group, secretaries and manager are allocated in s_i;

iv) $hc_{i2} = h_2 (d_i)$ if head of group, manager or some secretary are not allocated in s_i;

v) $hc_{i3} = 0$ if head of group, heads of projects and secretaries are allocated in s_i;

vi) $hc_{i3} = h_3 (d_i)$ if head of group, some head of project or some secretary are not allocated in s_i;

vii) $hc_{i4} = 0$ if all researchers have been allocated in s_i;

viii) $hc_{i4} = h_4 (d_i)$ if some researchers have not been allocated in s_i.

The h_i functions take as input a design model and produce as output an estimate of the cost of the relevant allocations. Because it is important to get these estimates as accurate as possible, the details of these functions can be quite complicated and therefore I won't describe them in detail. The basic idea is that the h_i functions try to estimate the minimal distance that can be achieved for the relevant group of YQT members in the current problem solving state. For instance, if neither the head of the group nor the secretaries are allocated, then the best we can hope for is to have them in adjacent rooms, i.e. the best estimate of the relevant distance is 1.

9.3.7.3.2. Performance of A*-design

Naturally, A*-design is able to find the optimal solution and its efficiency is comparable to that of HC-design (about 10%). Of course, its competence is better than HC-design, given that A*-design guarantees to find the optimal solution.

9.3.7.4. Summing up

The Sisyphus-I application is quite a simple one. As shown by the above results, a straightforward configuration of Gen-design-psm is able to solve the problem very efficiently and can also find a solution which is better than the one generated by the domain expert. No domain-specific heuristic are needed, a DSR-style focus selection strategy combined with the implicit dependency network defined by the preconditions of the design operators suffices. As a result, there is no need for hill-climbing-type design: the extra search effort required to make locally optimal decisions is unnecessary. The dependencies between classes of parameters enforced by the task specification strongly constrain the design process and therefore very little degrees of freedom exist when evaluating alternative design steps. The simplicity of the problem also means that there is no need for non-uniform approaches to design, e.g. Propose&Revise or Propose&Improve.

Finally, if an optimal solution is required, then the problem space is small enough to be tackled by means of A*-type search.

9.3.7.5. Comparison with other solutions to the Sisyphus-I problem

The special issue of the International Journal of Human-Computer-Studies edited by Marc Linster (Linster, 1994) includes a number of solutions to the Sisyphus-I problem. Here I

will compare the solutions presented in this section to some of those discussed in the special issue.

The solution provided by the DIDS researchers (Balkany et el., 1994) is closest to the approach followed here. They characterize the problem as a design one and describe it in terms of constraints, functions, part and preferences. Each person is modelled as a function, while rooms are modelled as parts. The problem solver exploits a constraint satisfaction engine, which, at each stage, selects the locally optimal assignment. Preferences are statements of the form "a is close to b". No ranking of preferences is imposed. Unfortunately, Balkany et el. do not provide precise figures on the performance of their problem solver, other than generic measures of complexity in the worst case. As a result, it is difficult to provide a detailed comparison between the two approaches. In general, I would expect their problem solver to exhibit a performance similar to the one shown by Gen-design-psm.

The solution presented by Schreiber follows a KADS approach and builds an abstract interpretation model from the problem solving behaviour exhibited by Siggi. This model characterizes Siggi's task as a *plan assembly* one. Problem solving begins by formulating a *plan*, i.e. specifying the order in which the various *components* (i.e. YQT members) should be assigned *resources* (i.e. rooms). Once a plan has been produced, then the various *plan elements* are designed, by carrying out the allocation of resources to components. The model tries to emulate Siggi's behaviour by allocating shared resources to groups of components, rather than performing each component allocation separately. These shared allocations are carried out by generating all possible consistent groupings for sets of components and then filtering those which satisfy the problem constraints. Multiple solutions are then achieved by merging together all mutually consistent groupings.

There are a number of differences between my application model and that developed by Schreiber, which concern the underlying approach to reuse and the specific method used to solve the problem.

Reuse. My approach to reuse is based on specifying reusable components which are quite rich in nature and can be directly instantiated to produce operational models. As a result these components are very usable. The approach used by Schreiber (which is essentially the KADS approach) formulates library of components at a higher level of abstraction. My experience with these libraries (Motta et al., 1994a; 1994b; 1996) is that while they provide strong support for the early stages of knowledge acquisition, they only afford limited support for detailed conceptual modelling. It is easy to see the problem when reading Schreiber's account of the model building process. Much work is needed to flesh out and instantiate the model. In contrast, my library of

problem solving methods provides 'ready-made' components which can be directly applied to a domain.

Problem solving. The method used by Schreiber appears to be very inefficient in general. Generating all possible groupings is subject to combinatorial explosion. The method works for the Sisyphus-I problem only because of the limited complexity of this domain.

The solution by Motta et al. also follows a KADS-based approach, although applies a different formulation, which is based on the VITAL methodology (Shadbolt et al., 1993). In particular, it uses a library of Generalized Directive Models (O'Hara, 1995) to drive the model building process. The comments made above concerning the KADS approach to reuse apply to this solution as well. Moreover, while the problem solving method used by Motta et al. is more efficient that the one used by Schreiber, it is much less generically applicable. Essentially, both approaches try to build generic models which emulate Siggi's problem solving. In my view this strategy reflects the weak support provided by the libraries used by the VITAL and KADS groups.

In contrast with these approaches, the solutions presented in the earlier sections are not concerned with 'emulating Siggi'. The process I used was in fact the opposite one. I characterized the problem in terms of the parametric design task ontology, configured a method from the library, and then executed it. The results showed that Siggi's behaviour could be emulated by a simple DSR heuristic. Thus, I did not try to abstract from Siggi's behaviour to build a model; I used a pre-existing model to give semantics to Siggi's problem solving.

9.4. THE KMI OFFICE ALLOCATION PROBLEM

In August 1997 the Knowledge Media Institute moved to a new building and therefore I had the opportunity to try out the parametric design technology on a real-world office allocation problem. The resulting application models are discussed in this section.

9.4.1. Domain model

The layout of the new KMI building is shown in figure 9.6. It comprises eighteen single offices (7 m^{2}), one double office (room-9-10) and six triple offices (21 m^{2}). There is also a meeting room and two other rooms which are not available to the allocation process (these are shaded in the figure).

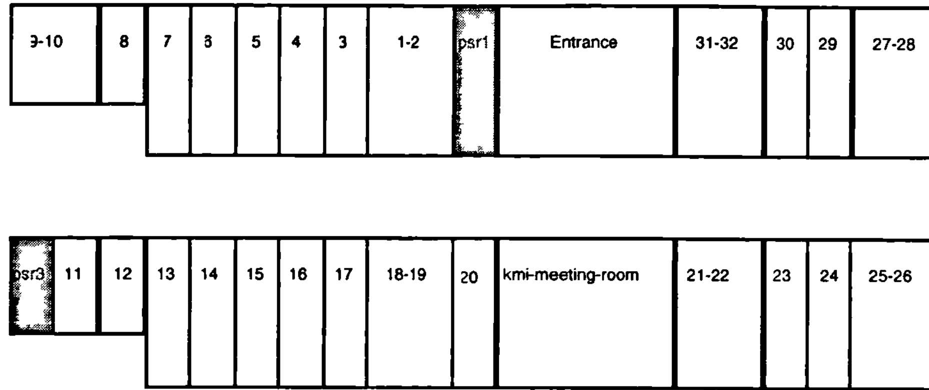


Figure 9.6. Layout of the KMI building

The personnel considered in this problem comprises thirty-three people, each of which belongs to one of fourteen different categories of staff members. These categories are shown in figure 9.7. The ovals in the figure represent classes of KMI members and all links denote `subclass-of` relations.

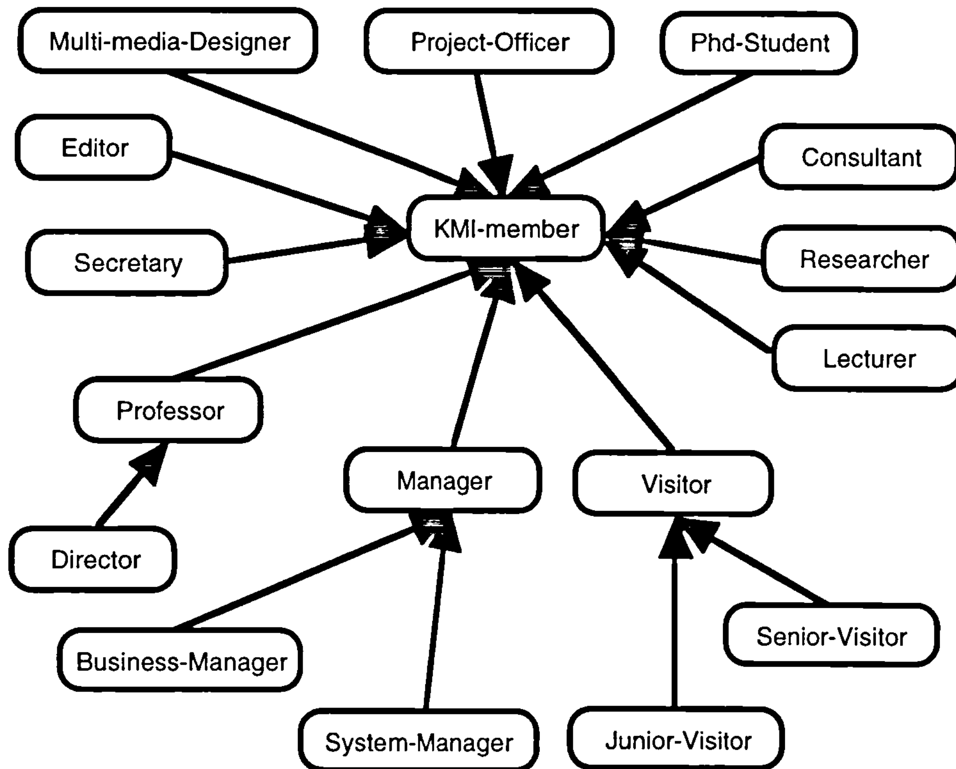


Figure 9.7. Taxonomy of KMI members.

Each member of the group is involved in a number of projects/activities. Hence, clusters of 'affinity groups' can be defined, which are important for the allocation process. Figure 9.8 shows a subset of the overall cluster of affinity groups for the KMI domain[3].

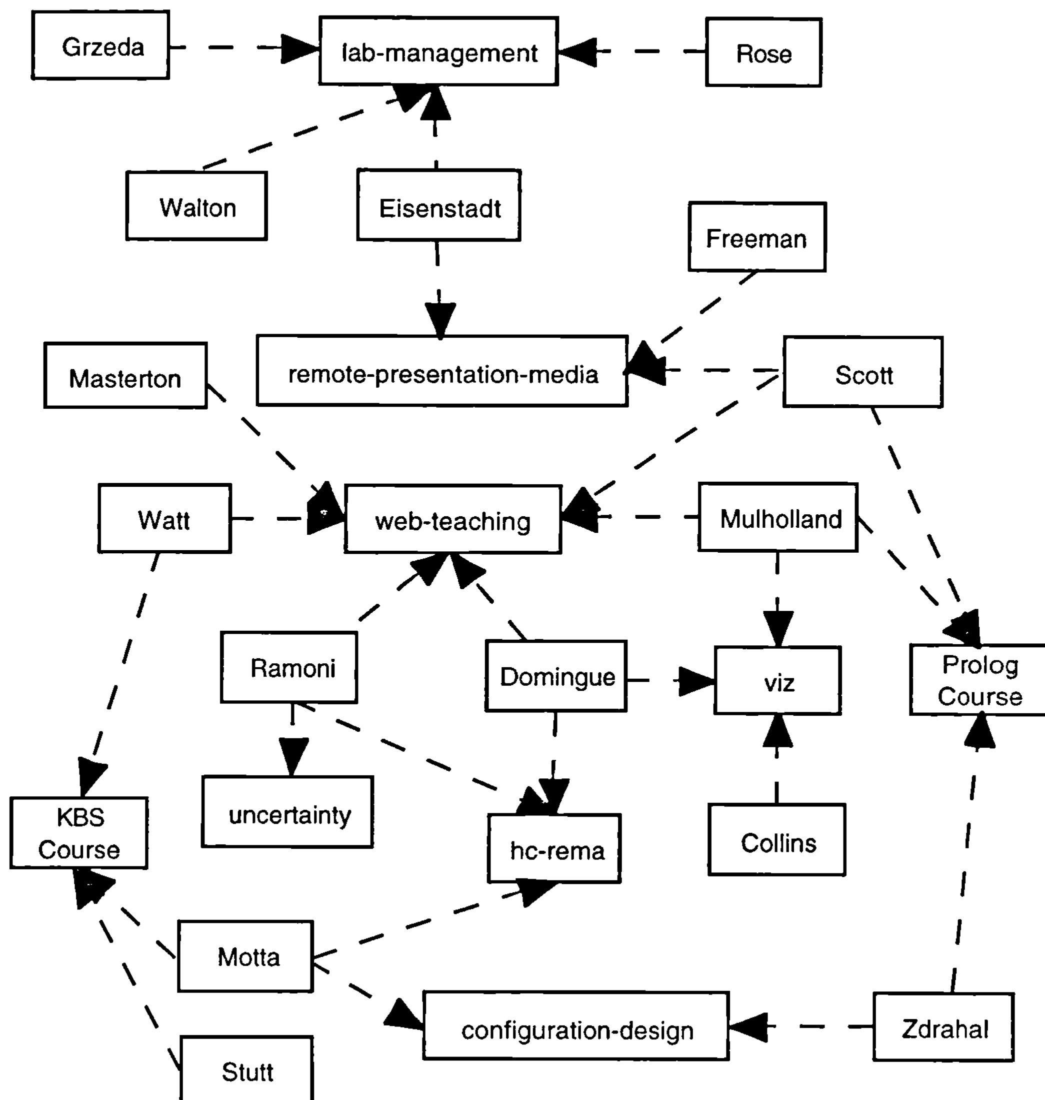


Figure 9.8. A subset of the network of affinity groups.

The function shown below uses the information about the clusters of activities in KMI to derive the collaborators of each member of the department.

[3] Of course the figure is only meant to give an idea of the various multidimensional clusters of 'affinity groups' which exist in KMI and to emphasize the difference with the simple 1-to-1 mapping of projects in Sisyphus-I. It is not meant to be a faithful representation of the KMI research web, either now or in the past.

```
(def-function collaborators (?x)
  :body (remove ?x
              (in-environment
               ((?activities . (setofall ?a (activity ?x ?a))))
               (setofall ?y
                         (and (activity ?y ?aa)
                              (member ?aa ?activities))))))
```

9.4.2. Task model

9.4.2.1. Parameters and value ranges

As in the case of Sisyphus-I, the KMI office allocation problem can be modelled as a parametric design problem in which the set of parameters maps to the set of KMI members. The value range of a parameter is specified in terms of the offices in which the corresponding member can be allocated. These are shown in table 9.8.

Type of KMI member	**Value Range**
Professor	all large offices (there are six of these)
Secretary	all large offices next to the entrance (room-31-32)
Research Fellows, Lecturers and Business Manager	all single offices
All others	any office

Table 9.8. Value ranges for classes of parameters in KMI application.

9.4.2.2. Requirements and constraints

The following set of requirements and constraints were elicited from the KMI director.

Requirements	Constraints
R1. Professors are entitled to a very large office (21 m^2). **R2.** The secretaries' office has to be next to the entrance. **R3.** The director's office should be close to the secretaries'. **R4.** Professors, research fellows, lecturers and the business manager should not share **R4.** Secretaries should not share with non-secretaries.	**C1.** do not exceed room size;

Table 9.9. Requirements and constraints in the KMI office allocation task.

As in the Sisyphus-I problem, the requirements and constraints express simple design prescriptions about sharing and space entitlement. A total ban on smoking means that there is no need to separate smokers from non-smokers.

9.4.2.3. Preferences and cost function

The requirement specification for the KMI problem includes a number of preferences which are similar in nature to the ones which make up the Sisyphus-I specification (e.g. director as close as possible to secretaries, media people as close as possible to media preparation room, etc.). These preferences concern a subset of the parameter set and are static in nature; therefore they can be treated locally during the allocation process.

Much more interesting is a global preference criterion which specifies that "people working on similar activities should be near each other". This can be rephrased as stating that the distance between each KMI member and all his collaborators should be minimized. A cost function based on this criterion can be formalized in OCML as follows.

```
(def-function compute-kmi-cost  (?dm)
  :body (/ (apply + (map (lambda (?assignment)
                           (compute-kmi-assignment-cost
                            (first ?assignment)
                            (rest ?assignment)
                            ?dm))
                         ?dm))
           (length ?dm)))

(def-function compute-kmi-assignment-cost (?p ?v ?d)
  :body (in-environment
         ((?kmi-member . (domain-reference ?p))
          (?l . (filter (collaborators ?kmi-member)
                        '(kappa (?c)
                          (has-value
                           (meta-reference ?c)
                           ?vv ?d))))))
         (if (null ?l)
           0
           (/ (compute-distance*
               ?v (findall ?vc
                           (and (member ?x ?l)
                                (has-value
                                 (meta-reference ?x)
                                 ?vc ?d))))
              (length ?l)))))
```

The function `compute-kmi-cost` defines the cost of a design model as the average cost of each assignment. Function `compute-kmi-assignment-cost` defines the cost of each assignment, say related to parameter `?p`, as the average distance between the KMI member denoted by `?p` and his/her collaborators.

The important feature of this cost function is that it specifies a global, non-monotonic criterion which may or may not increase with the size of the design model. This means that it is difficult to apply to the KMI domain a method such as HC-design, which requires a converging monotonic criterion.

9.4.3. Design operators

Six design operators were defined, which follow quite straightforwardly from the design prescriptions and preferences. Two of them deal with the allocation of secretaries, one allocates the director as close as possible to the secretaries, another one ensures that professors get a large room. The other two deal with the remaining classes of KMI members. The first one, `assign-shared`, deals with those KMI members who can share; the second one, `assign-single`, deals with those KMI members who go into a single office. Both these operators try to minimize the distance between a member and his/her collaborators, in accordance with the global cost function. Of course, given the non-monotonic nature of the global cost function there is no guarantee that a sequence of locally optimal assignments will lead to a global optimum. If no collaborators of the currently selected KMI member have been allocated, then the operators `assign-shared`

and `assign-single` use a simple heuristic, which consists of allocating the KMI member in question as far as possible from any current cluster of allocations. Thus, these operators are heuristic in nature and therefore different from all other operators discussed in earlier sections, which are derived from constraints, requirements, value ranges, and preferences - see section 7.3.3 for an analysis of the generic types of operators.

9.4.4. Solving the KMI office allocation problem

9.4.4.1. Solving by Gen-design-psm

Solving the problem by means of Gen-design-psm was very simple and efficient. It turns out that the space of solutions is very dense and the problem can be solved with hardly any search. The set of operators defined for the application could find a solution without ever violating a constraint, thus achieving 100% efficiency. Unfortunately, the application of Gen-design-psm to KMI does not produce particularly good solutions. The dynamic nature of the cost function means that a strategy based on locally optimal steps easily ends up generating a not-so-good solution. To see whether this problem could be addressed heuristically I tried out different focus selection strategies. The results of these different configurations of Gen-design-psm can be seen in table 9.10.

Focus Selection Strategy	Cost
widest range + max allocated collaborators + loneliest	5.19
DSR + min allocated collaborators + min collaborators + loneliest	5.46
widest range + min allocated collaborators + loneliest	5.6
widest range + max allocated collaborators + max collaborators + loneliest	5.7
DSR + max allocated collaborators + max collaborators + loneliest	6.2
widest range + min allocated collaborators + min collaborators + loneliest	7.2

Table 9.10. Performance of different configurations of Gen-design-psm.

The heuristic strategies used were as follows.

- **Allocated collaborators (max or min)**. Selects the parameter with the maximum (or minimum) number of allocated collaborators.
- **Collaborators (max or min)**. Selects the parameter with the maximum (or minimum) number of collaborators.
- **DSR**. Selects the parameter with the minimum number of possible values.

- **Widest range**. Selects the parameter with the maximum number of possible values.

As shown by table 9.10, in this problem it is not crucial to use a DSR strategy to obtain a good solution. The reason is that the problem is heavily under-constrained. This situation is in contrast with the Sisyphus-I problem, where we saw that a DSR strategy could emulate the problem solving behaviour of the domain expert.

Another clear indication of the table is that best results are obtained when consistent criteria (max or min) are chosen for the value-range-centred and for the collaborators-centred strategies. If a DSR strategy is used, then it is better to minimize the collaborator-centred selection criterion. If a widest range approach is used, then it is better to maximize the collaborator-centred selection criterion. This rule seems to be violated by the third row of the table which shows a good performance of an application model integrating a widest range policy with a strategy which minimizes the number of allocated collaborators. However, this result may be caused by the fact that a widest range strategy reduces the discriminating power of the allocated collaborators heuristic.

In any case, none of these models is particularly good. Hence, there is a need for a different approach which can produce better solutions. This approach is discussed in the next section.

9.4.4.2. Solving the KMI office allocation problem by means of Propose&Improve

We have seen that Gen-design-psm does not lead to very good solutions, despite the fact that it is able to make local optimal steps at each parameter assignment cycle. In order to improve the quality of the solution some other method must be used, which is able to reason about global, rather than local optimality. Of course, one such method is A*, which can guarantee an optimal solution, as long as the appropriate heuristic function is defined. The problem with the KMI office allocation problem is that it is very difficult to formulate an admissible and efficient heuristic function. Again, the non-monotonic nature of the global cost function implies that it is very difficult to predict the cost of the final solution from an incomplete model. As a result, only weak heuristic functions can be defined, which fail to prune the search space significantly.

An alternative to A* is HC-design. This method however suffers from the same problem as Gen-design-psm: it is only able to make local optimal steps. Moreover, it exhibits a worst-case complexity of $O(k^n)$, where k is the size of the average value range and n is the number of parameters (Stefik, 1995). These figures are much more expensive than those exhibited by the various configurations of Gen-design-psm shown in table 9.10. The relevant focus selection heuristics can be compiled very efficiently, so that the effective complexity of the various problem solvers is $O(n)$.

Methods such as Propose&Revise are designed to deal efficiently with severely constrained problems and therefore are of little or no use here.

The approach I took to improve on the solution achieved by Gen-design-psm is based on the idea of *global hill climbing*. A global hill climbing process takes as input a complete model and repeatedly performs a hill climbing algorithm until the required goal is achieved. In the KMI domain this approach can be implemented by repeatedly improving a solution by modifying part of the model until no more improvements are possible. Thus, we can avoid the problems caused by the non-monotonicity of the global cost function.

This approach was realized by configuring a Propose&Improve method for the KMI domain. The basic idea was to use a suitably configured Propose module for reaching as good a solution as possible and then performing a hill-climbing strategy which, at each cycle, chooses the best possible improvement to the current solution model. To keep the search space manageable, the only improvements I consider are those which can be achieved by swapping the values of two parameters in the current model.

Implementing such a configuration of the Propose&Improve method was rather trivial, as it consisted of defining only four CLOS methods. In particular, I defined a foci collection method which retrieved all possible pairs of parameters which could be tried during the state expansion process and an operator collection method which simply created a design modification operator which performed the selected swap.

The box below shows a synoptic trace of the behaviour of a Propose&Improve problem solver for the KMI domain. As shown by the trace, the method quickly (four moves) achieves a 20% improvement on the quality of the solution and then slowly converges to a method-optimal solution.

```
Propose phase completed...cost is now 5.194492544492546

Starting Improve phase......

Swapping KMI-PARAMETER-SUMNER and KMI-PARAMETER-MOTTA...cost is now
4.74485329485395

Swapping KMI-PARAMETER-THOMAS and KMI-PARAMETER-REILLY...cost is now
4.335942760942761

Swapping KMI-PARAMETER-PROJECT-OFFICER-1 and KMI-PARAMETER-
COLLINS...cost is now 4.200300625300626

Swapping KMI-PARAMETER-LEWIS and KMI-PARAMETER-THOMAS...cost is now
4.091534391534391

Swapping KMI-PARAMETER-PROJECT-OFFICER-2 and KMI-PARAMETER-
MULHOLLAND...cost is now 4.014862914862916

Swapping KMI-PARAMETER-PRICE and KMI-PARAMETER-STOREY...cost is now
3.970670995670996

Swapping KMI-PARAMETER-DOMINGUE and KMI-PARAMETER-MASTERTON...cost is
now 3.9514790764790764

Swapping KMI-PARAMETER-BUCKINGHAM-SHUM and KMI-PARAMETER-SUMNER...cost
is now 3.9393578643578 65

Swapping KMI-PARAMETER-WHALLEY and KMI-PARAMETER-MOTTA...cost is now
3.9385882635882634

Swapping KMI-PARAMETER-HAWKRIDGE and KMI-PARAMETER-FREEMAN...cost is
now 3.9329605579605587

No more improvements are possible
```

Figure 9.9. Trace of global hill climbing process from optimal propose model.

The above trace reflects the behaviour of a fairly 'brutal' hill-climbing approach which at each cycle generates and examines all possible swap pairs. As a result, its efficiency is very low (0.3%). Moreover, its complexity grows exponentially with the number of parameters.

An alternative configuration of the Propose&Improve method for the KMI domain limits the size of the search space by only considering swap pairs which include one of the five most expensive parameters in the current design model. This heuristic drastically prunes the search space, as it considers only 25% of the nodes which can be possibly examined at each stage of the hill climbing process. This kind of search is often called *beam search* (Stefik, 1995).

```
Propose phase completed...cost is now 5.194492544492546

Starting Improve phase......

Swapping KMI-PARAMETER-SUMNER and KMI-PARAMETER-MOTTA...cost is now
4.744853294853295

Swapping KMI-PARAMETER-THOMAS and KMI-PARAMETER-REILLY...cost is now
4.335942760942761

Swapping KMI-PARAMETER-MULHOLLAND and KMI-PARAMETER-ZDRAHAL...cost is
now 4.314297739297739

Swapping KMI-PARAMETER-LEWIS and KMI-PARAMETER-THOMAS...cost is now
4.205531505531506

Swapping KMI-PARAMETER-SUMNER and KMI-PARAMETER-ZDRAHAL...cost is now
4.202837902837904

Swapping KMI-PARAMETER-BUCKINGHAM-SHUM and KMI-PARAMETER-
ZDRAHAL...cost is now 4.196103896103896

Swapping KMI-PARAMETER-WHALLEY and KMI-PARAMETER-MOTTA...cost is now
4.195334295334295
```

Figure 9.10. Trace from optimal propose model with beam width = 5

Figure 9.10 shows the trace obtained by carrying out a global hill climbing process narrowed down to the five most expensive parameters. The starting model for the improve phase is the same as the one used for the trace in figure 9.9. Figure 9.10 shows that the restriction only marginally affects the quality of the solution. However its efficiency is about an order of magnitude better (2%).

An important aspect of the Propose&Improve method is that by imposing a converging criterion on the design process it is able to achieve drastic improvements on the quality of the design models. This means that when using a Propose&Improve method it is not necessary to achieve very good solutions at the end of the propose phase. Any reasonable solution suffices. This point can be clearly illustrated by showing the trace obtained by applying the restricted hill climbing strategy to the worst model in table 9.10. This trace, which is shown in figure 9.11, indicates that the algorithm is eventually able to achieve a solution which is even better (although by a very small margin) than the one obtained by applying the unrestricted hill climbing process to the best solution which could be obtained at the end of the propose phase.

```
Propose phase completed...cost is now 7.219420394420394

Starting Improve phase......

Swapping KMI-PARAMETER-PROJECT-OFFICER-1 and KMI-PARAMETER-
REILLY..cost is now 6.299518999519

Swapping KMI-PARAMETER-FOSTER and KMI-PARAMETER-GRZEDA..cost is now
5.921452621452621

Swapping KMI-PARAMETER-SUMNER and KMI-PARAMETER-SCOTT..cost is now
5.588335738335739

Swapping KMI-PARAMETER-PRICE and KMI-PARAMETER-COLLINS..cost is now
5.38505291005291

Swapping KMI-PARAMETER-QUICK and KMI-PARAMETER-FREEMAN..cost is now
5.199194324194324

Swapping KMI-PARAMETER-TAYLOR-M-J and KMI-PARAMETER-HAWKRIDGE..cost is
now 4.647318422318422

Swapping KMI-PARAMETER-WHALLEY and KMI-PARAMETER-SCOTT..cost is now
4.619035594035594

Swapping KMI-PARAMETER-WATT and KMI-PARAMETER-WHALLEY..cost is now
4.469805194805194

Swapping KMI-PARAMETER-MASTERTON and KMI-PARAMETER-WRIGHT..cost is now
4.248544973544974

Swapping KMI-PARAMETER-DOMINGUE and KMI-PARAMETER-WHALLEY..cost is now
4.223220298220298

Swapping KMI-PARAMETER-BUCKINGHAM-SHUM and KMI-PARAMETER-WHALLEY..cost
is now 4.0441438191438195

Swapping KMI-PARAMETER-ZDRAHAL and KMI-PARAMETER-WHALLEY..cost is now
4.022787397787398

Swapping KMI-PARAMETER-STUTT and KMI-PARAMETER-WHALLEY..cost is now
3.9882275132275122

Swapping KMI-PARAMETER-ZDRAHAL and KMI-PARAMETER-WHALLEY..cost is now
3.9121572871572874

Swapping KMI-PARAMETER-DOMINGUE and KMI-PARAMETER-WHALLEY..cost is now
3.8963564213564217
```

Figure 9.11. Trace from worst propose model with beam width = 5

The resulting allocation model is shown in figure 9.12.

9-10	8	7	6	5	4	3	1-2	psr1	Entrance	31-32	30	29	27-28
Wright Storey	Ramoni	Motta	Stutt	Mulholland	Watt	Lewis	Project-Off-2 Project-Off-1 Price			Walton Rose	Grzeda	Thomas	Vincent

psr3	11	12	13	14	15	16	17	18-19	20	kmi-meeting-room	21-22	23	24	25-26
	Sumner	B-Shum	Zdrahal	Domingue	Whalley	Scott	Foster	Collins Masterton Freeman	Hawkridge		Eisenstadt	Linney	Taylor	Reilly Quick Valentine

Figure 9.12. Allocation model with cost 3.89...

To recap, in this section I have discussed the KMI office allocation problem and shown that the dynamic nature of its cost function implies that the problem cannot be satisfactorily solved by means of methods which use greedy strategies. Therefore I tackled the problem by means of suitable configurations of Propose&Improve and showed that these can obtain very good solutions.

9.5. THE VT ELEVATOR DESIGN PROBLEM

The third application domain I am going to discuss in this chapter is the VT elevator design problem which was chosen as the common data set in the Sisyphus-II benchmarking initiative (Schreiber and Birmingham, 1996). The problem consists of configuring an elevator in accordance with a set of requirements. These are specified in terms of an initial assignment of values to a subset of the parameter set. The VT application knowledge is informally described in a document (Yost & Rothenfluh, 1996),[4] which describes the various parts of an elevator, the applicable constraints and the problem solving knowledge required to solve the problem by means of a Propose&Revise approach. In addition to the Yost document, the data set for the Sisyphus-II initiative also includes a formal ontology, which characterizes (or attempts to characterize) the application knowledge in a method-independent style (Gruber et al., 1996).

In the earlier sections on the Sisyphus-I and KMI office allocation problems, I showed how these application domains could be solved by applying a reuse-centred process and I discussed the relative pros and cons of alternative application models, which make use of different problem solving methods. Here, I will take a different approach and I will carry out a 'rational reconstruction' of the Propose&Revise-type solution to the VT design problem, originally developed by Marcus et al. (1988). The purpose of this exercise is to

---

[4] In the rest of this chapter I will refer to this document simply as 'the Yost document'.

show the analytical leverage provided by the modelling framework and library components presented in this thesis: these can be used to re-engineer an existing application, to clarify the nature of the embedded knowledge and to explain its problem solving behaviour in terms of a search-centred problem solving model.[5]

9.5.1. A critique of the VT domain model provided as part of the Sisyphus-II data set.

The description of the VT application given in (Yost and Rothenfluh, 1996) can be termed 'method-oriented', as it is based on the Propose&Revise model developed by Marcus et al. (1988). Specifically, the document describes the following types of knowledge.

- **Knowledge about design components**. This includes a description of the model components relevant to the VT domain (i.e. the design parameters), their value ranges and the formulas used to compute their values.
- **Knowledge about constraints and fixes**. This section lists the constraints applicable to the problem and the fixes which can be applied to restore consistency when one or more constraints have been violated.
- **Knowledge about the problem solving process**. This part of the document informally describes the Propose&Revise architecture developed by Marcus et al. - i.e. the P&R-Marcus problem solving method described in section 8.9.7.
- **Knowledge about the structure of a VT requirement specification**. This consists of an initial assignment of values to a subset of the parameter set.
- **Knowledge about preferences**. This knowledge is expressed implicitly by means of an association between costs and fixes. Given this association of costs to fixes it is possible to define a global cost function which characterizes the cost of a VT design model, say ?dm, as a ten-place vector, $<c_{10},......c_{1}>$, where each c_i represents the number of fix applications of cost i required to generate ?dm (Motta et al., 1996).

The other resource provided as part of the Sisyphus-II data set consisted of a set of formal ontologies developed by Gruber et al. (1996)[6]. These ontologies attempt to move away from the method-centred description provided by the Yost document (VT-Yost), to

---

[5] Those readers who are interested in the application of other problem solving methods to the VT domain are referred to the following papers: (Zdrahal and Motta, 1995; 1996; Motta and Zdrahal, 1996; Runkel at al., 1996).

[6] In the rest of this chapter I will refer to this formal model of the VT domain knowledge as 'VT-Onto'.

produce a model of the problem (a *task model* in my terminology) which can be expressed independently of any problem solving method. In order to achieve this result Gruber et al. reformulated the knowledge provided by Yost, according to the mapping schema shown in figure 9.13.

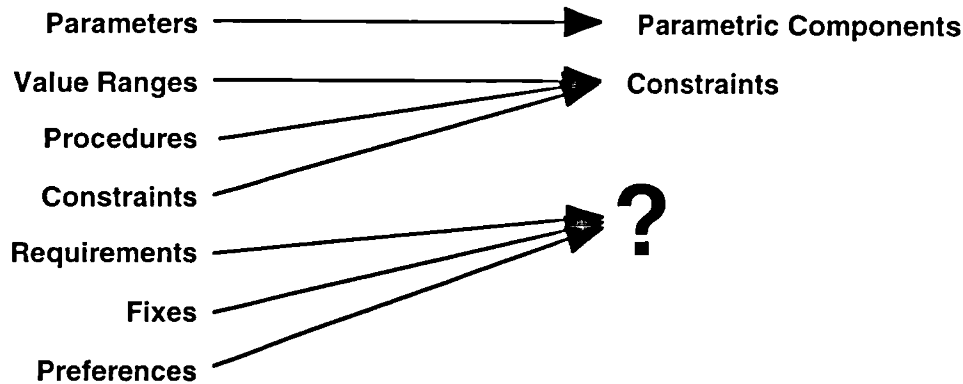


Figure 9.13. From VT-Yost to VT-Onto

If we analyse this mapping in terms of the framework discussed in chapter 7, some problems emerge quite clearly.

9.5.1.1. Mapping procedures to constraints.

Procedures are design extension operators. As discussed in section 7.3.3., design operators can be generated from value ranges, requirements and constraints. Which means that in some cases it is correct to map design operators to constraints. However, design operators can also be derived from preferences or from heuristic knowledge. Therefore the transformation shown in figure 9.13 is only correct if no operators of type C and D (see section 7.3.3) exist in the VT application. Unfortunately this is not the case. From the Yost document it is easy to see that the values of twenty-two parameters are computed by means of mechanisms which express either heuristic knowledge or preferences. Seventeen of these twenty-two procedures are trivial: they express default, initial or preferred values for parameters, which can be later modified by means of fixes. The remaining five are much more interesting: they describe mechanisms for deriving the values of some parameters from already bound ones. The five procedures associated with these parameters (Safety-Beam-Model, Car-Buffer-Blocking-Height, Cwt-Bottom-Reference, Car-Return-Left and Cwt-To-Platform-Rear) are erroneously modelled as constraints in VT-Onto.

It is interesting to note that the Yost document implicitly indicates that the procedures associated with two of these parameters (Car-Buffer-Blocking-Height and Cwt-Bottom-Reference) do not express constraints. In particular the document states: "*When fixing constraint violations, some of the values the initial* CAR BUFFER BLOCKING HEIGHT

is computed from may change. If this happens, the blocking height should NOT *be recomputed from the changed values*" (Yost & Rothenfluh, 1996 - page 22). The rationale for this statement is that the car buffer blocking height is computed initially by means of a procedure which does not define a functional constraint. A similar statement is made at page 24 in relation to the counterweight bottom reference.

In practice, mapping procedures to constraints overconstrains the problem. In particular the solution to the test case given in the Yost document does not satisfy the additional set of constraints introduced by the mapping.

9.5.1.2. Lack of knowledge about preferred or optimal solutions

Another problem with the VT-Onto model is that it cannot alone provide the basis for building application models for the VT problem. The main reason for this problem is that an essential aspect of task knowledge for parametric design problems, knowledge about preferences, is missing from the set of VT ontologies. This absence cannot be attributed to the lack of such knowledge in the Yost document, which clearly discusses a fix-based cost assignment mechanism. Hence, it is possible that this knowledge was 'thrown away', together with the knowledge about fixes, in order to produce a method-independent problem specification. Of course, this approach would be incorrect given that knowledge about preferences is related to a task rather than a method.

Finally, it is also possible that the designers of the VT ontologies did not aim to provide a complete, formalized problem specification, but simply a reusable domain model. This is of course a possibility, although in the paper published in the IJHCS special issue (Gruber et al., 1996) the authors explicitly say that their aim was to provide "a common specification of a problem".

To summarize, while the set of VT ontologies provide a common resource to develop reusable domain models, the VT domain model suffers from two problems: it overconstrains the space of solution designs and it does not provide a complete task specification. These two problems can clearly be seen by analysing the mapping from VT-Yost to VT-Onto in terms of the model discussed in section 7.3.3. Hence, in the next section I will illustrate an alternative task model for the VT problem.

9.5.2. Constructing a task model for the VT problem.

9.5.2.1. Parameters

The VT problem is characterized in terms of a number of components which are organized in a part-subpart hierarchy. Because each component is described in terms of a number of parameters it is quite natural to model the problem as a parametric design task. Such an approach is in a sense 'reductive', given that it fails to account for the component-centred structure of an elevator design. A more 'principled' approach would

require an explicit modelling of the structure of the VT components and the space of component assemblies. Such a component-centred approach informs the VT ontologies, which include modelling primitives to structure hierarchies of components and components assemblies. In general, this type of knowledge is very important in configuration design problems, because it helps to reduce the combinatorics of parameters. On the other hand, the space of solutions to the VT problem is homogeneous and very little component-related problem solving knowledge is provided in the Yost document. Therefore both the solution I developed with a number of colleagues in the context of the Sisyphus initiative (Motta et al., 1996) and the one I will discuss here do away with component hierarchies and characterize the VT problem as one of assigning values to 230 design parameters, in accordance with the relevant design prescriptions. Each parameter is associated to a value range which describes either a discrete set of possible values (e.g. motor model) or a continuous interval (e.g. the position of the counterweight).

9.5.2.2. Requirements

A VT requirement specification is formulated as an initial assignment of values to 26 parameters, which defines the input to a particular configuration problem. However, not all these assignments are, strictly speaking, requirements, given that it is possible to have solutions which violate some of them.

Specifically, the modifiable parameters in a requirement specification are divided into two categories: *major* and *minor contract specifications*. Fixes which modify a major contract specification have a very high cost (10); fixes which modify a minor contract specification have cost 6. Thus, I have modelled these modifiable requirements as preference knowledge and the remaining ones as 'proper' requirements.

9.5.2.3. Constraints

The Yost document lists 50 constraints. To these it is necessary to add the procedures which express functional constraints - there are 176 of them. Therefore the total number of constraints included in the VT problem is 226. Of course this number depends on the way constraints are modelled. In some cases it is easier to split a single constraint definition described in the Yost document into a number of different constraints. This means that the number of constraints varies with different formalizations. My model includes 239 constraints, while the VT domain theory lists 366. However, this number also comprises many specifications of value ranges which I represent separately in my task model. If I also count value ranges, then my model comprises 438 constraints. The discrepancy between these numbers is due to several factors: not all value ranges are modelled in the VT-Onto domain theory; some erroneous constraints included in the VT-Onto model are not included in mine and in many cases a single constraint described in

VT-Yost is represented by means of two or more constraints, either in my model, or in VT-Onto, or in both.

9.5.2.4. Preferences and cost function

The Yost document specifies preferences indirectly, by associating a cost measure to each fix, and none to procedures. Hence, we can assume that the ideal design solution is one which does not involve any revision step during the design process. In other words, preferences are expressed procedurally, by associating costs to design operators, rather than declaratively.

Of course, it could be possible to abstract a set of declarative preferences from the procedural model described in Yost by analysing the mapping between fixes and parameters and writing the appropriate expressions indicating that, for parameter x, certain values (i.e. those which can be obtained by means of a fix application) are less preferred than those obtained by applying a procedure. However, this exercise would be laborious, tedious and quite unnecessary: the method ontology discussed in chapter 7 provides adequate constructs for representing the VT procedural cost model directly. In particular, as indicated earlier, one possible formalization of the VT cost model, which is consistent with the procedural description given in the Yost document, formalizes the cost of a design model as a ten-place vector, $<c_{10},......c_1>$, where each c_i represents the number of applications associated with a fix of cost i (Motta et al., 1996). This cost model can be formalized in OCML as follows, using the notion of state transition discussed in chapter 7.

```
(def-function compute-vt-cost  (?dm)
  "This function computes the cost of a design model, ?dm, in the VT
   application.  If ?dm has no predecessor, then the cost is a
   10-place vector with all zeros.  This specifies  the cost of an
   empty model.  Otherwise the cost is computed by adding the cost
   associated with the operator to the cost of the predecessor."
  :body (in-environment
        ((?state .  (the ?state
                         (and (design-state ?state)
                              (has-design-model ?state ?dm)))))
        (if (state-transition ?pred ?op ?state)
          (add-vt-operator-cost
           (compute-vt-cost (the ?pred-dm
                              (has-design-model
                               ?pred ?pred-dm)))
           (the ?c (has-cost ?op ?c)))
          '(0 0 0 0 0 0 0 0 0 0))))
```

```
(def-function add-vt-operator-cost (?vector ?op-cost)
  "Adding the cost of an operator to a VT cost vector consists of
   increasing by 1 the field in the vector corresponding to the cost
   of the operator."
  :body (in-environment
        ((?v-pos . (- 10 ?op-cost)))
        (if (= ?op-cost 0)
         ?vector
         (append (sublist ?vector ?v-pos)
                 (list-of (+ 1 (elt   ?v-pos  ?vector)))
                 (nthrest ?vector (+ 1 ?v-pos))))))
```

9.5.3. Applying Propose&Revise to the VT domain

9.5.3.1. Modelling the Propose step

9.5.3.1.1. A classification of VT procedures

When discussing the application models developed for the KMI and Sisyphus-I domains, I illustrated an application development process which followed the completion of a task model with the specification of the design operators relevant to the application. Because of the artificial nature of the VT problem and the method-oriented problem specification given in the Yost document, the inverse process is necessary here: the construction of a task specification from a problem solving oriented description of the VT application.

Thus, the step of defining the operators needed for the VT model is rather simple. For each parameter in the VT domain the Yost document specifies a procedure to compute its values. These procedures can be modelled as design extension operators. Specifically, my application model specifies 202 procedures, which are grouped as follows.

- *Operators for handling invariant parameters.* The VT domain specification comprises five parameters which are not part of the requirement specification, nor

can they be modified during the configuration design process. An operator of type B[7] takes care of associating them with their values.

- *Operators for handling modifiable parameters in the requirement specification.* The requirement specification for a VT design comprises twenty-six parameters. Seven of these can be modified during the revision process. This means that these initial assignments should be characterized as preferences rather than requirements. The resulting operator is therefore of type C.
- *Operators for handling prescriptive parameters in the requirement specification.* This type B operator initializes the values of the parameters given in the requirement specification, which cannot be modified during the design process. There are 19 of these.
- *Operators for initializing default values of fixable parameters.* These type C operators initialize the values of parameters which can be modified during the revision process. The expressions associated with these operators do not depend on the values of any parameter. There are 16 of these operators.
- *Operators expressing preferred or heuristic problem solving knowledge.* There are six procedures in the Yost document which superficially look like expressing functional constraints but on closer inspection indicate preferred or heuristic values for some parameter. These procedures are associated with the following parameters: Cwt-To-Platform-Rear; Safety-Beam-Model; Cwt-Bottom-Reference; Car-Return-Left; Car-Buffer-Blocking-Height and Motor-Model. Without access to domain experts it is difficult to conclude whether these operators should be classified as type C or D.
- *Operators expressing functional constraints.* These operators express functional constraints which can be used to compute the values of parameters (type B). There are 177 of these.

Thus, my model comprises 202 procedures, which are used to compute 230 parameters.

9.5.3.1.2. Modelling VT procedures

A VT procedure is represented as a subclass of class `vt-design-operator`. A VT design operator refines the definition of design operators given in chapter 7 by means of two additional slots: `has-cost` and `depends-on`. The former is needed to model the association between costs and operators discussed in the Yost document. Here, I generalize the notion of 'cost of a fix' given by Yost to VT operators in general, by associating zero cost to procedures. The second slot, `depends-on`, is used to represent

---

7 Here I use the classification of design operators discussed in section 7.3.3.

explicitly the dependencies between parameters enforced by an operator. These dependencies of course do not need to be given 'by hand' but can be automatically derived from the operator expressions. Finally, some procedures are also constraints: these procedures are represented as instances of class `vt-functional-constraint` and are associated with a constraint object specifying the relevant constraint expression.

```
(def-relation HAS-COST (?op ?n)
    "This relation models the assignment of costs to operators
     in the VT domain"
   :iff-def (and (integer ?n)
                 (≥ ?n 0)
                 (≤ ?n 10)))

(def-class VT-DESIGN-OPERATOR (design-operator)
  ((depends-on :type list :default-value nil)
   (has-cost)))

(def-class VT-PROCEDURE (vt-design-operator
                         design-extension-operator)
   ((has-cost :value 0))
  :lisp-class-name vt-procedure)

(def-class VT-FUNCTIONAL-CONSTRAINT (vt-procedure)
   ((associated-constraint :type vt-constraint))
  :lisp-class-name vt-functional-constraint)
```

9.5.3.1.3. Configuring the Propose task for the VT application

If a CMR-type approach is used, then there is no need to worry about design focus selection in a design extension context. Given that i) the dependency network specified by the VT procedures is acyclic and ii) only one procedure exists for each parameter, it follows that the same design model is produced at the end of the Propose task, regardless of the order in which the parameters have been assigned.

Less straightforward is the situation in an EMR-type scenario, in which different sequences of parameter assignments lead to different solutions. In particular only a very specific (and statistically unlikely) sequence of design extension steps leads to the optimal solution discussed in the Yost document[8]. This sequence of constraint violations can be obtained by adding the appropriate focus selection knowledge to the VT knowledge base.

---

[8] Specifically, this sequence requires that i) the constraint violation concerning `max-machine-groove-pressure` is uncovered before computing the values associated with the `max-traction-ratio` constraint and that ii) the latter is fixed before all the values affecting constraint `min-machine-beam-section-modulus` have been computed.

However, such knowledge is not expressed in the system documentation[9]. Given that this knowledge is essential to achieve the solution discussed in the test case, it is likely that it was embedded in the original elevator design system, albeit in a compiled form.

In my model I addressed this problem simply by adding the appropriate focus ordering knowledge to the VT application knowledge base. In particular this knowledge ensures that the parameters participating in the max-machine-groove-pressure constraint are computed as soon as possible and that those participating in the min-machine-section-beam-modulus are computed as late as possible. A sample rule modelling part of the focus ordering knowledge is shown below. This rule states that parameter machine-groove-pressure should be selected before any parameter but those which participate in constraint max-machine-groove-pressure.

```
(def-rule design-focus-order-in-vt-1
  ((design-focus-order machine-groove-pressure ?x)
   if
   (not (member  ?x '(HOIST-CABLE-DIAMETER
                      SPEED
                      MACHINE-GROOVE-MODEL)))))
```

This body of application-specific knowledge ensures that EMR reaches the optimal solution when applied to the test case given in the VT specification. Of course, this is quite an ad hoc solution and therefore not completely satisfactory. However, without additional knowledge about the VT domain it is not possible to do better than this.

Finally, when discussing the KMI and Sisyphus-I applications I showed that domain-independent heuristics could be effectively used to drive the focus selection process. However, these heuristic techniques do not perform very well in the VT domain. The complexity of the interconnections between parameters, fixes and constraints requires application-specific focus selection knowledge.

9.5.3.2. Modelling the Revise task

9.5.3.2.1. Modelling fixes and fix combinations

In the previous chapter I pointed out that fixes are design modification operators directly associated with constraints. However, a quick comparison of the definition of class

---

9 The Yost document specifies that if constraints max-machine groove pressure and max-traction ratio are violated at the same time then max-machine groove pressure should be attempted first. Unfortunately there are two problems with this statement: i) it does not say that an EMR problem solver must ensure that these two constraint violations arise in this order and ii) this sequence is only a necessary rather than sufficient condition (i.e. it is possible to construct non optimal solution paths in which max-machine groove pressure is tackled before max-traction-ratio).

`design-fix` given in the previous chapter with the description of the fix mechanism given in the Yost document suggests that fixes in VT are much more complicated than those discussed in the previous chapter. Yost distinguishes between *incremental* and *non-incremental* fixes and indicates that each individual fix is a specific instance of a more general notion of *fix combination*. This is a structure comprising a number of fixes which are applied simultaneously. For a given constraint violation a number of fix combinations can be available and these should be tried according to a particular criterion. This is given by Yost informally, by showing how to order a four element combination comprising two fixes at desirability level 1 and two at desirability level 2. In (Motta et al., 1996) we formalized this criterion, by assigning a 'cost' to each fix combination. This is defined as follows. Let's suppose we have a fix combination Fc, comprising individual fixes $f_1, ..., f_n$. We know already that each f_i has an associated cost. The cost of Fc is then defined as a 10 element vector $<x_{10},....., x_1>$, where x_i is the number of individual fixes in Fc, with cost i. Fix combinations are then tried in order, from the cheapest to the most expensive.

Incremental fixes are defined as a subclass of class `multiple-design-fix`, which, in turn, is a subclass of `multiple-design-modification-operator`. This is analogous to `multiple-design-extension-operator`. Both describe operators which can be applied repeatedly to a state, to generate multiple state successors from the same operator. Class `vt-incremental-fix` is defined as follows.

```
(def-class vt-incremental-fix (vt-design-operator
                               multiple-design-fix)
   ((has-counter :default-value 0)))
```

The slot `has-counter` is needed to emulate the non-functional behaviour of incremental fixes. When an incremental fix has to be applied, a copy of the fix is created and the counter is increased after each application. This solution requires a VT-specific customization of method `collect-operators-applicable-to-focus`.

Strictly speaking, fix combinations are not a different kind of data structure but rather a way of expressing a particular search strategy through the revision space. In particular, the notion of fix combination and the particular fix application ordering policy presented in the Yost document define a search strategy for navigating the revision space in a cost-conscious manner. This point can be illustrated by considering the revision space associated with an inconsistent state s_i, which can be solved by means of three fixes, $f_1<f_2<f_3$, where f_1 is the cheapest and f_3 is the most expensive.

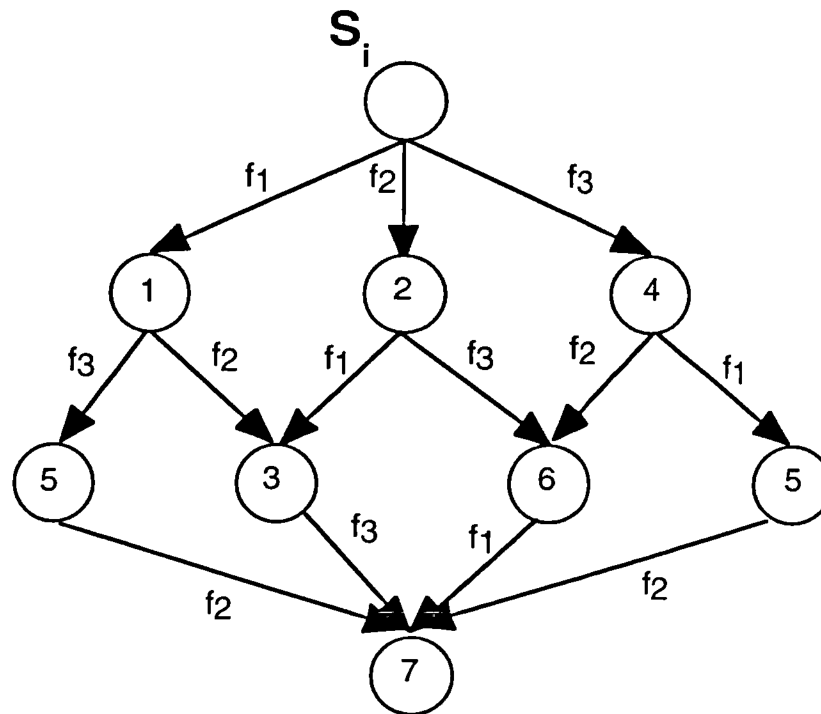


Figure 9.14. Cost-conscious search strategy in VT

Figure 9.14 shows the order in which the states in the revision space of s_i should be derived, in accordance with the cost-conscious policy informally expressed in the Yost document. Thus, it is not essential to model fix combinations explicitly; ordinary fixes suffice as long as the appropriate state selection strategy is defined. If fix combinations are explicitly modelled, then the state selection policy used by completion-centred methods - see section 8.10.1 - precisely captures the behaviour of an EMR problem solver.

9.5.3.2.2. Focus selection in a `:revise` context

The set of design foci in a `:revise` context consists of the constraints violated by the current design model. If an EMR approach is used, then constraint violations are dealt with on a first-come, first-served basis; therefore, focus selection is trivial. If a CMR approach is used, then 'clever' focus selection is only needed if the problem solver is not allowed to select an alternative constraint violation, after failing to fix the previously selected one. Of course, this would be an unnecessarily restrictive policy. Nevertheless, as already pointed out in section 9.5.3.1.3, it is interesting to note that different sequences of constraint violation fixing lead to different solutions (and if no backtracking to alternative foci is allowed, then many sequences lead to a deadend). This aspect is illustrated by figure 9.15 - taken from (Motta et al., 1996) - which shows a discrimination tree induced by trying out all admissible sequences of constraint fixing in the VT test case. The figure clearly shows that, out of 360 possible sequences, only 150 lead to a solution. Moreover, a finer-grained analysis - see (Motta et al., 1996; pages 364-367) -

shows that only 10% of the possible sequences of constraint fixing lead to the cheap solution documented in the test case.

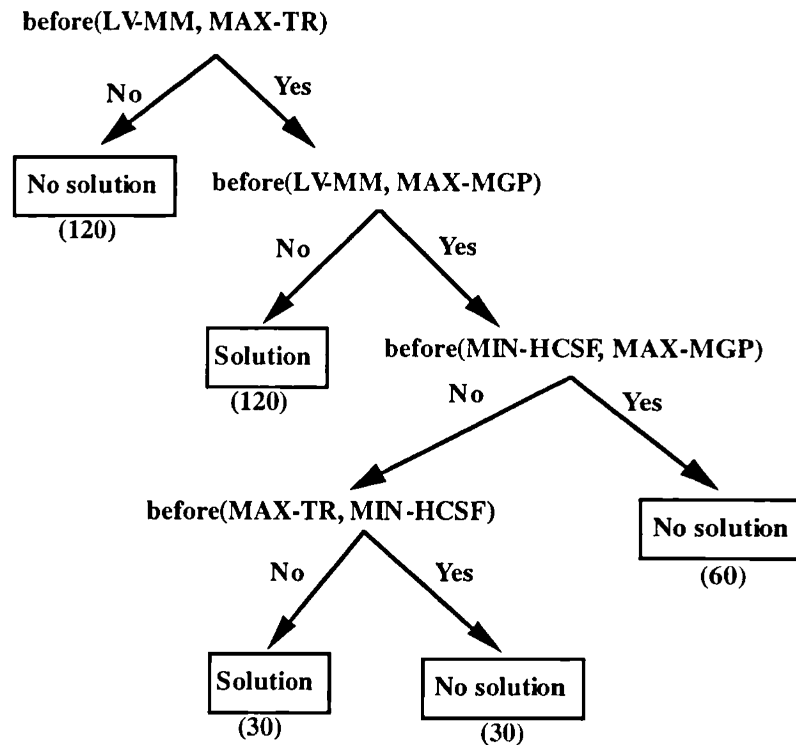


Figure 9.15. Solution and non-solution paths in the VT domain

9.5.3.2.3. Modelling the VT dependency network

The VT parameters are structured according to a dependency network defined by the chain of 177 functional constraints which are used to compute parameter values. The dependency network is modelled using the relations `depends-on` and `affects`, which are part of the generic method ontology for parametric design. No special customization was needed when implementing the VT system.

9.5.3.3. Experimental results

I run a number of trials applying both CMR and EMR to the VT knowledge base and I was able to reproduce the solution to the test case given in the Yost document with both models. In addition I was also able to generate other, more expensive solutions. The sequence of constraint fixing steps which produced the optimal solution is shown in the box below.

```
-- Focus on LEGAL-VALUE-MOTOR-MODEL
-- Current operator is INC-MACHINE-MODEL-5-14-9
-- New assignment is <MACHINE-MODEL 28>

-- Focus on MIN-PLATFORM-TO-HOISTWAY-LEFT
-- Current operator is INC-OPENING-TO-HOISTWAY-LEFT
-- New assignment is <OPENING-TO-HOISTWAY-LEFT 33>

-- Focus on MAX-VERTICAL-RAIL-FORCE
-- Current operator is INC-CAR-RAIL-UNIT-WEIGHT
-- New assignment is <CAR-RAIL-UNIT-WEIGHT 11>

-- Focus on MAX-MACHINE-GROOVE-PRESSURE
-- Current operator is INC-HOIST-CABLE-QUANTITY-2
-- New assignment is <HOIST-CABLE-QUANTITY 5>

-- Focus on MAX-TRACTION-RATIO
-- Current operator is COMBINED-FIX-FOR-MTR
-- New assignment is <CAR-SUPPLEMENT-WEIGHT 500>
-- New assignment is <CWT-TO-PLATFORM-REAR 1.75>
-- New assignment is <COMP-CABLE-MODEL 3/16-CHAIN>

-- Focus on MIN-MACHINE-BEAM-SECTION-MODULUS
-- Current operator is INC-MACHINE-BEAM-MODEL
-- New assignment is <MACHINE-BEAM-MODEL S10x35.0>

-- Design is now complete.  Solution cost is (0 0 1 0 2 0 4 1 0 0)
```

The EMR method proved to be on average more than twice more efficient than CMR (0.5 vs. 0.2). The main reason for this result is that EMR fixes most constraints in a subset of the overall space of parameters, thus reducing the number of constraint checks and dependency propagation steps compared to CMR.

9.5.3.4. Evaluation of the VT application

The VT application is specified in a highly optimized and method-oriented style in the Yost document. Such viewpoint provides both the strength and the weakness of the data set. On the plus side, the method-oriented specification makes it possible to implement

very efficient problem solvers for a complex problem such as VT. On the minus side this 'procedural' specification is both opaque and brittle. Given that I have already discussed the 'opacity' of the specification (in relation to focus ordering knowledge), I will now focus on the aspect of brittleness.

The brittleness of the VT knowledge base can be easily seen by trying out input specifications different from the one given in the Yost test case. In (Zdrahal and Motta, 1996) we presented some results, derived by testing a Propose&Revise problem solver on 25 input specifications. These were generated by combining all possible values of parameters `speed` and `capacity`. Table 9.11 shows that the knowledge base of fixes and procedures only allows us to fix less than 50% of the possible cases.

	Speed [ft/min]				
Capacity [pound]	**200**	**250**	**300**	**350**	**400**
2000	**Success**	**Success**	**Fail**	**Success**	**Success**
2500	**Fail**	**Fail**	**Success**	**Success**	**Success**
3000	**Fail**	**Success**	**Success**	**Success**	**Success**
3500	**Success**	**Fail**	**Fail**	**Fail**	**Fail**
4000	**Fail**	**Fail**	**Fail**	**Fail**	**Fail**

Table 9.11. Competence of the Propose&Revise solution to VT

Clearly these results are not consistent with our intuition that, if a solution (i.e. motor) is found for a certain speed and capacity, then some solution will also be found for the same speed and lower capacity, and for the same capacity and lower speed. For instance, while our VT problem solver can fix the test case given in Yost, [speed=250; capacity=3000], it cannot solve the apparently simpler case, [speed=200; capacity=3000]. Even worse, it is easy to verify that the solution to the case [speed=250; capacity=3000] is also a solution to the case [speed=200; capacity=3000].

The main reason for such brittle behaviour is the incompleteness of the knowledge base of fixes: not all potentially fixable constraints have fixes associated and not all possible fixes are associated to fixable constraints. A solution to this problem is to allow the behaviour of the VT problem solver to 'gracefully degrade' to dependency-directed backtracking when all fixes for a constraint have been tried. Another is to add the missing fixes. Both these approaches were able to solve the 'problematic' input specification, [speed=200; capacity=3000], by generating the solution model which also solves the [speed=250; capacity=3000] case. However, solving the [speed=200; capacity=3000] test case required more design revision steps than in the other case and

therefore it turns out that the cost of solving the input specification, [speed=200; capacity=3000], was higher than that required to solve [speed=250; capacity=3000]. This is of course counter-intuitive - even more so given that the two models are absolutely identical!

9.5.3.5. Conclusions

The VT problem is normally regarded as a complex design application (Stefik, 1995). Nevertheless, I was able to build several application models for it, by means of reusable library components. Very little configuration effort was required - only 10 additional definitions were needed in total, for building both CMR and EMR VT problem solvers.

The rational reconstruction discussed here has clarified the nature of various aspects of the VT application knowledge (e.g. the role of fix combinations; the nature of the 'breaks' in the dependency network mentioned in the Yost document; the opaque focus selection knowledge).

Finally, the discussion has also highlighted the brittleness of the VT application knowledge and the counter-intuitive results which can arise as a result of the method-oriented cost model described in the Yost document.

9.5.3.6. Comparison with some contributions from the Sisyphus-II initiative.

In what follows, I will briefly compare my analysis of the VT application with three alternative approaches proposed in the context of the Sisyphus-II initiative: Protégé-II (Rothenfluh et al., 1996), Common KADS (Schreiber and Terpstra, 1996) and DIDS (Runkel et al., 1996).

9.5.3.6.1. Protégé-II solution

The approach taken by the Protégé-II group is quite bottom-up. Rothenfluh et al. analyse the knowledge structures presented in the Yost document (e.g. fixes and constraints) and classify them in various categories (e.g. they distinguish three types of fixes). However this analysis only produces limited insights into the domain knowledge. For instance, they fail to note the heuristic nature of some of the 'constraints' present in the VT domain theory (i.e. in VT-Onto). Moreover, they also fail to address the cost-related aspects of the problem specification and the problem solving method. This seems to me a consequence of the limited modelling leverage provided by their task analysis - e.g. there is no notion of preference and cost. In contrast with their approach, my task analysis is driven by a rich task ontology, which makes it possible to characterize the nature of the VT task knowledge and identify eventual 'holes' in the specification.

Another important difference between the two approaches is at the methodological level. My framework distinguishes between task, method and domain ontologies and divides application configuration knowledge into mapping and application-specific knowledge.

Rothenfluh et al. define application knowledge as an application-specific, method-independent customization of a reusable domain model and they use mapping relations to link application and method ontologies. Thus, in their framework application knowledge is method independent. It seems to me that this view is problematic. The knowledge requirements imposed by a method cannot necessarily be satisfied by means of mapping knowledge. For instance, the heuristic nature of some design extension and fix knowledge cannot be envisaged in an application model unless a Propose&Revise method is selected. Another clear example is the heuristic function required by an A* method: it is difficult to imagine a method-independent body of application knowledge which includes the knowledge required to define such a function. Therefore, it seems to me that it is more appropriate to consider application configuration knowledge as the 'glue' which integrates reusable problem solving methods and domain models.

Finally, it is interesting to note that the Protégé-II solution uses a CMR-style approach - i.e. a complete solution model is generated before any revision is attempted. The revision strategy itself appears to pursue a hill-climbing approach in which all available fixes are applied in parallel and the best state (according to a state evaluation function) is chosen. A nice feature of this approach is that it does away with the notion of fix combination. A possible drawback of this approach is that the revision space is explored in a non-cost-conscious style (although, paradoxically, this search strategy appears to derive the optimal solution!).

9.5.3.6.2. Common KADS solution

The paper by Schreiber and Terpstra (1996) provides a very good description of a structured KBS analysis and design process, carried out in accordance with the Common KADS approach (Schreiber et al., 1994b). In particular, Schreiber and Terpstra clearly distinguish between the task-oriented aspects of the VT specification and the method-oriented description of a Propose&Revise problem solver, and propose ontology mappings to integrate the two. This approach is different from mine - in which a method ontology specializes a task ontology in accordance with a generic problem solving paradigm - and is, in principle, more flexible and modular.

Another interesting aspect of the Common KADS solution is that it effectively reuses the VT domain model provided as part of the Sisyphus-II data set. This reuse was accomplished by defining the appropriate *representation mappings* to translate between Ontolingua and Prolog, which is the language used by the SIADL environment (Terpstra, 1994) used to implement the VT system. Thus, the KBS development process described by Schreiber and Terpstra comprises a number of different mappings and transformations which are necessary for integrating the three modular specifications of the task, method and domain components. The resulting model can be seen as a generalization of the

modelling framework I have used to develop my library and the various application models.

However, while I like the 'big picture' constructed by Schreiber and Terpstra, I have some problems with the fine-grained aspects of their model; in particular with their approach to reusing VT-Onto. Specifically, it is difficult to reconcile their reuse of VT-Onto with the fact that their system solves the sample test case, given that the VT-Onto formalization overconstrains the problem. However, if we look more closely at their construction, we can see that not all constraints in VT-Onto are effectively used as constraints in the Common KADS model. In fact, this uses only the fifty constraints listed in the Yost document. The other constraints in VT-Onto, which correspond to procedures in VT-Yost, are not reused as constraints, but as procedures. Given that no distinction between these two classes of constraints is made in VT-Onto, it seems to me that such 'selective reuse' is somewhat unprincipled.[10]

9.5.3.6.3. DIDS solution

I have already written extensive comparisons between my approach and the DIDS one in earlier sections (6.5.2 and 7.5.1). These comparisons, especially section 7.5.1, draw heavily on the VT example and therefore I refer the reader to them.

9.6. CONCLUSIONS

The acid test of any approach to KBS reuse is the construction of effective problem solvers. The term 'effectiveness' in this case refers not just to the quality of the end system, but also to the rapidity of the system development process. The application models discussed in this chapter score highly on both criteria. They were developed by reusing the problem solving components from the library and very little configuration effort was needed. Nevertheless, i.e. despite the relatively low effort required to build the application models, all three application domains were tackled successfully, in each case achieving high-quality solutions, both in terms of competence and performance.

From an analytical point of view, the models presented in this chapter have provided a number of novel insights into the nature of the domain knowledge in the various application domains. These insights were obtained thanks to the use of i) rich task and method ontologies, which structure task and application analysis, and ii) the search

---

[10] In (Motta and Zdrahal, 1995) we have discussed this point more in detail and criticized the approach taken by Schreiber and Terpstra, who distinguish between 'real' constraints and procedures in VT-Onto by looking at the syntactic structure of the associated expressions. It seems to us that this approach goes against the idea of a reusable ontology providing a semantic basis for reuse, i.e. a "content-specific agreement" (Gruber, 1993).

paradigm as a basis for understanding the role of problem solving components expressed in task-specific terminology.

Hence, on the basis of the evidence presented in this chapter, the following claims can be made.

- The TMDA framework provides a useful epistemology for characterizing the generic components of an application model.
- The OCML language effectively supports the specification and prototyping of knowledge models.
- The use of search as a foundational basis for problem solving makes it possible to understand the computational role of problem solving components specified in task-specific terms.
- The parametric design task ontology provides a rich conceptual structure, which supports an effective, model-based knowledge acquisition process for developing task models.
- The library of generic problem solving components for parametric design provides effective support for KBS development by reuse. The dual principles of component modularity and method generality both limit the need for method configuration and, at the same time, facilitate it.

In a nutshell, the examples discussed in this chapter suggest that the proposed technologies afford both engineering and analytical leverage. Both kinds of leverage are needed, given that the goals of understanding knowledge-intensive problem solving and constructing effective problem solvers are tightly coupled.

Chapter 10. Concluding Remarks

In this final chapter I restate the main contributions of this thesis, in particular discussing the relevance of the results presented here to a number of generic research areas, such as knowledge acquisition, ontologies, problem solving and software reuse. Having emphasized what I regard as the main strengths of the proposed research, I will then discuss a number of open research issues, which arise directly from the work described here, or are closely associated with it. Some of these issues are essentially scholarly in nature, in the sense that they pertain to the future research work which needs to be carried out, in order to further evaluate and enhance this research. In addition to these 'scholarly issues', I will also look at the broader issues concerning knowledge modelling technology and, in particular, I will discuss the problems which need to be tackled, if we wish to lower the high entry barriers which currently prevent the diffusion of knowledge modelling technology on a large scale.

10.1 LEGACY OF THE WORK

In this thesis I have illustrated an approach to the development of a library of reusable components for knowledge modelling and to the construction of application models by reuse. This approach has been applied to parametric design problem solving, thus producing a number of reusable technologies and several fully configured application models. Hence, as discussed in section 1.1.2, this thesis can be seen as contributing to research in both knowledge modelling and design problem solving. More generally, the results presented here are relevant to a number of research areas, such as knowledge acquisition, ontologies, problem solving, and software reuse. In what follows, I will restate the main contributions of the thesis, in relation to several research areas.

10.1.1 Epistemological foundations of knowledge-based systems.

In chapter 3 I presented the TMDA modelling framework, which characterizes the generic classes of knowledge-based components which constitute an application model. While the TMDA framework has much in common with other proposals - e.g. VITAL, KADS, Protégé-II - in this thesis I have argued that it provides the 'right' set of distinctions, which are needed for structuring library construction and application development. This claim has been validated empirically, by showing that the application models developed

according to the TMDA framework exhibit advantages in relation to both application analysis and development. In particular, I argued these points by comparing the application models I developed for the Sisyphus applications with alternative proposals. The positive results in terms of reuse and quality of application models provide evidence for the validity and utility of the TMDA framework.

10.1.2 Problem solving.

Generally speaking, there has been very little intersection between research on symbol-level algorithms and research on knowledge-level models.[1] The consequences of maintaining such a dichotomy have been negative for both knowledge modelling and symbol-level AI. In particular, the following problems can be identified.

- While several weak methods can be usefully applied in knowledge engineering and integrated with knowledge-intensive problem solvers (as shown in chapters 7 and 8), the traditional antithesis between strong and weak method has posed a kind of 'ideological barrier' to a fruitful transfer of results and techniques from symbol-level AI (in particular search methods) to knowledge engineering.
- Deprived of task-independent computational foundations, problem solving methods have been formulated in task-specific terms. As a result, they are only meaningful in a restricted context, thus limiting the possibilities for reuse (van Heijst and Anjewerden, 1996).
- While symbol-level AI enjoys established computational foundations, e.g. search, foundational studies for knowledge modelling, e.g. the competence theory (Wielinga et al., 1995), have so far achieved only limited results. Thus, only task-specific foundations are normally available for knowledge models - see, e.g., (Benjamins, 1993). Unfortunately, as discussed in chapter 9 when analysing the Propose&Revise problem solver developed by Marcus et al. (1988), without a sound computational basis, problem solving methods may end up being both opaque and brittle.

---

[1] A notable exception is the work on Soar, which i) characterizes search and knowledge-level analysis as two distinct levels of system description (Smith and Johnson, 1993) and ii) proposes a framework for integrating them. The main difference between my approach and that used in Soar is that while I use search as a methodological device, to mediate between task and method dimensions, Soar uses search as a distinct, fully fledged computational architecture - i.e. as a kind of high level Turing machine. In contrast with Soar, my emphasis is on identifying the generic problem solving components - ontologies and generic tasks; search only provides a problem solving foundation, rather than an actual problem solving architecture.

Thus, the work presented here goes beyond the simple dichotomies used in the past and shows how libraries of problem solving components can be given a dual foundation: a task-independent one provided by the search paradigm and a task-specific one provided by a task ontology. As shown by the discussion in chapters 7, 8, and 9, once such a dual foundation is in place, it becomes possible to construct problem solvers which capitalize on the results from the 'weak method' literature, while subscribing to task-specific, knowledge-intensive paradigms.

10.1.3 Ontologies.

In this thesis I have contributed to the area of ontological engineering both by constructing practical, reusable ontologies and - more importantly - by showing how the notion of ontology can be integrated with a modelling framework. The results of such integration include: i) a methodology for defining reusable components; ii) a methodology for building well-structured applications out of reusable components; and iii) a typology of ontologies, which defines a generic epistemology for knowledge-based systems.

10.1.4 Libraries of problem solving components.

The contribution of this work to the development of libraries of problem solving components is both practical and theoretical. From a practical point of view I have developed a set of reusable components for parametric design (as well as the OCML base ontology) which can be effectively used to perform model-based knowledge acquisition, to specify and configure problem solving methods, and to develop application models. More generally, in this work I have outlined a principled approach to the construction of a task-specific library of problem solving components. In the proposed approach, which is based on the TMDA framework, a class of problems is described by means of a task ontology. Then, a generic, method-independent, but task-specific problem solving model is defined, by instantiating a search model of problem solving in terms of the concepts in the task ontology. This generic problem solving model provides the foundation from which alternative problem solving methods for a class of tasks can be defined. Specifically, the generic problem solving model provides i) a generic method ontology; ii) a set of highly generic building blocks for specifying problem solving methods, and iii) an initial, minimally committed problem solving method. More specific problem solving methods can then be (re-)constructed from the generic problem solving model through a process of ontology specialization and method-to-task application.

In contrast with general-purpose, coarse-grained libraries, such as CommonKADS and VITAL, the library presented here provides components which can be directly applied to produce fully configured problem solving methods. Moreover, the library is not just a set of task-specific definitions, but it enjoys both a task-specific and a task-independent foundation.

10.1.5 Software Reuse.

In this thesis I have proposed a framework for KBS reuse, which is based on the TMDA framework, and I have shown how different types of ontologies can be used to support the development of reusable model components. Moreover, I have discussed the epistemological basis for KBS reuse and emphasized the application-specific nature of problem solving knowledge. As a result, application development involves both a process of knowledge acquisition and one of ontology mapping. This view subsumes both the approach taken by those in the strong interaction camp as well as that taken by those in the weak interaction camp. Each of these two views tells a partial story. The framework proposed here shows the complete picture.

10.1.6 Knowledge Acquisition.

Although the term 'knowledge acquisition' nowadays covers a wide range of activities to do with knowledge elicitation, modelling and management, 'traditionally' the term has been used to refer to the problem of acquiring the knowledge needed for performing expert problem solving. As discussed in chapter 1, an important step forward in knowledge acquisition was provided by the emergence of the modelling view and the use of generic models of problem solving to drive the knowledge acquisition process. Although some evaluation studies have cast doubts on the effectiveness of model-based knowledge acquisition (Corbridge et al., 1995), it is fair to say that the majority of researchers and practitioners in the knowledge acquisition community still subscribe to the assumption that the contextualized nature of knowledge requires the use of strong models for supporting effective knowledge acquisition and for constructing robust and maintainable performance systems. In a nutshell, knowledge acquisition is effective if we know what are we looking for, i.e. if the right distinctions are enforced. Given this perspective, I can highlight two contributions of this thesis to knowledge acquisition: i) at the epistemological level, the TMDA framework provides a set of generic distinctions which can be used to structure the knowledge acquisition process; ii) at the conceptual level the proposed task and method ontologies for parametric design provide conceptual templates which guide the construction of a task model and the configuration of generic problem solvers. In particular, the validity of the proposed epistemological and conceptual frameworks was demonstrated by comparing the quality of the task and application models developed here with that of alternative proposals in the literature.

10.1.7 Knowledge modelling languages.

The current library of OCML models comprises several hundred definitions and has been used to construct dozens of ontologies and application models. In particular, the integration of operational and non-operational features in the language has proven very important. As a result, the definitions in the library are semantically rich, while, at the

same time, they can be verified by means of rapid prototyping. This property of operationality, together with the provision of several mechanisms for integrating OCML models with other software components, also facilitates the development of application systems. As discussed in the previous chapter, it was possible to build reasonably efficient prototypical applications, simply by defining symbol level mechanisms implementing the task structure of problem solvers, with no need for re-coding the entire OCML model.

10.1.8 Design Problem Solving.

While the main aim of this work was to illustrate a principled approach to library and application development, the application of the approach to design problem solving has produced a number of useful and novel results in this area. In particular, the library of parametric design components illustrated in this thesis includes: i) task and method ontologies; ii) a set of building blocks for constructing parametric design problem solvers and iii) a number of fully configured problem solving methods. These technologies provide a useful set of resources for parametric design, as illustrated by the application models discussed in chapter 9.

10.2 OPEN ISSUES FOR FUTURE RESEARCH.

10.2.1 'Strategic' issues

The knowledge modelling techniques and resources discussed in this thesis are fairly specialized technologies, which can only be applied by a relatively small number of reasonably skilled knowledge engineers. As a result, without additional research and development, the technology discussed here, just like so much 'intelligent' technology developed in three decades of knowledge engineering research, will only have a very limited impact on the software industry and on 'ordinary' computer users.

Thus, I believe it is appropriate to begin the discussion about outstanding research issues by focusing on the problems which need to be overcome, if we wish to lower the high entry barriers, which currently prevent the diffusion of knowledge modelling technology on a larger scale than is currently the case. In particular, both the access costs and the level of specialized skill, required to exploit knowledge modelling technologies, can be significantly reduced by providing intelligent tools supporting the kind of application development by reuse illustrated in this thesis. A newly set up, collaborative research project ($IBROW^3$, 1997) aims to address these issues, by developing an intelligent brokering service enabling third party knowledge-component reuse through the World-Wide Web. In particular, the parametric design library presented in this thesis will provide the baseline resource for the initial pilot study. More generally, the scenario envisaged by the project is one in which suppliers will provide libraries of knowledge

components, according to some standard specification format, and customers will be able to consult these libraries - through intelligent brokers - to configure a knowledge system suited to their needs, through a process of method selection and adaptation. Because the service will be web-based and will rely on the intelligent brokering service, it is hoped that it will make knowledge modelling technology more accessible and cost effective.

In order to make the IBROW^3 vision a reality, a number of research topics will need to be addressed, which include: the specification of a standard PSM description language, the specification of an application description language, protocols allowing the interoperability of knowledge components at both the knowledge and symbol levels, and intelligent configuration software supporting semi-automated method selection and application configuration.

Another line of research - also related to making knowledge modelling technology more accessible to 'ordinary' computer users - concerns the development of task-specific application configuration shells, which can integrate the problem solving technology described here - ontologies and problem solving methods - with sophisticated user interfaces and knowledge acquisition front ends, to provide intelligent problem-solving services. For example, a resource assignment application configuration toolkit could be developed with relatively little effort, by customizing the parametric design technology presented here for resource assignment problems and building the relevant front ends. Again, the provision of such products would make this kind of technology accessible to large, non-specialist user bases, thus contributing to the diffusion of the technology.

10.2.2 'Scholarly' issues

In addition to the issues discussed above, which are, in a sense, strategic to the whole knowledge modelling area, there are also (of course!) a number of important, 'scholarly' issues, which need to be addressed, in order to validate and improve on the approach presented here.

10.2.2.1. Validation issues

In order to fully validate the approach proposed here, this will have to be tested on a different class of tasks, e.g. diagnosis, to show that the ideas presented here are applicable to problems other than parametric design. For instance, an interesting research exercise would be to carry out a rational reconstruction of the diagnostic library developed by Benjamins (1993), in accordance with the approach proposed here.

10.2.2.2. Application delivery issues

The technology presented here only provides limited support for the efficient delivery of high performance end systems. The simple shell for parametric design used in this work is not adequate to support large scale, industrial strength system delivery. Hence,

powerful tools, supporting efficient system delivery from robust application models, are needed. These tools will need to provide smart compilation facilities to generate efficient code from knowledge models, as well as support the design and maintenance of the application. While several of these tools exist in the software engineering area, to my knowledge none is currently available for KBS development.

10.2.2.3. Foundational issues

On the theoretical side, more work needs to be carried out to produce a 'complete theory' for the knowledge engineering field. Such a theory should characterize both the nature and the development process of problem solving methods. The approach presented here provides an initial basis for such a theory: problem solving methods are characterized in task-oriented terms, as specializations of a task-specific model of problem solving - see discussion in section 3.5.1. This model has a dual foundation, provided by a task ontology and the search paradigm.

As discussed in section 8.10.4, the task-specific framework proposed here can be easily generalized to provide task-independent specifications of problem solving methods (Fensel et al., 1997). Such a generalization provides another dimension for reuse. If we further generalize, we can characterize method specification as a process of navigating a three-dimensional space consisting of *problem-solving paradigms*, e.g. search; *problem spaces*, i.e. task ontologies; and *domain assumptions*, i.e. method ontologies (Fensel and Motta, 1998). This navigation process can be carried out by formulating the relevant *adapters* (Fensel, 1997). As discussed in (Fensel and Motta, 1998), we aim to develop this framework to provide a comprehensive theory of problem solving methods, subsuming both task-independent and task-specific approaches, and integrating knowledge-based development with 'conventional' software engineering approaches. The aforementioned paper provides an initial formulation of the theory.

10.3 CONCLUDING, VISIONARY, TECHNO-POLITICAL REMARKS

Stutt (1997) has suggested that the new *knowledge age*, which is characterized by *knowledge economies*, *global communication*, *cyberspaces*, and the *epistemification of technology* (Stutt and Motta, 1997), will require a new educational trivium, comprising *knowledge engineering ontologies*, *constructivist epistemology*, and *computer rhetoric*. Of course, many may regard this suggestion as an intellectual provocation (they would probably point out that only a very small percentage of the people on the planet have access to the Internet and that illiteracy is still with us, and not just in the third world). Nevertheless, it is clear that the emerging new world will require new forms of literacy. In particular, in (Stutt and Motta, 1997) we argue that knowledge modelling will become an *organic* (i.e. essential and fundamental) technology for the new knowledge age. If such a prediction turns out to be correct, then it will be crucial that not just the means of

knowledge fruition (e.g. web browsers), but also the means of production (e.g. knowledge modelling technology) will be available to the cyber users. Otherwise, a potentially democratic technology will turn out to be yet another form of intellectual imperialism - much as Latin was in the middle ages.

Hence, the human-centred development of knowledge modelling and the introduction of this technology as part of a basic computer curriculum for the knowledge age will be - at some point in the future - essential, not just to ensure the success of the technology, but also to ensure democratic forms of knowledge production.

References

Aben, M. (1994). Canonical Functions: Common KADS Inferences. In J. A. Breuker and W. Van de Velde (Editors), *The CommonKADS Library for Expertise Modelling.* IOS Press, Amsterdam, The Netherlands.

Akkermans, J. M., Wielinga, B. J. and Schreiber, A. T. (1993). Steps in constructing problem-solving methods. In N. Aussenac, G. Boy, M. Linster, J-G Ganascia and Y. Kodratoff (Editors). *Knowledge Acquisition for Knowledge-Based Systems - EKAW '93.* Lecture Notes in Artificial Intelligence, LNCS 723, Springer-Verlag. 1993.

Angele, J., Decker, S., Perkuhn, R. and Studer, R. (1996). Modeling Problem Solving Methods in New KARL. In B. Gaines and M. Musen (Editors), *Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop.* Banff, Alberta, Canada.

Anjewierden A., Wielinga B. and Shadbolt N. (1992). Supporting knowledge acquisition: The ACKnowledge project. In B. Lepape and L. Steels (Editors), *Enhancing the Knowledge Engineering Process -- Contributions from ESPRIT*, pp. 143-172. Elsevier Science, Amsterdam, The Netherlands.

Balkany A., Birmingham W.P. and Runkel J. (1994). Solving Sisyphus by Design. *International Journal of Human-Computer Studies, 40(2).* pp. 221-241.

Balkany A., Birmingham W.P., and Tommelein J. (1993). An Analysis of Several Configuration Design Systems. *Artificial Intelligence in Engineering, Design, and Manufacturing*, 7(1), pp. 1-17.

Benjamins, R. (1993). *Problem Solving Methods for Diagnosis.* PhD Thesis, Department of Social Science Informatics, University of Amsterdam.

Benjamins, R. and Pierret-Golbreich, C. (1996). Assumptions of Problem Solving Methods. *Proceedings of the 6th Workshop on Knowledge Engineering Methods and Languages.* Gif-sur-Yvette, France, January 15-16.

Bonnardel, N. and Sumner, T. (1996). Supporting Evaluation in Design. *Acta Psychologica, 91*, pp. 221-244.

Brachman, R. J. (1979). On the Epistemological Status of Semantic Networks. In N. V. Findler (Editor), *Associative Networks: Representation and Use of Knowledge by Computers.* Academic Press, New York, pp. 3-50.

Brachman, R. J., Fikes, R. E. and Levesque, H. J. (1985). KRYPTON: A Functional Approach to Knowledge Representation. In R. J. Brachman and H. J. Levesque (Editors). *Readings in Knowledge Representation.* Morgan Kaufmann, Los Altos, CA.

Brachman, R. J. and Levesque, H. J. (Editors) (1985). *Readings in Knowledge Representation.* Morgan Kaufmann, Los Altos, CA.

Brachman, R. J. and Smith, B. C. (Editors) (1980). Special Issue on Knowledge Representation. *SIGART Newsletter, 70.* February 1980.

Bradshaw, J. (1996). An Introduction to Software Agents. In Bradshaw, J. (Editor), *Software Agents.* AAAI Press/MIT Press, Menlo Park, California.

Brazier, F. and Wijngaards, N. (1997). A purpose driven method for the comparison of modelling frameworks. *Proceedings of the 7th Workshop on Knowledge Engineering: Methods and Languages - KEML '97.* The Open University, 22-24 January, 1997.

Breuker, J. A. and Van de Velde, W. (1994). *CommonKADS Library for Expertise Modelling.* IOS Press, Amsterdam, The Netherlands.

Breuker J. and Wielinga B.J. (1989). Models of Expertise in Knowledge Acquisition. In G. Guida and C. Tasso (Editors), *Topics in Expert Systems Design,* North-Holland.

Breuker J., Wielinga B.J., van Someren M., de Hoog R., Schreiber G., de Greef P., Bredeweg B., Wielemaker J., Billault J.-P., Davoodi M. and Hayward S. (1987). *Model Driven Knowledge Acquisition: Interpretation Models.* Deliverable D1, Esprit Project P1098, KADS. University of Amsterdam.

Brooks, R., A. (1991). Intelligence without Representation. Artificial Intelligence, *47(1-3)*, pp. 139-160. January 1991.

Buchanan, B. G. and Shortliffe, E. H. (1984). *Rule-Based Expert Systems.* Addison-Wesley, Reading, MA.

Bylander, T. (1991). Complexity Results for Planning. *Proceedings of the 12th International Joint Conference on Artificial Intelligence - IJCAI '91*, Sidney, Australia.

Bylander, T., Allemang, D., Tanner, M. C., and Josephson, J. R. (1991). The Computational Complexity of Abduction. *Artificial Intelligence 49.*

Bylander, T. and Chandrasekaran, B. (1988). Generic Tasks in Knowledge-Based Reasoning: The Right Level of Abstraction for Knowledge Acquisition. In B. Gaines and J. Boose (Editors), *Knowledge Acquisition for Knowledge-Based Systems - volume 1*, pp. 65-77. Academic Press, London.

Causse, K. (1993). Heuristic Control Knowledge. In N. Aussenac, G. Boy, M. Linster, J-G Ganascia and Y. Kodratoff (Editors). *Knowledge Acquisition for Knowledge-Based Systems - EKAW '93.* Lecture Notes in Artificial Intelligence, LNCS 723, Springer-Verlag. 1993.

Chandrasekaran, B. (1983). Toward a taxonomy of problem solving types. *AI Magazine, 4(4)*, pp. 9-17.

Chandrasekaran, B. (1986). Generic Tasks in Knowledge-based Reasoning: High-Level Building Blocks for Expert System Design. *IEEE Expert 1(3)*, pp. 23-30.

Chandrasekaran, B. (1990). Design Problem Solving: A Task Analysis. *AI Magazine, Winter Issue, 11(4)*, pp. 59-71.

Chandrasekaran, B., Johnson, T.R. and Smith, J.W. (1992). Task-Structure Analysis for Knowledge Modelling. *Communications of the ACM 35(9)*, pp. 124-137.

Cheeseman, P., Kanefsky, B and Taylor, W. M. (1991). Where the really hard problems are. *Proceedings of the 12th International Joint Conference on Artificial Intelligence - IJCAI '91.* Sidney, Australia, pp. 331-337.

Clancey W. J. (1983). The epistemology of a rule-based expert system: A framework for explanation. *Artificial Intelligence, 20(3)*, pp. 215-251.

Clancey W. J. (1985). Heuristic Classification. *Artificial Intelligence, 27*, pp. 289-350.

Conlon, T. and Pain, H. (1996). Persistent collaboration: A methodology for applied AIED. *Journal of Artificial Intelligence in Education, 7(3/4)*, pp. 219-252.

Corbridge, C., Major, N. P., and Shadbolt, N. R. (1995). Models exposed: an empirical study. In B. Gaines and M. Musen (Editors), *Proceedings of the 9th Banff Knowledge Acquisition Workshop*, Banff, Canada, January 26th -February 3rd, 1995

Cross, N., de Vries, M., and Grant, D. (1993). *Design Methodology and Relationships with Science.* Kluwer Academic Publishers, Dordrecht.

Davis, R. (1979). Interactive Transfer of Expertise: Acquisition of New Inference Rules. *Artificial Intelligence 12*, pp. 121-158.

Davis, R. and King, J. J. (1977). The Origin of Rule-Based Systems in AI. In Elcock, E. W. and Michie, D. (Editors), *Machine Intelligence 8: Machine Representations of Knowledge.* Ellis Horwood Ltd., Chichester, England.

Dechter, R. (1988). Constraint Processing Incorporating Backjumping, Learning and Cutset-Decomposition. *Proceedings of the 4th Conference on Artificial Intelligence Applications - CAIA '88*, pp. 312-319.

Dechter, R., and Meiri, I. (1989). Experimental Evaluation of Preprocessing Techniques in Constraint Satisfaction Problems. *Proceedings of the 11th International Joint Conference on Artificial Intelligence - IJCAI '89*, pp. 271-277. San Mateo, CA, Morgan-Kaufman.

Dechter, R., and Pearl, J. (1988). Network-based Heuristics for Constraint Satisfaction Problems. *Artificial Intelligence Journal, 34*, pp. 1-38.

Domingue, J., Motta, E. and Watt, S. (1993). The Emerging VITAL Workbench. In N. Aussenac, G. Boy, M. Linster, J-G Ganascia and Y. Kodratoff (Editors). *Knowledge Acquisition for Knowledge-Based Systems - EKAW '93.* Lecture Notes in Artificial Intelligence, LNCS 723, Springer-Verlag. 1993.

Dreyfus, H. (1979). *What Computers Can't do: A Critique of Artificial Reason.* Freeman.

Ehn, P. (1989). *Work-Oriented Design of Computer Artifacts.* Arbetslivscentrum, Stockholm.

Eshelman, L. (1988). MOLE: A Knowledge Acquisition Tool for Cover-and-Differentiate Systems. In S. Marcus (Editor), *Automating Knowledge Acquisition for Expert Systems*, pp. 37-80. Kluwer Academic Publishers.

Falasconi, S. and Stefanelli, M. (1994). A library of medical ontologies. In N. Mars (Editor), *Proceedings of the ECAI-94 Workshop on Comparison of Implemented Ontologies*, pp. 81-91. Amsterdam, 8-10 August 1994.

Farquhar, A., Fikes, R., and Rice, J. (1996). The Ontolingua Server: A Tool for Collaborative Ontology Construction. In B. Gaines and M. Musen (Editors), *Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop.* Banff, Alberta, Canada.

Feigenbaum, E. A. (1977). The Art of Artificial Intelligence: Themes and Case Studies of Knowledge Engineering. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, MA.

Fensel, D. (1995a). *The Knowledge Acquisition and Representation Language KARL.* Kluwer, Dordrecht.

Fensel, D. (1995b). Assumptions and Limitations of a Problem Solving Method: A case study. In B. Gaines and M. Musen (Editors), *Proceedings of the 9th Banff Knowledge Acquisition Workshop*, Banff, Canada, January 26th -February 3rd, 1995

Fensel, D. (1997). The Tower-of-Adapter Method for Developing and Reusing Problem-Solving Methods. In R. Benjamins and E. Plaza (Editors). *Knowledge Acquisition, Modeling, and Management. Proceedings of the 10th European Workshop, EKAW '97.* Lecture Notes in Artificial Intelligence 1319, Springer-Verlag.

Fensel, D. and Motta, E. (1998). Dimensions for Method Refinement. *Submitted to the 11th Workshop on Knowledge Acquisition, Modeling and Management (KAW'98).* Banff, Canada, April 18th - 23rd.

Fensel, D., Motta, E., Decker, S., and Zdrahal, Z. (1997). The use of Ontologies for Specifying Tasks and Problem Solving Methods: A Case Study. In R. Benjamins and E. Plaza (Editors). *Knowledge Acquisition, Modeling, and Management. Proceedings of the 10th European Workshop, EKAW '97.* Lecture Notes in Artificial Intelligence 1319, Springer-Verlag.

Fensel, D. and Schoenegge, A. (1997a). Specifying and Verifying Knowledge-Based Systems with KIV. *Proceedings of the European Symposium on the Validation and Verification of Knowledge Based Systems - EUROVAV-97*, Leuven Belgium, June 26-28, 1997.

Fensel, D. and Schoenegge, A. (1997b). Hunting for Assumptions as Developing Method for Problem-Solving Methods. In *Proceedings of the Workshop on Problem-solving Methods for Knowledge-based Systems.* Workshop held in connection with the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97). Nagoya, Japan, August 23-25, 1997.

Fensel, D. and Straatman, R. (1996). Problem solving methods: Making Assumptions for Efficiency Reasons. In N. Shadbolt, K. O'Hara, and Schreiber, G. (Editors). *Advances in Knowledge Acquisition - EKAW '96.* Lecture Notes in Artificial Intelligence, 1076. Springer-Verlag, Heidelberg.

Fensel, D. and van Harmelen, F. (1994). A Comparison of Languages which Operationalize and Formalize KADS Models of Expertise. *The Knowledge Engineering Review, 9(2).*

Fikes, R. E. and Kehler, T. (1985). The Role of Frame-Based Representation in Reasoning. *Communications of the ACM, 28(9)*, September 1985.

Flemming, U., Baykan, C. A., Coyne, R. F., and Fox, M. S. (1992). Hierarchical generate and test vs constraint-directed search. In J. S. Gero (Editor), *Artificial Intelligence in Design '92*, pp. 817-838. Kluwer Academic.

Ford, K., Stahl, H., Adams-Webber, J., Novak, J., Canas, A. and Jones, J. (1990). ICONKAT: A Constructivist Knowledge Acquisition Tool. In *Proceedings of the fifth Banff Knowledge Acquisition Workshop*, pp. 7.1-7.20. Banff, Canada.

Friedland, P. and Iwasaki, Y. (1985). The Concepts and Implementation of Skeletal Plans. *Journal of Automated Reasoning 1*, pp. 161-208.

Fuchs, N. E. (1992). Specifications are (preferably) executable. *Software Engineering Journal.* pp. 323-334. September 1992.

Gaines, B. R. (1994). A situated classification solution of a resource allocation task represented in a visual language. *International Journal of Human-Computer Studies, 40(2).* pp. 243-271.

Gaschnig, J. (1977). A General Backtracking Algorithm that Eliminates Most Redundant Tests. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, MA.

Gaschnig, J. (1978). Experimental case studies of Backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. *Proceedings of the 2nd Biennal Conference of the Canadian Society for Computational Studies of Intelligence.* Toronto, July 1978.

Gaschnig, J. G. (1979). Performance Measurement and Analysis of Certain Search Algorithms. *Doctoral Dissertation*, Department of Computer Science, Carnegie-Mellon University, 1979.

Gaspari, M., and Motta, E. (1994). Symbol-level Requirements for Agent-level Programming. *Proceedings of the 11th European Conference on Artificial Intelligence, ECAI '94.* Amsterdam, August 1994.

Gaspari, M., Motta, E, and Stutt, A. (1993). Inferring in Lego-land: an architecture for the integration of heterogeneous inference modules. In P. Torasso (Editor), *Advances in Artificial Intelligence*, Lecture Notes in Artificial Intelligence, LNCS 728, Springer-Verlag, pp. 142-153.

Gaspari, M., Motta, E, and Stutt, A. (1995). An Open Framework for Cooperative Problem Solving. *IEEE Expert, 10(3)*, pp. 48-55. June 1995.

Genesereth, M. R. and Fikes, R. E. (1992). Knowledge Interchange Format, Version 3.0. *Technical Report Logic-92-1*, Computer Science Department, Stanford University.

Genesereth, M. R. and Nilsson, N. J. (1988). *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann, Los Altos, CA.

Gennari, J. H., Tu, S. W., Rothenfluh, T. E. and Musen, M. A. (1994). Mapping Domains to Methods in Support of Reuse. *Proceedings of the 8th Banff Knowledge Acquisition Workshop*, Banff, Canada.

Gero, J. S. (1990). Design Prototypes: A Knowledge Representation Schema for Design. *AI Magazine, 11(4)*, pp. 26-36.

Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, MA.

Golomb, S. W. and Baumert, L. D. (1965). Backtrack Programming. *JACM 12*, pp. 516-524.

Greenbaum, J. and Kyung, M. (1991). *Design at Work: Cooperative Design of Computer Systems.* Lawrence Erlbaum Associates, Hillsdale, NJ.

Gruber, T. R. (1991). The Role of Common Ontology in Achieving Shareable, Reusable Knowledge Bases. In Allen, J. A, Fikes, R., and Sandewall, E. (Editors), *Principles of Knowledge Representation and Reasoning: Proceedings of the 2nd International Conference*, pp. 601-602. Morgan-Kaufmann, Cambridge, MA.

Gruber, T. R. (1993). A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition, 5(2).*

Gruber, T. R. (1995). Toward Principles for the Design of Ontologies Used for Knowledge Sharing. *International Journal of Human-Computer Studies 43(5/6)*, pp. 907-928.

Gruber, T. R., Olsen, G. R., and Runkel, J. (1996). The configuration design ontologies and the VT elevator domain theory. *International Journal of Human-Computer Studies 44 (3/4).*

Guarino, N. (1994). The Ontological Level. In R. Casati, B. Smith and G. White (Editors), *Philosophy and the Cognitive Science.* Holder-Pichler-Tempsky, Vienna, Austria.

Guarino N. (1997). Understanding, Building and Using Ontologies. A Commentary to "Using Explicit Ontologies in KBS Development", by van Heijst, Schreiber, and Wielinga. *International Journal of Human-Computer Studies, 46(2/3)*, pp. 293-310.

Guarino, N. and Giaretta, p. (1995). Ontologies and Knowledge Bases: Towards a Terminological Clarification. In N. Mars (Editor), *Towards Very Large Knowledge Bases: Knowledge Building and Knowledge Sharing.* IOS Press, Amsterdam, pp. 25-32.

Haralick, R. M. and Elliott, G. L. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence, 14*, pp. 263-313.

Hatala M. (1997). The OCML terminal. *Technical Report.* Knowledge Media Institute, The Open University.

Hayes-Roth, F., Waterman, D. A. and Lenat, D. B. (1983). *Building Expert Systems.* Addison-Wesley, New York.

Hayes, I. J. and Jones, C. B. (1989). Specifications are not (necessarily) executable. *Software Engineering Journal, 4(6)*, pp. 330-338. November 1989.

Hewitt, C. (1971). PLANNER: A Language for proving theorems in robots. *Proceedings of the 2nd International Joint Conference on Artificial Intelligence - IJCAI '71*, London, UK.

Horak J., Valasek M. and Bauma V. (1995). Knowledge Level Analysis of Bearings Design. *TR-Encode-CVUT-1-95* (Encode Project Report). Czech Technical University, Faculty of Mechanical Engineering. Prague.

IBROW^3 (1997). IBROW^3: An Intelligent Brokering Service for Knowledge-Component Reuse on the World-Wide Web. Esprit Open LTR proposal, #27169.

Jonker, W. and Spee, J. W. (1992). Yet Another Formalization of KADS Conceptual Models. In Th. Wetter, K.-D. Althoff, J. Boose, B.R. Gaines, M. Linster and F. Schmalhofer (Editors) *Current Developments in Knowledge Acquisition - EKAW '92*, pp. 112-32. LNAI 599, Springer-Verlag, Berlin.

Keng, N. and Yun, D. Y. Y. (1989). A planning/scheduling methodology for the constrained resource problem. *Proceedings of the 11th International Joint Conference on Artificial Intelligence - IJCAI '89*, pp. 998-1003. San Mateo, CA, Morgan-Kaufman.

Kiczales, G., des Rivieres, J., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA.

Kidd, A. (Editor) (1987). *Knowledge Acquisition for Expert Systems.* Plenum Press, New York.

Kirkpatrick, S., Gelatt, Jr, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science, 200*, pp. 671-680.

Klinker, G., Bhola, C., Dallemagne, G., Marques, D., and McDermott, J. (1991). Usable and Reusable Programmin Constructs. *Knowledge Acquisition 3*, pp. 117-136.

Klinker, G., Marques, D., McDermott, J., Mersereau, T., and Stinson, L. (1993). The Active Glossary: Taking Integration Seriously. *Knowledge Acquisition 5*, pp. 173-197.

Krueger, C. W. (1992). Software Reuse. *ACM Computing Surveys 24(2)*, pp. 131-183.

Laird, J., Newell, A., and Rosenbloom, P. (1987). Soar: An Architecture for General Intelligence. *Artificial Intelligence 33*, pp. 1-64.

Le Roux, B., O'Hara, K., Shadbolt, N., Outtandy, S., Laublet, P., and Motta, E. (1993). The VITAL Library for Knowledge Modelling. *VITAL Project Report DD215*, August 1993.

Lenat, D.B. and Guha, R.V. (1990). *Building Large Knowledge-Based Systems: Representation and Inference in the Cyc Project.* Addison-Wesley, Reading, MA.

Levesque, H. J. (1984). Foundations of a functional approach to knowledge representation. *Artificial Intelligence 23(2)*, pp. 155-212.

Linster, M. (1994). Problem Statement for Sisyphus: Models of Problem Solving. *International Journal of Human-Computer Studies 40(2)*, pp. 187-192.

MacGregor, R. (1991). Using a Description Classifier to Enhance Deductive Inference. *Proceedings of the 7th IEEE Conference on AI Applications.* Miami, Florida, February 1991.

Marcus S. (Editor) (1988). *Automatic Knowledge Acquisition for Expert Systems.* Kluwer Academic, Boston, MA.

Marcus, S. and McDermott, J. (1989). SALT: A Knowledge Acquisition Language for Propose and Revise Systems. *Journal of Artificial Intelligence, 39(1)*, pp. 1-37.

Marcus, S, Stout, J., and McDermott, J. (1988). VT: An Expert Elevator Designer that uses Knowledge-Based Backtracking. *AI Magazine*, 9(1), pp. 95-112, Spring 1988.

Martin, J. and Odell, J. J. (1995). *Object-Oriented Methods: A Foundation.* Prentice-Hall, Englewood Cliffs, New Jersey.

McDermott, J. (1988). Preliminary steps toward a taxonomy of problem-solving methods. In S. Marcus (Editor), *Automating Knowledge Acquisition for Expert Systems*, Kluwer Academic.

Merrian-Webster. (1997). The Merrian-Webster English Dictionary. Available online at URL http://www.m-w.com/home.htm.

Minton S., Johnson M.D., Philips A.B. and Laird P. (1992). Minimising conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence 58.* (1992). pp. 161–205.

Mittal, S. and Frayman, F. (1989). Towards a Generic Model of Configuration Tasks. *Proceedings of the 11th International Joint Conference on Artificial Intelligence - IJCAI '89*, pp. 1395-1401. San Mateo, CA, Morgan-Kaufmann.

Motta, E. (1995). KBS Modelling in OCML. *Proceedings of the fifth Workshop on Modelling Languages for KBS*, Vrije Universiteit Amsterdam, 30-31 January, 1995.

Motta, E. (1997). Trends in Knowledge Modelling: Report on the 7th KEML Workshop. *The Knowledge Engineering Review, Vol. 12(2)*, June 1997.

Motta E., O'Hara, K., and Shadbolt, N. (1994a). Grounding GDMs: A Structured Case Study. *International Journal of Human-Computer Studies 40(2)*, pp. 315-347.

Motta, E., O'Hara, K., Shadbolt, N., Stutt, A., and Zdrahal, Z. (1994b). A VITAL Solution to the VT Elevator Design Problem. Proceedings of the*Knowledge Acquisition for Knowledge Based Systems Workshop*, Banff, Canada, February 1994.

Motta, E., Rajan, T. and Eisenstadt, M. (1989). A Methodology and Tool for Knowledge Acquisition in KEATS-2. In G. Guida and C. Tasso (Editors), *Topics in Expert Systems Design*, North-Holland.

Motta E., Stutt A., Zdrahal Z., O'Hara K. and Shadbolt N. (1996). Solving VT in VITAL: a Study in Model Construction and Knowledge Reuse. *International Journal of Human-Computer Studies 44(3/4)*, pp. 333-371.

Motta E., Zdrahal Z. (1995). The Trouble with What: Issues in method-independent task specifications. In B. R. Gaines and M. Musen (Editors), *Proceedings of the 9th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, pp. 30-1 - 30- 17.

Motta, E. and Zdrahal, Z. (1996). Parametric Design Problem Solving. In Gaines, B. and Musen, M. (editors), *Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW'96)*, Banff, Canada, November 9th-14th, 1996.

Motta, E. and Zdrahal, Z. (1997). An approach to the organization of a library of problem solving methods which integrates the search paradigm with task and method ontologies. Submitted for publication. Available from http://kmi.open.ac.uk/~enrico/papers/ijhcs_psm.ps.gz. July 1997.

Murray, K. and Porter, B. (1988). Developing a Tool for Knowledge Integration: Initial Results. *Proceedings of the 3rd Banff Knowledge Acquisition Workshop*, Banff, Canada.

Musen, M. A. (1989). *Automated Generation of Model-Based Knowledge Acquisition Tools*. Research Notes in Artificial Intelligence, Pitman, London.

Musen, M. A., Fagan L. M., Combs, D. M., and Shortliffe E. H. (1987). Use of a Domain Model to Drive an Interactive Knowledge-Editing Tool.. *International Journal of Man-Machine Studies 26*, pp. 105-121.

Nebel, B. (1996). Artificial Intelligence: A Computational Perspective. In G. Brewka (Editor), *Essentials in Knowledge Representation*, Springer-Verlag.

Neches, R., Fikes, R., Finin, T., Gruber, T., Patil, R., Senator, T. and Swartout, W. (1991). Enabling Technology for Knowledge Sharing. *AI Magazine 12(3)*, pp. 37-56.

Newell A. (1980). Reasoning, Problem Solving, and Decision Processes: The Problem Space as a Fundamental Category. In R.S. Nickerson (Ed.), *Attention and Performance VIII*, Lawrence Erlbaum Associates, Hillsdale, New Jersey.

Newell A. (1982). The knowledge level. *Artificial Intelligence, 18(1)*, pp. 87-127.

Newell A. (1990). *Unified Theories of Cognition*. Harvard University Press.

Newell, A. and Simon, H. A. (1972). *Human Problem Solving*. Prentice Hall, Englewood, NJ.

Newell, A. and Simon, H. A. (1976). Computer Science as Empirical Enquiry: Symbols and Search. *Communications of the ACM, 19(3)*, pp. 113-126, March 1976.

Nilsson, N. J. (1980). Principles of Artificial Intelligence. Morgan Kaufmann, Los Altos, CA.

Nonaka, I. and Takeuchi, H. (1995). *The Knowledge Creating Company: How Japanese Companies Create the Dynamics of Innovation*. New York, Oxford University Press.

O'Hara, K. (1993). A Representation of KADS-I Interpretation Models Using a Decompositional Approach. In C. Löckenhoff, D. Fensel and R. Studer (Editors), *Proceedings of 3rd KADS Meeting*, pp.147-69. Siemens AG, Munich.

O'Hara, K. (1995). The GDM Grammar, v.4.6. *VITAL Project Report NOTT/T252.3.3*. Available from the author at AI Group, Department of Psychology, University of Nottingham, UK.

Orsvärn, K. (1996). Principles for Libraries of Task Decomposition Methods - Conclusions from a Case-study. In N. Shadbolt, K. O'Hara, and G. Schreiber (Editors). *Advances in Knowledge Acquisition - EKAW '96*. Lecture Notes in Artificial Intelligence, 1076. Springer-Verlag, pp. 48-65.

Poeck, K. and Gappa, U. (1993). Making Role-Limiting Shells More Flexible. In N. Aussenac, G. Boy, M. Linster, J-G Ganascia and Y. Kodratoff (Editors). *Knowledge Acquisition for Knowledge-Based Systems - EKAW '93*. Lecture Notes in Artificial Intelligence, LNCS 723, Springer-Verlag. 1993.

Poeck, K. and Puppe, F. (1992). COKE: Efficient Solving of Complex Assignment Problems with the Propose-and-Exchange Method. *5th International Conference on Tools with Artificial Intelligence*. Arlington, Virginia.

Puerta, A. R., Egar, J. W., Tu, S. W., and Musen, M. A. (1992). A multiple-method knowledge-acquisition shell for the automatic generation of knowledge-acquisition tools. *Knowledge Acquisition, 4(2)*, pp. 171-196.

Punch, W. (1989). *A diagnosis system using a task integrated problem solver architecture (TIPS), including causal reasoning*. PhD Thesis, Ohio State University.

Rich, E. and Knight, K. (1991). *Artificial Intelligence*. McGraw-Hill, USA.

Rothenfluh, T. E., Gennari, J. H., Eriksson, H., Puerta, A. R., Tu, S. W., and Musen, M. A. (1996). Reusable Ontologies, Knowledge-Acquisition Tools, and Performance Systems: PROTEGE-II Solutions to Sisyphus-II. *International Journal of Human-Computer Studies 44 (3/4)*, pp. 303-332.

Runkel J. T., Birmingham W.P., Darr, T. P., Maxim, B. R. and Tommelein, I. D. (1992). Domain-Independent Design System. In J. S. Gero (Editor), *Artificial Intelligence in Design '92*, pp. 21-40. Kluwer Academic Publishers.

Runkel, J. T., Birmingham, W. B., Balkany, A. (1994). Separation of Knowledge: a Key to Reusability. *Proceedings of the 8th Banff Knowledge Acquisition Workshop*. Banff, Canada, 1994.

Runkel, J. T., Birmingham, W. B., Balkany, A. (1996). Solving VT by Reuse. *International Journal of Human-Computer Studies 44 (3/4)*, pp. 403-433.

Sadeh, N. and Fox, M. S. (1996). Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence, 86(1)*, pp. 1-41, September 1996.

Schoen, D. A. (1983). *The Reflective Practitioner: How Professionals Think in Action*. New York, Basic Books.

Schreiber, A.T. (1992). *Pragmatics of the Knowledge Level.* PhD Thesis, University of Amsterdam.

Schreiber, A. T. (1994). Applying KADS to the office assignment domain. *International Journal of Human-Computer Studies, 40(2)*, pp. 349–377.

Schreiber, A.T. and Birmingham W.P. (Editors) (1996). Special Issue of the *International Journal of Human-Computer Studies, 44(3/4)*. March/April 1996.

Schreiber, A. T. and Terpstra, P. (1996). Sisyphus-VT: A CommonKADS solution. *International Journal of Human-Computer Studies, 44(3/4)*, pp. 373–402, March/April 1996.

Schreiber, A.Th., Wielinga B. J., Akkermans, H., van de Velde, W., and Anjewierden, A. (1994a). CML: The CommonKADS Conceptual Modelling Language. In L. Steels, A. T. Schreiber, and W. van de Velde (Editors), *A Future for Knowledge Acquisition, Proceedings of the 8th European Knowledge Acquisition Workshop*. Springer Verlag, LNAI 867, pp. 283-300.

Schreiber, A. T., Wielinga B. J., de Hoog, R., Akkermans, H., van de Velde, W., and Anjewierden, A. (1994b). CommonKADS: A Comprehensive Methodology for KBS Development. *IEEE Expert*, December 1994, pp. 28-37.

Schreiber, A.Th., Wielinga B. J. and Jansweijer, W. H. J. (1995). The KACTUS View on the 'O' Word. In J. C. Bioch and Y.-H. Tan (Editors). *Proceedings of the 7th Dutch National Conference on Artificial Intelligence - NAIC '95*. EURIDIS, Erasmus University, Rotterdam, The Netherlands, pp. 159–168.

Shadbolt, N., Motta, E., and Rouge, A. (1993). Constructing Knowledge Based Systems. *IEEE Software, 10(6)*, pp. 34-38.

Shortliffe, E. H. (1976). *Computer-Based Medical Consultations: Mycin*. American Elsevier, New York.

Shortliffe, E., Scott, A., Bischoff, M., van Melle, C., and Jacobs, W. (1981). ONCOCIN: An Expert System for Oncology Protocol Management. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, 1981.

Smith, B. C. (1982). *Reflections and Semantics in a Procedural Language*. PhD Thesis, Massachusetts Institute of Technology, MIT-TR-272. Prologue reprinted in (Brachman and Levesque, 1985).

Smith, J., W. and Johnson, T., R. (1993). A Stratified Approach to Specifying, Designing, and Building Knowledge Systems. *IEEE Expert*, June 1993.

Stallman, R. M. and Sussman, G. J. (1977). Forward Reasoning and Dependency-directed Backtracking in a System for Computer-aided Circuit Analysis. *Artificial Intelligence, 9*, pp. 135-196.

Steele, G. (1992). *Common Lisp: The Language*. Digital Press.

Steels, L. (1990). Components of Expertise. *AI Magazine, 11(2)*, pp. 29-49.

Stefik M. (1995). *Introduction to Knowledge Systems*. Morgan Kaufmann, San Francisco, CA.

Stout, J., Caplain, G., Marcus, S. and McDermott, J. (1988). Toward Automating Recognition of Differing Problem-Solving Demands. *International Journal of Man-Machine Studies, 29(5)*, pp. 599-611.

Stutt, A. (1997). Knowledge Engineering Ontologies, Constructivist Epistemology, Computer Rhetoric: A *Trivium* for the Knowledge Age. *ED-MEDIA '97*. Calgary, Canada, 14-19 June, 1997.

Stutt, A. and Motta, E. (1997). Knowledge Modelling: The Organic Technology for the Knowledge Age. *Technical Report*, Knowledge Media Institute, The Open University, UK. November 1997.

Swartout, B., Patil, R., Knight, K. and Russ, T. (1996). Toward Distributed Use of Large-Scale Ontologies. In B. Gaines and M. Musen (Editors), *Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop.* Banff, Alberta, Canada.

Terpstra, P. (1994). An environment for application design. *Common KADS Project Deliverable DM7.5a.* University of Amsterdam, The Netherlands.

Tu, S. W., Shahar, Y., Dawes, J., Winkles, J., Puerta, A. R., and Musen, M. A. (1992). A problem solving model for episodic skeletal-plan refinement. *Knowledge Acquisition 4(2)*, pp. 197-216.

Valasek M. and Zdrahal Z. (1997) Experiments with Applying Knowledge Based Techniques to Parametric Design. *ICED 97*. Tampere. Finland.

Valente, A. and Breuker, J. A. (1996). Towards Principled Core Ontologies. In B. Gaines and M. Musen (Editors), *Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop.* Banff, Alberta, Canada.

Valente, A., Breuker, J. A. and Van de Velde, W. (1994). The CommonKADS Expertise Modelling Library. In J. A. Breuker and W. van de Velde, *The CommonKADS Library for Expertise Modelling.* IOS Press, Amsterdam, The Netherlands.

van de Velde, W. (1988). Inference structure as a basis for problem solving. *Proceedings of the 8th European Conference on Artificial Intelligence*, pp. 202-207, London, Pitman.

van de Velde, W. (1994). An Overview of CommonKADS. In J. A. Breuker and W. van de Velde (Editors), *The CommonKADS Library for Expertise Modelling.* IOS Press, Amsterdam, The Netherlands.

van der Spek, R. and de Hoog, R. (1994). Towards a Methodology for Knowledge Management. *Technical Note, Knowledge Management Network*, CIBIT, Utrecht, The Netherlands. http://www.cibit.hvu.nl/web/kmn/pospapers.nsf.

van Harmelen, F. and Balder, J. R., (1992). $(ML)^2$: A Formal Language for KADS Models of Expertiese. *Knowledge Acquisition, 4(1)*, pp. 127-161.

van Heijst, G. (1995). *The Role of Ontologies in Knowledge Engineering.* PhD thesis, University of Amsterdam, May 1995.

van Heijst, G. and Anjewerden, A. (1996). Four Propositions Concerning the Specification of Problem-Solving Methods. In Shadbolt N., O'Hara, K., and Schreiber, G. (Editors), *Supplementary Proceedings of the 9th European Knowledge Acquisition Workshop - EKAW-96.* Nottingham, England, 14-17 May , 1996.

van Heijst, G., Schreiber, A. T. and Wielinga, B. J. (1997). Using explicit ontologies for KBS development. *International Journal of Human-Computer Studies, 46(2/3)*, pp. 183-292.

Van Heijst G., Terpstra P., Wielinga B. and Shadbolt N. (1992). Using Generalized Directive Models in Knowledge Acquisition. In Th. Wetter, K.-D. Althoff, J. Boose, B. R. Gaines, M. Linster and F. Schmalhofer (Editors), *Current Developments in Knowledge Acquisition - EKAW '92.* Lecture Notes in Artificial Intelligence (LNAI) 599, Springer-Verlag, pp.112-32.

van Melle, W., Shortliffe, E. H., and Buchanan, B. G. (1984). EMYCIN: A Knowledge Engineer's Tool for Constructing Rule-Based Expert Systems. In Buchanan, B. G. and Shortliffe, E. H. (Editors), *Rule-Based Expert Systems*, Addison-Wesley, Reading, MA.

Weiss, S. M. and Kulikowski, C. A. (1984). A Practical Guide to Designing Expert Systems. Rowman and Allanheld Publishers.

Weyhrauch, R. W. (1980). Prolegomena to a theory of mechanized formal reasoning, *Artificial Intelligence (13)1-2*, pp. 133-170

Wielinga B. J., Akkermans J. M.. and Schreiber A. T.. (1995). A Formal Analysis of Parametric Design Problem Solving. In B. Gaines and M. A. Musen (Editors), *Proceedings of the 9th Banff Knowledge Acquisition Workshop*, pp. 37-1 - 37- 15.

Wielinga B. J. and Breuker J. A. (1984). Interpretation Models for Knowledge Acquisition. In T. O'Shea (Editor), *Advances in Artificial Intelligence* (ECAI '84, Pisa), North-Holland, Amsterdam.

Wielinga B. J. and Breuker J. A. (1986). Models of Expertise. In B. du Boulay, D. Hoggs and L. Steels (Editors), *Advances in Artificial Intelligence* (ECAI '86, Brighton), North-Holland, Amsterdam.

Wielinga B.J. and Schreiber, A.,Th. (1997). Configuration-design problem solving. *IEEE Expert, 2.* Special issue on AI and design.

Wielinga B. J., Schreiber A.T. and Breuker J., (1992a). KADS: A Modelling Approach to Knowledge Engineering. *Knowledge Acquisition 4(1)*, pp. 5-53.

Wielinga, B., Van de Velde, W., Schreiber, G. and Akkermans, H. (1992b). The CommonKADS Framework for Knowledge Modelling. *Proceedings of the 7th Banff Knowledge Acquisition Workshop.* Banff, Alberta, Canada.

Wiig, K. M. (1994). A Knowledge Management Framework. Practical Approaches to Managing Knowledge. Schema Press, Arlington, Texas.

Winograd, T. and Flores, F. (1986). *Understanding Computers and Cognition.* Ablex Publishing.

Yen, J., Neches, R. and MacGregor, R. (1988). Classification-based Programming: A Deep Integration of Frames and Rules. *Technical Report ISI/RR-88-213.* USC/Information Science Institute, March 1988.

Yost G.R. and Rothenfluh T.R. (1996). Configuring elevator systems. *International Journal of Human-Computer Studies, 44(3/4),* pp. 521-568.

Zdrahal, Z. & Domingue, J. (1997) The World Wide Design Lab: An Environment for Distributed Collaborative Design. In *Proceedings of the 11th International Conference on Engineering Design.* 19-21 August, 1997, Tampere, Finland.

Zdrahal Z., Motta E. (1995). An In-Depth Analysis of Propose & Revise Problem Solving Methods. In B. R. Gaines and M. Musen (Editors), *Proceedings of the 9th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, pp. 38-1 - 38- 20.

Zdrahal Z., Motta E. (1996). Improving Competence by Integrating Case-Based Reasoning and Heuristic Search. In B. Gaines and M. Musen (Editors), *Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop.* Banff, Alberta, Canada.

Appendix 1
Additional details on the OCML language

In this appendix I provide additional information on the OCML language. In particular I describe the informal semantics of the primitive constructors for functional and control terms; the OCML inheritance mechanism; the interpreters and the proof system.

1.1. FUNCTIONAL TERM CONSTRUCTORS

A BNF specification of each term constructor is provided as well as an informal description of its operational semantics.

`Setofall`

setofall-term	::=	`setofall` *template basic-log-expression*
template	::=	`nil` | (*term* . *term*)
term	::=	*constant* | *variable* | *string* | (*fun* {*term*}*) | *findall-term* | *the-term* | *in-env-term* | *quote-term* | *if-term* | *cond-term*
fun	::=	the name of a function or a term constructor
constant	::=	A symbol whose first character is not '?'
variable	::=	A symbol whose first character is '?'
string	::=	A lisp string, e.g. "`string`".
log-expression	::=	*quant-log-expression* | *basic-log-expression*
quant-log-expression	::=	(`forall` *schema-or-var log-expression*)| (`exists` *schema-or-var log-expression*)
basic-log-expression	::=	(`and` {*log-expression*}+) | (`or` {*log-expression*}+) | (`=>` *log-expression log-expression*) | (`<=>` *log-expression log-expression*) (`not` *log-expression*) | *rel-expression*
schema-or-var	::=	*schema* | *variable*
schema	::=	(*variable* . *schema*) | `nil`
rel-expression	::=	(*rel* {*term*}*)
rel	::=	a symbol naming a relation

`setofall` finds all solutions (i.e. *environments*) to *basic-log-expression* and then returns the list obtained by instantiating *template* in all the returned environments, ensuring that

the list contains no duplicates. If no solutions are found then the empty list (i.e. `nil`) is returned.

`Findall`

findall-term ::= `findall` *template basic-log-expression*

`findall` is the same as `setofall` except that it does not remove duplicate solutions.

`The`

the-term ::= `the` *template basic-log-expression*

`the` finds one solution (i.e. environment) to *basic-log-expression* and then returns the list obtained by instantiating *template* in the returned environment. If no solutions are found then the constant `:nothing` is returned.

`In-environment`

in-env-term ::= `in-environment` *pairs body*

pairs ::= `nil` | (*pair* . *pairs*)

pair ::= (*variable* . *term*)

body ::= *term*

The primitive `in-environment` takes a list, possibly empty, of pairs ((var_1 . $term_1$)...) and a *body*, and returns the result of evaluating this in an environment in which each var_i is bound to $term_i$.

`Quote`

quote-term ::= `'`*term* | (`quote` *term*)

The value of an expression such as (`quote` *term*) is *term*.

`If`

if-term ::= (`if` *log-expression then-term* {*else-term*})

then-term ::= *term*

else-term ::= *term*

The first action which is carried out when evaluating an *if-term* is to check whether *log-expression* is satisfied. If this is the case, then *then-term* is evaluated in the environment which satisfies *log-expression*. If *log-expression* cannot be satisfied in the current model, then there are two possibilities. If *else-term* is specified, then this is evaluated, and the value obtained is returned as the value of the *if-term*. If *else-term* is not present and *log-expression* cannot be proved, then the constant `:nothing` is returned.

`Cond`

cond-term ::= (*cond* {*cond-clause*}$^+$)

cond-clause ::= (*log-expression term*)

The interpreter iterates through each clause of a *cond-term*, until it finds one whose *log-expression* is satisfied. If none is found, then `:nothing` is returned. Otherwise, let's assume $cond\text{-}clause_i$. is the first clause whose *log-expression* is satisfied. In this case the value of the *cond-term* is obtained by evaluating the term associated with $cond\text{-}clause_i$.

1.2. CONTROL TERM CONSTRUCTORS

A BNF specification of each control term constructor is provided as well as an informal description of its operational semantics.

In-environment

in-env-control-term ::= `in-environment` *pairs control-body*

pairs ::= `nil` | (*pair . pairs*)

pair ::= (*variable . term*)

control-body ::= *control-term*

control-term ::= *term* | *in-env-control-term* | *if-control-term* | *cond-control-term* | *do-control-term* | *loop-control-term* | *repeat-control-term* | *return-control-term*

if-control-term ::= (`if` *log-expression then-control-term* {*else-control-term*})

then-control-term ::= *control-term*

else-control-term ::= *control-term*

cond-control-term ::= (*cond* {*cond-control-clause*}+)

cond-control-clause ::= (*log-expression control-term*)

loop-control-term ::= (`loop for` *variable* `in` *term* `do` {*control-term*}+)

do-control-term ::= (`do-actions` {*control-term*}+)

repeat-control-term ::= (`repeat-actions` {*end-test*} {*control-term*}+) | (`repeat-actions` {*control-term*}+ {*end-test*})

end-test ::= `while` *test* | `until` *test*

test ::= *log-expression*

return-control-term ::= (`return` *term*)

The primitive `in-environment` takes a list, possibly empty, of pairs ((var_1 . $term_1$)...) and a *control-body*, and returns the result of evaluating this in an environment in which each var_i is bound to $term_i$.

If

if-control-term ::= (`if` *log-expression then-control-term* {*else-control-term*})

The first action which is carried out when evaluating an *if-control-term* is to check whether *log-expression* is satisfied. If this is the case, then *then-control-term* is evaluated in the environment which satisfies *log-expression.* If *log-expression* cannot be satisfied in the current model, then there are two possibilities. If *else-control-term* is specified, then this is evaluated, and the value obtained is returned as the value of the *if-control-term.* If *else-control-term* is not present and *log-expression* cannot be proved, then the constant `:nothing` is returned.

Cond

cond-control-term ::= (*cond* {*cond-control-clause*}+)

The interpreter iterates through each clause of a *cond-control-term*, until it finds one whose *log-expression* is satisfied. If none is found, then `:nothing` is returned. Otherwise, let's assume *cond-clause*$_i$. is the first clause whose *log-expression* is satisfied. In this case the value of the *cond-control-term* is obtained by evaluating the control-term associated with *cond-clause*$_i$.

Loop

loop-control-term ::= (`loop for` *variable* `in` *term* `do`
{*control-term*}+)

The control construct **`loop`** provides a simple mechanism for iterating over lists. It first evaluates a *term*, which should return a list, say *L*. Then it iterates over each element of *L*, say *I*, and evaluates *control-term* in an environment in which *variable* is bound to *I*.

Do

do-control-term ::= (`do-actions` {*control-term*}+)

The control construct `do` is a simple sequencing primitive. The control terms in its body are evaluated sequentially, once only.

Repeat

repeat-control-term ::= (`repeat-actions` {*end-test*} {*control-term*}+) |
(`repeat-actions` {*control-term*}+ {*end-test*})

The control term constructor `repeat` repeats the control term(s) specified in its body until the end test is satisfied, if the test has the form '`until` test'. Otherwise, if the test has the form '`while` *test*', then `repeat-actions` stops as soon as the test fails. If the end test is specified after the control terms, then the control terms are carried out at least once - i.e. the end test is verified at the end of each cycle. If the end test is specified before the control terms, then the test is verified at the beginning of each cycle. If no test is provided, then all control expression in the body of a `repeat-actions` are repeated *ad infinitum*.

Return

return-control-term ::= (`return` *term*)

This is a simple way of exiting from the body of a `loop` or `repeat` construct. When a control term such as (`return` *term*) is encountered, the most specific loop or repeat construct in the current execution stack is exited and the value obtained from evaluating *term* is returned.

1.3. INHERITANCE AND DEFAULT VALUES

Generally speaking default values are values which apply unless other alternatives can be used. In the OCML language the notion of default value is operationalized as follows.

Instances inherit values and default values from their superclasses down the inheritance hierarchy specified by `instance-of` and `subclass-of` links. For a given slot, say `s`, of a sample instance, say `I`, the following scenarios can arise:

i) `I` has not inherited any default value. In this case the value of `s` in `I` is given by all the values `I` has inherited from its superclasses, plus any value locally specified for slot `s` of `I`.

ii) `I` has inherited some default values as well as non-default ones. In this case the default values are ignored and rule (i) is applied. We say that the default values are *overridden* by the non-default ones.

iii) `I` has inherited only default values and local values have been specified. As in the previous case, the default values are ignored and only the local values are considered.

iv) `I` has inherited only default values and no local values have been specified. In this case there are two possibilities. If the `:inheritance` facet has not been specified, or it has been specified and it is `:merge`, all default values apply. If the `:inheritance` facet has been specified and it is `:supersede`, then the value of `s` in `I` is obtained by (i) ranking the *ancestors* of `I` according to the *class precedence order* of the *parent* of `I`, and (ii) retrieving the default value of the first class in the class precedence order which specifies a (default) value for `s`. The details of the algorithm used to compute the class precedence order are given at pp. 782-786 of the Common Lisp specification (Steele, 1992). This algorithm produces a total order (if this exists) based on two ordering principles: (i) a class, say `C`, precedes all its direct superclasses, and (ii) a direct superclass of `C` precedes the direct superclasses of `C` specified to its right in the list of direct superclasses of `C`.

1.4. INTERPRETERS AND PROOF SYSTEM

1.4.1. The OCML interpreter for functional terms

The OCML interpreter is implemented by means of a Lisp macro, `ocml-eval`.

The evaluation of a functional term, *term*, in an environment, *env*, is carried out according to the following rules.

i) If *term* is a variable, then the binding of *term* in *env* is returned.

ii) If *term* is a string or a constant, then *term* is returned.

iii) If *term* has the format (*pfun* $term_0$, ..., $term_n$), with $n \geq 0$, where *pfun* is a *primitive term constructor*, then *term* is evaluated in *env*, according to criteria which depend on *pfun*.

iv) If *term* has the format (*fun* $term_0$, ..., $term_n$), with $n \geq 0$, where fun is the name of a function, and a Lisp body associated with *fun* exists, then `ocml-eval` returns the value obtained by applying the Lisp body to the values obtained by evaluating each $term_i$ in *env*.

v) If *term* has the format (*fun* $term_0$, ..., $term_n$), with $n \geq 0$, where *fun* is the name of a function, and no Lisp body associated with *fun* exists, then `ocml-eval` returns the value obtained by applying the body of *fun* to the values obtained by evaluating each $term_i$ in *env*.

vi) In all other cases `ocml-eval` signals an error.

1.4.2. The OCML interpreter for control terms

Control terms are interpreted in a manner analogous to functional terms. The control term interpreter is implemented by a Lisp macro, `ocml-control-eval`, which has the following behaviour.

i) If *term* is a functional term, then it is evaluated according to the rules given in section 4.5.1.

ii) If *term* has the format (*proc* $term_0$, ..., $term_n$), with $n \geq 0$, where *proc* is a *primitive control operator*, then *term* is evaluated in *env*, according to criteria which depend on *proc*.

iii) If *term* has the format (*proc* $term_0$, ..., $term_n$), with $n \geq 0$, where *proc* is the name of a procedure, and a Lisp body associated with *proc* exists, then `ocml-control-eval` returns the value obtained by applying the Lisp body to the values obtained by evaluating each $term_i$ in *env*.

iv) If *term* has the format (*proc* $term_0$, ..., $term_n$), with n ≥ 0, where *proc* is the name of a procedure, and no Lisp body associated with *proc* exists, then `ocml-control-eval` returns the value obtained by applying the body of *proc* to the values obtained by evaluating each $term_i$ in *env*.

vi) In all other cases `ocml-control-eval` signals an error.

1.4.3. The OCML proof system

1.4.3.1. Procedure for proving basic goal expressions in OCML

Let's suppose we want to find all solutions to a basic *goal expression*, say *G*, with format (*rel* {*fun-term*}*), where *rel* is the name of a relation and *fun-term* a functional term. In general we might be interested in one, some or all solutions. Therefore the order in which solutions are generated might be important. The algorithm used by the OCML proof system is as follows.

1. If *rel* is not a defined relation, then signal an error. Otherwise initialize *SOL1*, *SOL2*, *SOL3*, *SOL4*, *SOL5* and *SOL6* to the empty set and go to step 2.
2. Retrieve all the assertions present in the current model, whose type (i.e. first element) is *rel*. Match each assertion with *G*. All successful matches, we call this set *SOL1*, provide solutions to *G*. Go to step 3.
3. If a Lisp attachment exists for *rel*, then evaluate it in the Lisp environment to find eventual additional solutions to *G*, say *SOL2*. Go to step 8.
4. If a `:prove-by` proof condition, say *prove-rel-expression*, has been specified for relation *rel*, then compute all solutions to *prove-rel-expression*, say *SOL3*. Each of these is also a solution to *G*. Go to step 8.
5. If a `:iff-def` proof condition, say *iff-rel-expression*, has been specified for relation *rel*, then compute all solutions to *iff-rel-expression*, say *SOL4*. Each of these is also a solution to *G*. Go to step 8.
6. If a `:sufficient` proof condition, say *suff-rel-expression*, has been specified for relation *rel*, then compute all solutions to *suff-rel-expression*, say *SOL5*. Each of these is also a solution to *G*. Go to step 7.
7. If a backward chaining rule has been specified, associated with relation *rel*, then invoke it to find all other solutions to *G*, say *SOL6*. Go to step 8.
8. The set of all solutions to query *G* is obtained by appending the lists *SOL1*, *SOL2*, *SOL3*, *SOL4*, *SOL5* and *SOL6*.

The algorithm shown above provides an operational semantics for the various relation-forming constructs provided by OCML. In particular the following two points should be highlighted.

- Assertions inherited through an isa hierarchy are always cached at definition time. This means that they are retrieved at step 2, when the goal is matched against the current set of known facts.
- The results returned by non-logical mechanisms such as Lisp attachments and `:prove-by` are only merged with the results obtained by simple assertion-matching (step 2). In other words they are meant to provide efficient proof mechanisms which override those provided by definition-forming options, such as `:iff-def` and `:sufficient`.

1.4.3.2. Proof rules for non-basic goal expressions

The bullet points below describe how non-basic goal expressions are proven in OCML.

- `(and` *A B*`)`. This expression is satisfied if both *A* and *B* can be proven in the current model.
- `(or` *A B*`)`. This expression is satisfied if either *A* or *B* can be proven in the current model.

- (=> *A B*). This expression is satisfied if either *A* cannot be proven, or, if *B* can be proven in each environment which is a solution to *A* .
- (<=> *A B*). This expression is satisfied if both (=> *A B*) and (=> *B A*) can be proven.
- (`not` *A*). This expression is satisfied if *A* cannot be proven in the current model.
- (`exists` *schema-or-var A*). This expression is satisfied if *A* can be proven in the current model.
- (`forall` *schema-or-var* (=> *A B*)). This expression is satisfied if either *A* cannot be proven, or, if *B* can be proven in each environment which is a solution to *A*.

Thus, the proof mechanism supported by OCML is not complete with respect to first-order logic statements. In particular, disjunctions can only be proved by proving each clause separately and negated expressions are only proved by default.

Appendix 2
Full specification of the task-method ontology

This appendix provides a complete specification of the task-method ontology.

```
(in-ontology base-ontology)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;Task-related definitions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(def-class  TASK () ?task
  "An OCML task is characterised by its
   input roles, output role, and goal. The goal expression is a
   kappa expression which takes as argument the task itself and a
   value (which is meant to be a possible result from carrying out the
   task.  The goal is satisfied if the kappa expression holds for its
   two arguments.
   A role is a slot of a task.
   Tasks divide into two main subclasses:
   goal-specification-task and executable-task.   The former
   provides only a goal specification, while the latter provides
   also an 'organic' method for achieving the task"
  ((has-input-role :type role)
   (has-output-role :type role)
   (has-goal-expression :type legal-task-goal-expression))
  :constraint (=> (has-role ?task ?role)
                  (and (slot-of ?role ?task)
                       (functional-relation ?role)))
  :axiom-def (exhaustive-subclass-partition
              task
              (set-of goal-specification-task
                      executable-task))
  :lisp-class-name task)

(def-class TASK-TYPE () ?x
  :iff-def (subclass-of ?x task))

(def-class LEGAL-TASK-GOAL-EXPRESSION (kappa-expression) ?exp
  "A task goal expression is a kappa expression with arity 2,
   which does not contain free variables.  The first argument to
   the kappa expression represents a task-instance, the second
   the result of the task"
  :iff-def (and (= (arity ?exp) 2)
                (= ?exp (kappa ?schema ?sent))
                (= ?vars (all-free-vars-in-sentence ?sent))
                (= (length ?vars) 2)
                (member (namestring (first ?vars))
                        (map namestring ?schema))
                (member (namestring (second ?vars))
                        (map namestring ?schema))))
```

```
(def-class GOAL-SPECIFICATION-TASK (task) ?task
  "A goal-specification-task is a task with a goal
   expression and no body"
  ((has-goal-expression :cardinality 1))
  :constraint (not (exists ?body
                          (has-body ?task ?body))))

(def-class PROBLEM-TYPE (goal-specification-task)
  "A problem type is a goal specification task which defines a
   generic class of applications - e.g. parametric design")

(def-class EXECUTABLE-TASK (task)
  "An executable task is a task with a body - i.e. a
   task whose specification also includes a mechanism for
   achieving it"
  ((has-body :type unary-procedure))
  :axiom-def (exhaustive-subclass-partition
              executable-task
              (set-of primitive-task
                      composite-task))
  :lisp-class-name executable-task)

(def-class COMPOSITE-TASK (executable-task) ?task
  "A composite task is a task which introduces a subtask
   decomposition.  Something is an instance of this task if
   its parent introduces a generic task-subtask
   decomposition"
  :iff-def (and (direct-instance-of ?task ?c)
                (has-generic-subtasks ?c ?subs)))

(def-relation HAS-GENERIC-SUBTASKS (?task-type ?subs)
  "Use this to model generic task-subtask decompositions"
  :constraint (and (subclass-of ?task-type composite-task)
                   (every ?subs task-type)))

(def-class PRIMITIVE-TASK (executable-task) ?task
  "An executable task which is not a composite task"
  :iff-def (not (exists (?c ?subs)
                        (and (direct-instance-of ?task ?c)
                             (has-generic-subtasks ?c ?subs)))))

(def-procedure ACHIEVE-GENERIC-SUBTASK (?supertask
                                        ?task-type
                                        &rest ?actual-role-pairs)
  :body (in-environment ((?name . (new-symbol ?task-type)))
          (tell (append (list-of ?task-type ?name)
                        ?actual-role-pairs))
          (tell (subtask-of ?name ?supertask))
          (solve-task ?name)))

(def-procedure INSTANTIATE-GENERIC-SUBTASK
  (?supertask ?task-type &rest ?actual-role-pairs)
  :body (in-environment
         ((?name . (new-symbol ?task-type)))
         (tell (append (list-of ?task-type ?name)
                       ?actual-role-pairs))
         (tell (subtask-of ?name ?supertask))
         ?name))
```

```
(def-relation ACHIEVED (?task-inst ?result)
  "A task has been achieved if its goal holds in the current model.
   A method has been achieved either if its goal has been achieved or
   if its associated task has."
  :iff-def (or (and (= ?exp (role-value ?task-inst has-goal-
expression))
                    (holds ?exp ?task-inst ?result))
               (and (problem-solving-method ?task-inst)
                    (tackles-task ?task-inst ?task-inst2)
                    (achieved ?task-inst2 ?result))))

(def-procedure PERFORM-EXECUTABLE-TASK (?task-instance)
  :body (if (has-body ?task-instance ?body)
          (execute-task-body ?body ?task-instance)))

;;;This is the same as PERFORM-EXECUTABLE-TASK..here only for
;;;compatibility with the previous versions of the ontology
(def-procedure EXECUTE-PRIMITIVE-TASK (?task-instance)
  :body (if (has-body ?task-instance ?body)
          (execute-task-body ?body ?task-instance)))

(def-procedure EXECUTE-TASK-BODY (?body ?task-instance)
  :body (call ?body ?task-instance)
  :lisp-fun #'execute-task-body)

(def-relation SUBTASK-OF (?inst1 ?inst2)
  "This relation is used to model the specific
   task-subtask hierarchy constructed at
   execution time"
  :constraint (and (executable-task ?inst1)
                   (task ?inst2))
  :sufficient (and (problem-solving-method ?inst1)
                   (tackles-task ?inst1 ?inst2)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;Role-related definitions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(def-class ROLE  (slot) ?role
  "A role is a binary relation associated with a task by
   means of the 'has-role' relation.  The value cardinality
   of a role-defining slot is 1."
  :constraint (forall (?i)
                      (=> (and (has-role ?class ?role)
                               (instance-of ?i ?class))
                          (has-one ?i ?role)))
  :iff-def (exists ?c
                   (and (task-type ?c)
                        (has-role ?c ?role))))
```

```
(def-relation HAS-INPUT-ROLE (?class ?role)
  "This definition generalises the notion of
   'having an input role' to classes as well
    as tasks instances.  If ?class is a method, then
    it also 'inherits the input roles from the task type
    to which it is applicable"
  :sufficient (and (subclass-of ?class task)
                   (or (and (slot-of has-input-role ?class)
                            (member ?role (all-class-slot-values
                                           ?class has-input-role)))
                       (and (subclass-of ?class
                                          problem-solving-method)
                            (member ?task-type (all-class-slot-values
                                                 ?class
                                                 tackles-task-type))
                            (has-input-role ?task-type ?role)))))

(def-relation HAS-OUTPUT-ROLE (?class ?role)
  "This definition generalises the notion of
   'having an output role' to classes as well
    as tasks instances.  If ?class is a method, then
    it also 'inherits the output role from the task type
    to which it is applicable"
  :sufficient (and (subclass-of ?class task)
                   (or (and (slot-of has-output-role ?class)
                            (member ?role (all-class-slot-values
                                           ?class has-output-role)))
                       (and (subclass-of ?class
                                          problem-solving-method)
                            (member ?task-type (all-class-slot-values
                                                 ?class
                                                 tackles-task-type))
                            (has-output-role ?task-type ?role)))))

(def-relation HAS-CONTROL-ROLE (?thing ?role)
  :sufficient (or (and (instance-of ?thing composite-task)
                       (has-control-role (the-parent ?thing) ?role))
                  (and (subclass-of ?thing composite-task)
                       (or
                        (member ?role (all-class-slot-values
                                       ?thing has-control-role))
                        (and (has-generic-subtasks ?thing ?subs)
                             (member ?sub ?subs)
                             (has-output-role ?sub ?role))))))

(def-relation HAS-ROLE (?thing ?role)
  "Generalises from input output and control roles"
  :iff-def (or (and (task ?thing)
                    (has-role (the-parent ?thing) ?role))
               (and (task-type ?thing)
                    (member ?role
                            (union (setofall ?r (has-input-role
                                                 ?thing ?r))
                                   (setofall ?r (has-control-role
                                                 ?thing ?r))
                                   (setofall ?r (has-output-role
                                                 ?thing ?r)))))))

(def-function TASK-ROLES (?class) -> ?roles
  :constraint (and (task-type ?class)
                   (every ?roles role))
  :body (setofall ?x (has-role ?class ?x)))
```

```
(def-function LOCAL-ROLE-VALUE (?task ?role)

  "The local value of a role, say ?role, in a task, say ?task,
   is defined as the value of the slot ?role in ?task.
   If the max cardinality of ?role in ?task is either
   undefined or defined and greater than 1, then the set of all
   slot values are returned.  Otherwise only one is returned"

  :body (the ?v (holds ?role ?task ?v)))

(def-function ROLE-VALUE (?task ?role)
  "The value of a role is its local value if it exists.
   If it does not then the subtask-of hierachy is searched
   for a value."
  :body (in-environment ((?value . (local-role-value
                                     ?task ?role)))
          (if (and (= ?value :nothing)
                   (subtask-of ?task ?supertask))
            (role-value ?supertask ?role)
            ?value)))

(def-procedure SET-ROLE-VALUE (?task ?role ?value)
  :body (set-slot-value ?task ?role ?value))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;Problem Solving Methods
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(def-class PROBLEM-SOLVING-METHOD (executable-task)
  "A problem solving method is an executable task which
   is associated with (tackles) a class of tasks.
   The slot has-output-mapping specifies a function which maps
   the result returned by the method to a task.
   The reason for this mapping is to allow the decoupling
   of the type of result returned by the method from that expected
   by the task.  This provides greater flexibility and also makes it
   possible to specify solution conditions for a method which use
   different types of output from that used by a task"
  ((tackles-task :type goal-specification-task)
   (has-output-mapping
    :value '(lambda (?psm ?result)
             ?result)))
  :lisp-class-name problem-solving-method)

(def-relation TACKLES-TASK-TYPE (?method-class ?task-type)
  "This relation provides a fairly coarse-grained indexing of the
   library: each method is associated to a class of tasks to which
   it can be applied"
  :constraint (and (subclass-of ?method-class problem-solving-method)
                   (task-type ?task-type)))

(def-relation APPLICABILITY-CONDITION (?method-class ?exp)
  "This relation provides a more fine-grained test to check
   the applicability of a class of methods to a specific task"
  :constraint (and (subclass-of ?method-class problem-solving-method)
                   (unary-relation ?exp)))

(def-relation APPLICABLE-TO-TASK (?method-class ?task-inst)
  :iff-def (or (not (applicability-condition ?method-class  ?exp))
               (and (applicability-condition ?method-class  ?exp)
                    (holds ?exp  ?task-inst))))
```

```
(def-class PRIMITIVE-METHOD (problem-solving-method primitive-task)
  "A method with no subtasks")

(def-class DECOMPOSITION-METHOD (problem-solving-method composite-
task)
  "A method with subtasks")

(def-procedure SOLVE-TASK (?task-instance)
  "A task, ?task, is executed by evaluating its body
   in an environment in which the schema of the class
   corresponding to the parent of ?task is bound to
   ?task."
  :body (if (executable-task ?task-instance has-body ?body)
          (execute-task-body ?body ?task-instance)
          (if (= ?best-psm
                 (choose-best-method-class
                  (setofall ?psm-type
                            (and
                             (subclass-of ?psm-type
                                          problem-solving-method)
                             (tackles-task-type ?psm-type  ?c)
                             (instance-of ?task-instance ?c)
                             (applicable-to-task ?psm-type
                                                 ?task-instance)))))

            (in-environment
             ((?method . (instantiate-generic-subtask
                          ?task-instance ?best-psm)))
             (apply-method-to-task ?method ?task-instance)))))

(def-function CHOOSE-BEST-METHOD-CLASS (?psm-types)
  :body (if (null ?psm-types)
          :nothing
          (if (= (length ?psm-types) 1)
            (first ?psm-types)
            (if (exists ?x
                        (and
                         (member ?x ?psm-types)
                         (use-method ?x ?c ?m)))
              (choose-from-use-method-statements ?psm-types)
              (first ?psm-types)))))
```

```
(def-function CHOOSE-FROM-USE-METHOD-STATEMENTS (?psm-types)
  :body (if (and (use-method ?x ?c (the-current-task))
                 (member ?x ?psm-types))
          ?x
          (in-environment
           ;;try to pick the most specific use-method
           ;;statement for this subtask
           ((?psm-type . (the ?x
                              (and
                               (member ?x ?psm-types)
                               (use-method ?x ?c ?m)
                               (instance-of (the-current-method) ?m)
                               (not
                                (exists
                                 (?m2 ?x2)
                                 (and (member ?x2 ?psm-types)
                                      (use-method ?x2 ?c ?m2)
                                      (subclass-of ?m2 ?m)
                                      (instance-of
                                       (the-current-method)
                                       ?m2))))))))
           (if (= ?psm-type :nothing)
             (first ?psm-types)
             ?psm-type))))

(def-relation use-method (?sub-method ?sub-task ?thing)
  "Use instances of this relation to specify which sub-method
   to use when solving a generic subtask of a problem. The third
   argument can be used to contextualise this statement within a
   problem solving method or a particular problem.
   EXAMPLE: (use-method HC-CONTROL DESIGN-FROM-STATE HC-DESIGN)"
  :constraint (and (subclass-of ?sub-method problems-solving-method)
                   (subclass-of ?sub-task task)
                   (or (task ?thing)
                       (subclass-of ?thing problem-solving-method))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;Applications
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(def-class APPLICATION-DOMAIN ())

;;;APPLICATION
(def-class application ()
  ((tackles-domain :type application-domain)
   (uses-method :type problem-solving-method)
   (tackles-task :type goal-specification-task))
  :lisp-class-name application)

(def-procedure SOLVE-APPLICATION (?appl)
  :body (if (application ?appl
                         uses-method ?method-inst
                         tackles-task ?task-inst)
          (do
            (unassert (current-application ?any))
            (tell (current-application ?appl))
            (apply-method-to-task ?method-inst ?task-inst))))

(def-function the-current-task ()
  :body (if (current-application ?appl)
          (the ?x (tackles-task ?appl ?x))))
```

```
(def-function the-current-method ()
  :body (if (current-application ?appl)
          (the ?x (uses-method ?appl ?x))))

(def-procedure APPLY-METHOD-TO-TASK (?method-inst ?task-inst)
  :body (do
          (tell (tackles-task ?method-inst ?task-inst))
          (in-environment
           ((?output-role . (the-slot-value
                             ?task-inst
                             has-output-role))
            (?fun . (the ?fun (has-output-mapping ?method-inst ?fun)))
            (?result . (execute-primitive-task ?method-inst)))
           (set-slot-value ?task-inst
                           ?output-role
                           (call ?fun ?method-inst ?result))
           ?result)))
```

Appendix 3
Full specification of the parametric design ontology

This appendix provides a complete specification of the parametric design task ontology.

```
(def-ontology parametric-design
    "This ontology defines parametric design tasks, which are design
     tasks where the solution is expressed in terms of an assignment
     of values to parameters.")

(in-ontology parametric-design)

;;; DESIGN-TASK ---This is just a token class.  We do not characterize
;;;it in this ontology
(def-task design-task (goal-specification-task))

;;;PARAMETRIC-DESIGN
(def-task parametric-design (design-task) ?task
  ((has-input-role :value has-parameters
                   :value has-constraints
                   :value has-requirements
                   :value has-cost-function
                   :value has-cost-algebra
                   :value has-preferences)
   (has-output-role :value has-design-model :cardinality 1)
   (has-design-model :type design-model :max-cardinality 1)
   (has-parameters :type list :cardinality 1)
   (has-constraints :type list :max-cardinality 1)
   (has-requirements :type list :max-cardinality 1)
   (has-preferences :type list :max-cardinality 1)
   (has-cost-function :type cost-function :max-cardinality 1)
   (has-cost-algebra :default-value '(+ - <) :cardinality 1)

   (has-goal-expression
    :type legal-parametric-design-goal
    :default-value  (kappa (?task ?design-model)
                           (design-model-solution
                            ?design-model
                            ?task))))

(def-class LEGAL-PARAMETRIC-DESIGN-GOAL (?rel)
  :iff-def (and (binary-relation ?rel)
                (subrelation-of ?rel
                                (inverse design-model-solution))))

(def-class PARAMETER  () ?p
  "A parameter is something which plays the role of 'parameter'
   in a parametric design task"
  ((has-value-range :type set))
  :iff-def (exists ?task (and (parametric-design ?task)
                              (member ?p (has-parameters ?task ?l)))))
```

```
(def-relation HAS-VALUE (?p ?v ?dm)
  "Parameters have values w.r.t a particular design model"
  :iff-def (and (parameter ?p)
                (design-model ?dm)
                (element-of (?p . ?v) ?dm))
  :constraint (or (not (exists ?vr
                               (has-value-range ?p ?vr)))
                  (element-of ?v ?vr))
  :prove-by (element-of (?p . ?v) ?dm))

(def-relation BOUND-PARAMETER (?x ?dm)
   "True if ?x has a value in ?dm"
   :iff-def (exists ?v (has-value  ?x ?v ?dm)))

(def-relation UNBOUND-PARAMETER (?x ?dm)
   "True if ?x has not a value"
   :iff-def (not (bound-parameter ?x ?dm)))

(def-function PARAMETER-VALUE (?x ?dm)
  :constraint (and (parameter ?x)
                   (design-model ?dm))
  :body (the ?value (has-value ?x ?value ?dm)))

(def-class DESIGN-PRESCRIPTION () ?c
  "The definitions commmon to constraints and requirements.
   A design prescription is characterised in terms of the associated
   expression.  This is a kappa expression predicated over a design
   model"
  ((applicability-condition :default-value (kappa (?d)
                                                   (true))
                            :type legal-prescriptive-expression)
   (has-expression :cardinality 1
                   :type legal-prescriptive-expression)))

(def-relation DESIGN-PRESCRIPTION-APPLIES (?c ?dm)
  :iff-def (holds (the ?x (applicability-condition ?c ?x))
                  ?dm))

(def-class LEGAL-PRESCRIPTIVE-EXPRESSION ()
  ?exp
  "This is an expression parametrized over one argument, which denotes
   a design model"
  :iff-def (and (kappa-expression ?exp)
                (== ?exp (kappa ?schema ?sentence))
                (= ?vars (all-free-vars-in-sentence ?sentence))
                (= (length ?vars) 1)
                (= (length ?schema) 1)
                (= (namestring (first ?schema))
                   (namestring (first ?vars)))))

(def-relation design-model-satisfies (?dm ?c)
   :constraint (and (design-prescription ?c))
   :iff-def (holds (the ?x
                        (has-expression ?c ?x))
                   ?dm))

(def-relation design-model-violates (?dm ?c)
   :constraint (design-prescription ?c)
   :iff-def
                (not (holds (the ?x
                                 (has-expression ?c ?x))
                            ?dm)))
```

```
;;;CLASS CONSTRAINT
(def-class constraint (design-prescription)
  :lisp-class-name constraint)

(def-class REQUIREMENT (design-prescription)
  "A requirement is characterised in the same way as a constraint.
   The difference here is conceptual, rather than logical"
  :lisp-class-name requirement)

(def-class COST-FUNCTION (unary-function) ?cf
  "A cost criterion is a function which takes a design model and
   returns its cost.  The output can be either a real number or a
   vector"
  :iff-def (and (domain ?cf design-model)
                (range ?cf cost)))

(def-class COST () ?x
  "The costs I use are always real numbers of vectors.
   This definition leaves other possibilities open"
  :sufficient (or (real-number ?x)
                  (vector ?x)))

(def-relation HAS-COST-ORDER-RELATION (?pardes-task ?rel)
  :iff-def (= ?rel (third (has-cost-algebra ?pardes-task ?alg))))

(def-relation HAS-COST-DIFFERENCE-FUNCTION (?pardes-task ?rel)
  :iff-def (= ?rel (second (has-cost-algebra ?pardes-task ?alg))))

(def-relation HAS-COST-SUM-FUNCTION (?pardes-task ?rel)
  :iff-def (= ?rel (first (has-cost-algebra ?pardes-task ?alg))))


(def-relation CHEAPER-DESIGN (?rel ?dm1 ?dm2)
  "A design model, ?dm1, is cheaper than ?dm2 according to a cost
   order relation, ?rel, if (?rel ?dm1 ?dm2) is provable."
  :constraint (and (order-relation ?rel)
                   (design-model ?dm1)
                   (design-model ?dm2))
  :iff-def (holds ?rel1 ?dm1 ?dm2))

(def-function ADD-VECTOR-COSTS (?c1 &rest ?other-costs)
  :constraint (and (= (length ?c1) ?n)
                   (every ?other-costs (kappa (?c)
                                         (= (length ?c) ?n))))
  :body (if (null ?c1)
          nil
          (cons (apply +  (map  first (cons ?c1 ?other-costs)))
                (apply add-vector-costs
                       (map  rest (cons ?c1 ?other-costs))))))

(def-function SUBTRACT-VECTOR-COSTS (?c1 &rest ?other-costs)
  :constraint (and (= (length ?c1) ?n)
                   (every ?other-costs (kappa (?c)
                                         (= (length ?c) ?n))))
  :body (if (null ?c1)
          nil
          (cons (apply -  (map  first (cons ?c1 ?other-costs)))
                (apply subtract-vector-costs
                       (map  rest (cons ?c1 ?other-costs))))))
```

```
(def-relation CHEAPER-VECTOR-COST (?c1 ?c2)
  :iff-def (and (not (null ?c1))
                (not (null ?c2))
                (or (< (first ?c1)
                       (first ?c2))
                    (cheaper-vector-cost (rest ?c1)(rest ?c2)))))

(def-class PREFERENCE () ?p
  "A preference defines an order over two design models.  The
   difference between a preference and a constraint or requirement is
   that these distinguish good from bad models, while preferences
   distinguish between better and worse models."
  ((has-expression :cardinality 1 :type prefer-expression)))

(def-class PREFER-EXPRESSION (proof-expression) ?exp
  "A prefer expression is a backward rule clause which tries to
   prove a prefer relation instance"
  ((proves-relation :value prefer))
  :constraint (and (== ?exp (?tail if . ?rest))
                   (== ?tail (prefer ?d1 ?d2))))

(def-relation PREFER (?d1 ?d2)
  "Use this relation to express preferences between design models"
  :constraint (and (design-model ?d1)(design-model ?d2))
  :axiom-def (defines-partial-order prefer))

(def-axiom COST-SUBSUMES-PREFERENCES
  "This axiom states that teh cost function subsumes
   each preference. That is, teh cost function is
   constructed by 'combining' preference-specific
   cost functions"
  (forall (?d1 ?d2) =>
   (and (parametric-design ?task has-preferences ?prs
                          has-cost-function ?cf)
        (has-cost-order-relation ?task ?rel)
       (member ?pr ?prs)
       (has-expression ?pr ?exp)
       (proves ?exp `(prefer ?d1 ?d2)))
   (cheaper-design ?rel ?d1 ?d2)))

(def-axiom COST-PREFERENCES-CONSISTENCY
  "This axiom states that the cost function should not
   contradict any pratial order expressed by preferences"
  (forall (?d1 ?d2)
         (=>
          (and (parametric-design ?task has-preferences ?prs
                                  has-cost-function ?cf)
               (has-cost-order-relation ?task ?rel)
               (cheaper-design ?rel ?d1 ?d2))
          (not (exists ?pr
                       (member ?pr ?prs)
                       (has-expression ?pr ?exp)
                       (proves ?exp `(prefer ?d2 ?d1)))))))
```

```
(def-class DESIGN-MODEL (set) ?d
  "A design model is defined as a functional set of parameter-
   assignments.
   Thus a design model is associated with a binary membership
   function, whose first arg is a parameter, and the second is its
   value in the design model"
  ((membership-test :type design-model-membership-relation
                    :max-cardinality 1))
  :iff-def (and (= ?pairs (setofall ?pair
                                    (element-of ?pair ?d)))
                (every ?pairs parameter-assignment)))

;;;DESIGN-MODEL-MEMBERSHIP-RELATION
(def-class design-model-membership-relation (binary-relation)
  ((domain :value parameter)
   (range :value legal-value)))

(def-class DESIGN-MODEL-TYPE () ?c
  :iff-def (subclass-of ?c design-model))

(def-class PARAMETER-ASSIGNMENT () ?pair
  "A parameter assignment is a pair (?p . ?v),
   where ?p is a parameter"
  :iff-def (and
                (== ?pair (?p . ?v))
                (parameter ?p)
                (legal-value ?v)))

(def-class LEGAL-VALUE () ?v
  "This is a weak definition of legal value. Anything which does not
   contain variables will be allowed as value.  Of course, specific
   design applications might want to specialise this by restricting
   the legal values for each parameter"
  :iff-def (ground-thing ?v))

(def-relation OPTIMAL-SOLUTION (?dm1 ?task)
  "A solution design model, ?dm1, is an optinal solution to a
   parametric design task, ?task, if there is no other solution design
   model, ?dm2, which is cheaper - according to the task cost function
   - than ?dm1"
  :iff-def (and (design-model-solution ?dm1 ?task)
                (not (exists ?dm2
                             (and (design-model-solution ?dm2 ?task)
                                  (has-cost-order-relation
                                   ?task  ?rel)
                                  (cheaper-design ?rel ?dm2 ?dm1))))))

(def-relation DESIGN-MODEL-SOLUTION (?dm ?task)
  :iff-def (and (design-model-complete
                 ?dm
                 (role-value ?task has-parameters))
                (design-model-valid
                 ?dm
                 (role-value ?task has-constraints)
                 (role-value ?task has-requirements))))

(def-relation DESIGN-MODEL-COMPLETE (?dm ?parameters)
   "A design model is complete if all the parameters are bound"
   :iff-def (not (exists ?x
                         (and (member ?x  ?parameters)
                              (unbound-parameter ?X ?dm)))))
```

```
(def-relation DESIGN-MODEL-VALID (?dm ?constraints ?reqs)
  :iff-def (and (design-model-consistent  ?dm ?constraints)
                (design-model-suitable  ?dm ?reqs)))

(def-relation DESIGN-MODEL-CONSISTENT (?dm ?constraints)
   "A design model is consistent if no constraint is violated"
   :iff-def (not (exists ?x
                       (and (member ?x  ?constraints)
                            (design-model-violates ?dm ?x)))))

(def-relation DESIGN-MODEL-SUITABLE (?dm ?reqs)
  "A design model is suitable is all requirements are
   applicable and satisfied.  ?dm is assumed to be complete"
  :constraint (design-model-complete ?dm ?reqs)
  :iff-def (every ?reqs
                (kappa (?req)
                       (and (design-prescription-applies ?req ?dm)
                            (design-model-satisfies ?dm ?req)))))

(def-relation DESIGN-MODEL-EXTENDS (?dm1 ?dm2)
  "A design model, ?dm1, extends a design model, ?dm2,
   if every parameter bound in dm2 is also bound
   in ?dm1 and there is some parameter which is bound
   in ?dm1 and unbound in ?dm2"
  :iff-def (and (forall ?p
                       (=> (bound-parameter ?p ?dm2)
                           (bound-parameter ?p ?dm1)))
                (exists ?p2
                       (and (bound-parameter ?p2 ?dm1)
                            (unbound-parameter ?p2 ?dm2)))))
```

Appendix 4
Full specification of gen-design-psm

This appendix provides a complete specification of the most generic method in the library. This method acts as a kind of 'generic method template', from which all the other methods in the library can be constructed through a method specialization process.

```
(def-class PROBLEM-SOLVING-METHOD-FOR-PARAMETRIC-DESIGN
  (problem-solving-method)
  :own-slots ((tackles-task-type  parametric-design)))

(def-class GEN-DESIGN-PSM
  (problem-solving-method-for-parametric-design
   decomposition-method)
  ((has-input-role :value has-design-operators)
   (has-output-role :value has-solution-state)
   (has-solution-state :type design-state)
   (has-design-operators :type design-operator)
   (has-output-mapping
    :value '(lambda (?psm ?state)
              (the ?dm
                (has-design-model ?state ?dm))))
   (has-body :value
             '(lambda (?psm)
                (in-environment
                 ((?s . (achieve-generic-subtask
                         ?psm gen-design-control
                         has-current-pardes-task (the ?task
                                                  (tackles-task
                                                   ?psm ?task)))))
                 (if (design-state ?s)
                   ?s)))))
  :own-slots ((has-generic-subtasks '(gen-design-control))))
```

```
(def-class GEN-DESIGN-CONTROL (composite-task)
  ((has-input-role :value has-design-operators
                   :value has-current-pardes-task)
   (has-output-role :value has-solution-state)
   (has-solution-state :type design-state)
   (has-design-operators :type design-operator)
   (has-current-pardes-task :type parametric-design)
   (has-body :value
            '(lambda (?psm)
               (in-environment
                ((?design-space . (achieve-generic-subtask
                                   ?psm initialise-design-space
                                   has-current-pardes-task
                                   (role-value
                                    ?psm
                                    has-current-pardes-task))))
                (REPEAT
                 (in-environment
                  ((?state . (achieve-generic-subtask
                              ?psm Select-design-state
                              has-design-space ?design-space)))
                  (if  (= ?state :nothing)
                    (RETURN :nothing)
                    (if (achieved (the-current-method) ?state)
                      (return ?state)
                      (do
                        (achieve-generic-subtask
                         ?psm reflect-design-state
                         has-design-state ?state)
                        (achieve-generic-subtask
                         ?psm design-from-state
                         has-design-state ?state
                         has-design-space ?design-space))))))))))
  :own-slots ((has-generic-subtasks
               '(initialise-design-space
                 design-from-state
                 reflect-design-state
                 select-design-state))))

(def-class SEARCH-CONTROL-RECORD ()
  "This structure records the control information associated
   with a state.  It is necessary to be able to support
   generic control regimes"
  ((has-design-state :type design-state :cardinality 1)
   (has-design-focus :type design-focus  :cardinality 1)
   (has-current-operator :type design-operator :max-cardinality 1)
   (has-design-operators :type list  :cardinality 1)
   (has-design-foci :type list  :cardinality 1)))

(def-function THE-STATE-SEARCH-CONTROL-RECORD (?state)
  :body (the ?record (and (search-control-record ?record)
                          (has-design-state ?record ?state))))
```

```
(def-relation STATE-FULLY-EXPANDED (?state)
  :iff-def (and (= ?record (the-state-search-control-record ?state))
                (has-design-foci ?record nil)
                (has-design-operators ?record nil)))


(def-class DESIGN-SPACE () ?x
  "A design space is a set of design states associated
   with a parametric design task"
  ((associated-with-task :type parametric-design :cardinality 1)
   (has-states :type set :cardinality 1 :default-value nil))
  :constraint (=> (member ?s (the ?set (has-states ?x ?set)))
                  (design-state ?s)))

(def-function DESIGN-SPACE-STATES (?space)
  :constraint (design-space ?space)
  :body (the ?states (has-states ?space ?states)))

(def-class INITIALISE-DESIGN-SPACE (composite-task) ?psm
  "Creates an initial design state (which is empty) and
   then returns a list containing only this state"
  ((has-input-role :value has-current-pardes-task)
   (has-output-role :value has-design-space)
   (has-control-role :value has-design-model)
   (has-current-pardes-task :type task)
   (has-design-space :type design-space)
   (has-body :value (lambda (?psm)
                      (in-environment
                       ((?name . (new-symbol 'design-space)))
                       (tell (design-space
                              ?name
                              has-states nil
                              associated-with-task
                              (role-value
                               ?psm
                               has-current-pardes-task)))
                       (achieve-generic-subtask
                        ?psm
                        new-design-state
                        has-design-model nil
                        has-design-space ?name)
                       ?name))))
  :own-slots ((has-generic-subtasks '(new-design-state))))
```

```
(def-class NEW-DESIGN-STATE (composite-task) ?psm
  "Creates a design state"
  ((has-output-role :value has-design-state)
   (has-input-role  :value has-design-model
                    :value has-design-space)
   (has-design-space :type design-space)
   (has-design-state :type design-state)
   (has-design-model :type design-model)
   (has-body
    :value (lambda (?psm)
             (in-environment
              ((?design-model . (the ?dm2
                                  (has-design-model
                                   ?psm ?dm2)))
               (?design-space . (role-value
                                 ?psm has-design-space))
               (?name . (new-symbol 'design-state)))
              (tell (design-state ?name
                                  has-design-model
                                  ?design-model))
              (append-slot-value
               ?design-space has-states  ?name)
              (achieve-generic-subtask ?psm
                                       evaluate-design-state
                                       has-design-state ?name)
              ?name))))
  :own-slots ((has-generic-subtasks  '(evaluate-design-state))))

(def-class SELECT-DESIGN-STATE (goal-specification-task) ?task
  ((has-input-role :value has-design-space)
   (has-output-role :value has-design-state)
   (has-goal-expression
    :value (kappa (?task)
                  (exists ?s
                          (and (design-state ?s)
                               (has-design-state ?task ?s)))))
   (has-design-space :type design-space)
   (has-design-state :type design-state)))

(def-function filter-feasible-consistent-states (?states)
  :body (setofall ?state
                  (and (member ?state ?states)
                       (not (deadend-state ?state))
                       (not (constraint-violations
                             ?state ?cs)))))

(def-function filter-maximal-states (?states)
  :body (setofall
         ?state
         (and (member ?state ?states)
              (has-design-model ?state ?dm)
              (= ?l (length ?dm))
              (not
               (exists
                ?state2
                (and (member ?state2 ?states)
                     (has-design-model ?state2 ?dm2)
                     (= ?l2 (length ?dm2))
                     (> ?l2 ?l)))))))
```

```
(def-function filter-cheapest-states (?states ?cost-order-rel)
  :body (setofall ?state
                  (and (member ?state ?states)
                       (state-cost ?state ?cost)
                       (not (exists
                             ?state2
                             (and (member ?state2 ?states)
                                  (state-cost ?state2 ?cost2)
                                  (holds ?cost-order-rel
                                         ?cost2 ?cost)))))))

(def-class CONSISTENT-MAX-CHEAPEST-STATE-SELECTION (primitive-method)
  ((has-body
    :value (lambda (?psm)
             (in-environment
              ((?cost-algebra . (role-value ?psm has-cost-algebra))
               (?cost-rel . (third ?cost-algebra))
               (?space . (role-value ?psm has-design-space))
               (?states . (design-space-states ?space)))
             (first
              (filter-cheapest-states
              (filter-maximal-states
               (filter-feasible-consistent-states ?states))
             ?cost-rel))))))
  :own-slots ((tackles-task-type  select-design-state)))

(def-class CONSISTENT-MAX-STATE-SELECTION (primitive-method)
  ((has-body
    :value (lambda (?psm)
             (in-environment
              ((?cost-algebra . (role-value ?psm has-cost-algebra))
               (?cost-rel . (third ?cost-algebra))
               (?space . (role-value ?psm has-design-space))
               (?states . (design-space-states ?space)))
             (first
              (filter-maximal-states
               (filter-feasible-consistent-states ?states)))))))
  :own-slots ((tackles-task-type  select-design-state)))

(def-class CONSISTENT-CHEAPEST-MAX-STATE-SELECTION (primitive-method)
  ((has-body
    :value (lambda (?psm)
             (in-environment
              ((?cost-algebra . (role-value ?psm has-cost-algebra))
               (?cost-rel . (third ?cost-algebra))
               (?space . (role-value ?psm has-design-space))
               (?states . (design-space-states ?space)))
             (first
              (filter-maximal-states
               (filter-cheapest-states
                (filter-feasible-consistent-states ?states)
                ?cost-rel)))))))
  :own-slots ((tackles-task-type  select-design-state)))
```

```
(def-class DESIGN-FROM-STATE (goal-specification-task) ?task
  "This task provides a place to define the main strategy
   to move from a state which is not a solution to one
   which is 'better'.  Of course, criteria are method-dependent.
   The :constraint option below states that the input state
   is not a solution to the current problem"
  ((has-input-role :value has-design-state
                   :value has-design-space)
   (has-output-role :value has-output-state)
   (has-output-state :type design-state)
   (has-design-state :type design-state)
   (has-design-space :type design-space)
   (has-goal-expression
    :value (kappa (?task ?s)
                  (design-state ?s))))
  :constraint (and (has-design-state ?task ?s)
                   (has-design-model ?s ?dm)
                   (= ?pd-problem (role-value
                                   ?task has-current-pardes-task))
                   (not (achieved ?pd-problem ?dm))))

(def-class EXTEND-INCOMPLETE-STATE (decomposition-method)
  ((has-input-role :value has-design-state)
   (has-output-role :value generates-design-state)
   (has-design-state :type design-state)
   (generates-design-state :type design-state)
   (has-goal-expression
    :value (kappa (?task ?s)
                  (design-model-extends
                   (the ?dm (has-design-model ?s ?dm))
                   (the ?dm (has-design-model
                             (role-value
                              ?task has-design-state)
                             ?dm)))))
   (has-body
    :value
    (lambda (?psm)
      (in-environment
       ((?state . (role-value ?psm has-design-state))
        (?design-model . (the ?dm (has-design-model
                                   ?state ?dm)))
        (?constraints . (role-value ?psm has-constraints))
        (?parameters . (role-value ?psm has-parameters)))
       (if (deadend-state ?state)
         :nothing
         (if (constraint-violations ?state ?constraints)
           (tell (deadend-state ?state))
           (if (state-complete ?state ?parameters)
             (tell (solution-state ?state))
             (achieve-generic-subtask
              ?psm
              generate-state-successor
              has-design-state ?state
              has-design-context :extend))))))))

  :own-slots ((tackles-task-type  design-from-state)
              (has-generic-subtasks generate-state-successor)))
```

```
(def-relation deadend-state (?state)
 "True if a state is a 'failure'.  Failure is a
   PSM-related concept.  For instance many psms
   work in a consistency-first style, and therefore
   regard inconsistent states as failure"
 :constraint (design-state ?state))

(def-relation state-complete (?state ?parameters)
  :iff-def (and (has-design-model ?state ?design-model)
                (design-model-complete ?design-model ?parameters)))

(def-relation constraint-violations (?state ?cs)
  "Associates a state to the constraints violated by
   the design model associated with the state"
   :constraint (and (design-state ?state)
                    (list ?cs)
                    (every ?cs constraint)))

(def-class EVALUATE-DESIGN-STATE (composite-task) ?task
  ((has-input-role :value has-design-state)
   (has-design-state :type design-state)
   (has-body
    :value (lambda (?task)
             (in-environment
              ((?state . (role-value ?task has-design-state)))
              (achieve-generic-subtask ?task reflect-design-state
                                       has-design-state ?state)
              (achieve-generic-subtask ?task evaluate-consistency
                                       has-design-state ?state)
              (achieve-generic-subtask ?task evaluate-completeness
                                       has-design-state ?state)
              (achieve-generic-subtask ?task evaluate-cost
                                       has-design-state ?state)
              (achieve-generic-subtask ?task evaluate-feasibility
                                       has-design-state ?state))))))

(def-class REFLECT-DESIGN-STATE (goal-specification-task) ?task
  ((has-input-role :value has-design-state)
   (has-design-state :type design-state)))

(def-class EVALUATE-COMPLETENESS (primitive-task) ?task
  ((has-input-role :value has-design-state)
   (has-design-state :type design-state)
   (has-body
    :value (lambda (?task)
             (in-environment
              ((?state . (role-value ?task has-design-state))
               (?design-model . (the ?dm
                                  (has-design-model ?state ?dm)))
               (?parameters . (role-value ?task has-parameters)))
              (if (design-model-complete ?design-model ?parameters)
                (tell (state-complete ?state))))))))

(def-class EVALUATE-COST (goal-specification-task) ?task
  ((has-input-role :value has-design-state)
   (has-output-role :value has-cost)
   (has-design-state :type design-state)
   (has-cost :type cost)
   (has-goal-expression
    :value (kappa (?task ?cost)
                  (and (cost ?cost)
                       (has-cost ?task ?cost))))))
```

```
(def-class DEFAULT-COST-EVALUATION (primitive-method) ?psm
  ((has-body
    :value (lambda (?psm)
             (in-environment
              ((?state . (role-value ?psm has-design-state))
               (?design-model . (the ?dm (has-design-model
                                          ?state ?dm)))
               (?cost-fun . (role-value ?psm has-cost-function))
               (?cost . (call ?cost-fun ?design-model)))
              (do
                (tell (state-cost ?state ?cost))
                ?cost)))))
  :own-slots ((tackles-task-type  evaluate-cost)))

(def-class EVALUATE-CONSISTENCY (primitive-task) ?task
  ((has-input-role :value has-design-state)
   (has-design-state :type design-state)
   (has-body
    :value (lambda (?task)
             (in-environment
              ((?state . (role-value ?task has-design-state))
               (?design-model . (the ?dm (has-design-model
                                          ?state ?dm)))
               (?constraints . (role-value ?task has-constraints))
               (?vs . (setofall ?c (and (member ?c ?constraints)
                                        (design-model-violates
                                         ?design-model ?c)))))
              (if (not (null ?vs))
                (tell (constraint-violations ?state ?vs)))
             ?vs)))))

(def-class EVALUATE-FEASIBILITY (primitive-task) ?task
  ((has-input-role :value has-design-state)
   (has-design-state :type design-state)
   (has-body :value (lambda (?task)
                      true))))
```

```
(def-class generate-state-successor (composite-task)
  ((has-input-role :value has-design-state
                   :value has-design-context)
   (has-output-role :value generates-design-state)
   (has-design-context :type design-context)
   (has-design-state :type design-state)
   (generates-design-state :type design-state)
   (has-body :value (lambda (?task)
               (in-environment
                ((?state . (role-value ?task has-design-state))
                 (?params . (role-value ?task has-parameters))
                 (?context . (role-value ?task has-design-context)))

                (if (search-control-record ?record
                                           has-design-state ?state)

                  ;;we are effectively backtracking to ?state
                  (in-environment
                   ((?result .  (achieve-generic-subtask
                                 ?task resume-state
                                 has-design-state ?state
                                 has-design-context ?context)))
                   (if (design-state ?result)
                     ?result
                     (achieve-generic-subtask
                      ?task
                      design-from-context
                      has-design-state ?state
                      has-design-context ?context)))

                  ;;?state is a newly-created state
                  (in-environment
                   ((?foci . (achieve-generic-subtask
                              ?task collect-state-foci
                              has-design-state ?state
                              has-design-context ?context)))
                   (new-search-control-record ?state ?foci)
                   (achieve-generic-subtask
                    ?task design-from-context
                     has-design-state ?state
                     has-design-context ?context))))))
   :own-slots ((has-generic-subtasks '(resume-state
                                       design-from-context
                                       collect-state-foci))))

(def-class collect-state-foci (goal-specification-task) ?task
  ((has-input-role :value  has-design-context
                   :value  has-design-state)
   (has-output-role :value  has-design-foci)
   (has-design-foci :type list)
   (has-design-state :type design-state)
   (has-design-context :type design-context)))
```

```
(def-class collect-computable-parameters  (primitive-method)
  ((has-body
    :value (lambda (?psm)
             (all-computable-parameters
              (role-value ?psm has-parameters)
              (the ?dm (has-design-model
                        (role-value ?psm has-design-state)
                        ?dm))))))
  :own-slots ((tackles-task-type collect-state-foci)
              (applicability-condition
               (kappa (?task)
                      (= (role-value ?task
                                     'has-design-context)
                         :extend)))))

(def-procedure  new-search-control-record (?state ?foci)
  :body (tell (search-control-record
               (new-symbol 'state-search-control-record)
               has-design-state ?state
               has-design-foci ?foci)))
```

```
(def-class DESIGN-FROM-CONTEXT (composite-task) ?task
  ((has-input-role :value has-design-state
                   :value has-design-context)
   (has-output-role :value generates-design-state)
   (has-control-role :value has-design-foci
                     :value has-search-control-record)
   (has-design-context :type design-context)
   (has-design-state :type design-state)
   (generates-design-state :type design-state)
   (has-body
    :value
    (lambda (?task)
      (REPEAT
       (in-environment
        ((?state . (role-value ?task has-design-state))
         (?record . (the-state-search-control-record
                     ?state))
         (?foci . (the-slot-value ?record
                                  'has-design-foci))
         (?sub . (instantiate-generic-subtask
                  ?task select-design-focus
                  has-design-foci ?foci))
         (?focus . (solve-task ?sub)))
        (if (achieved ?sub ?focus)
          (do
            (achieve-generic-subtask
             ?task
             update-search-control-record-on-focus-selection
             has-search-control-record ?record
             has-design-focus ?focus)
            (in-environment
             ((?ops . (achieve-generic-subtask
                       ?task collect-focus-operators
                       has-design-focus ?focus))
              (?sorted-ops . (achieve-generic-subtask
                              ?task sort-design-operators
                              has-design-operators ?ops)))
             (if (null ?sorted-ops)
               (achieve-generic-subtask
                ?task
                update-search-control-record-on-focus-failure
                has-search-control-record ?record
                 has-design-focus ?focus)
               (do
                 (set-slot-value ?record
                                 has-design-operators
                                 ?sorted-ops)
                 (in-environment
                  ((?result .  (achieve-generic-subtask
                                ?task design-from-focus
                                has-design-state ?state)))
                  (if (design-state ?result)
                    (return ?result)))))))
          (do
            (tell (deadend-state ?state))
            (return :nothing))))))))
  :own-slots ((has-generic-subtasks
               '(select-design-focus
                 collect-focus-operators
                 sort-design-operators
                 update-search-control-record-on-focus-failure
                 update-search-control-record-on-focus-selection
                 design-from-focus))))
```

```
(def-class UPDATE-SEARCH-CONTROL-RECORD-ON-FOCUS-FAILURE
           (goal-specification-task) ?task
  ((has-input-role :value  has-search-control-record
                   :value has-design-focus)
   (has-design-focus :type design-focus)
   (has-search-control-record :type search-control-record)))

(def-class DEFAULT-SEARCH-CONTROL-RECORD-ON-FOCUS-FAILURE-UPDATE
          (primitive-method) ?psm
  ((has-body
    :value (lambda (?psm) :nothing)))
   :own-slots ((tackles-task-type update-search-control-record-on-
focus-failure)))

(def-class UPDATE-SEARCH-CONTROL-RECORD-ON-FOCUS-SELECTION
           (goal-specification-task)
  ((has-input-role :value has-search-control-record
                   :value has-design-focus)
   (has-design-focus :type design-focus)
   (has-search-control-record :type search-control-record)))

(def-class DEFAULT-SEARCH-CONTROL-RECORD-ON-FOCUS-SELECTION-UPDATE
           (primitive-method) ?psm
  ((has-body
    :value (lambda (?psm)
             (in-environment
              ((?focus . (role-value ?psm has-design-focus))
               (?record . (role-value ?psm
                                      has-search-control-record)))
              (set-slot-value ?record has-design-foci
                              (remove ?focus
                                      (the-slot-value
                                       ?record has-design-foci)))
              (set-slot-value ?record has-design-focus ?focus)))))

     :own-slots ((tackles-task-type
                  update-search-control-record-on-focus-selection)))

(def-class RESUME-STATE (goal-specification-task)
  ((has-input-role :value has-design-state
                   :value has-design-context)
   (has-output-role :value has-output-design-state)
   (has-design-state :type design-state)
   (has-design-context :type design-context)
   (has-output-design-state :type design-state)
   (has-goal :value (kappa (?task ?s)
                    (and (design-state ?s)
                         (not (= ?s (role-value
                                     ?task
                                     has-design-state)))))))
```

```
(def-class TRY-DIFFERENT-STATE-OPERATOR (primitive-method) ?psm
  ((has-body
    :value (lambda (?psm)
             (achieve-generic-subtask
              ?psm design-from-focus
              has-design-state (role-value ?psm has-design-state)))))
   :own-slots ((tackles-task-type resume-state)
               (applicability-condition
                (kappa
                 (?task9)
                 (basic-operator
                  (the ?op
                    (has-current-operator
                     (the-state-search-control-record
                      (role-value
                       ?task9 has-design-state))
                     ?op)))))))

(def-class RETRY-STATE-OPERATOR (primitive-method) ?psm
  ((has-body
    :value (lambda (?psm)
             (in-environment
              ((?state . (role-value ?psm has-design-state))
               (?record . (the-state-search-control-record ?state))
               (?op . (the ?op2 (has-current-operator ?record ?op2))))
              (if (has-design-focus ?record ?focus)
                (in-environment
                  ((?sub . (instantiate-generic-subtask
                            ?psm try-design-operator
                            has-design-operator ?op
                            has-design-focus ?focus
                            has-design-model (the-slot-value
                                              ?state
                                              'has-design-model)))
                   (?result . (solve-task ?sub)))
                  (if (achieved ?sub2 ?result)
                    ?result
                    (achieve-generic-subtask
                     ?psm design-from-focus
                     has-design-state ?state)))))))
   :own-slots ((tackles-task-type resume-state)
               (applicability-condition
                (kappa
                 (?task9)
                 (multiple-operator
                  (the ?op
                    (has-current-operator
                     (the-state-search-control-record
                      (role-value
                       ?task9 has-design-state))
                     ?op)))))))
```

```
(def-class DESIGN-FROM-FOCUS (composite-task)
  ((has-input-role :value has-design-state)
   (has-output-role :value has-output-design-state)
   (has-control-role :value has-design-model
                     :value has-design-operator)
   (has-design-state :type design-state)
   (has-output-design-state :type design-state)
   (has-body
    :value
    (lambda (?task)
      (REPEAT
       (in-environment
        ((?state . (role-value ?task has-design-state))
         (?record . (the-state-search-control-record
                     ?state))
         (?focus . (the-slot-value
                    ?record 'has-design-focus))
         (?ops .  (the-slot-value
                   ?record 'has-design-operators))
         (?sub . (instantiate-generic-subtask
                  ?task select-design-operator
                  has-design-focus ?focus
                  has-design-operators ?ops))
         (?op . (solve-task ?sub)))
        (set-slot-value ?record has-current-operator ?op)
        (if (achieved ?sub ?op)
          (DO
            (set-slot-value
             ?record
             has-design-operators
             (remove ?op ?ops))
            (in-environment
             ((?sub2 . (instantiate-generic-subtask
                        ?task try-design-operator
                        has-design-operator ?op
                        has-design-focus ?focus
                        has-design-model (the-slot-value
                                          ?state
                                          'has-design-model)))
              (?result . (solve-task ?sub2)))
             (if (achieved ?sub2 ?result)
               (RETURN ?result))))
          (RETURN :nothing)))))))
  :own-slots ((has-generic-subtasks '(select-design-operator
                                      try-design-operator))))

(def-class TRY-DESIGN-OPERATOR (goal-specification-task) ?task
  ((has-input-role :value has-design-operator
                   :value has-design-focus
                   :value has-design-model)
   (has-output-role :value generates-design-state)
   (has-design-focus :type design-focus)
   (has-design-operator :type design-operator)
   (has-design-model :type design-model)
   (generates-design-state :type design-state)
   (has-goal-expression
    :value (kappa (?task8 ?s)
                  (and (design-state ?s)
                       (generates-design-state ?task8 ?s))))))
```

```
(def-class TRY-DESIGN-EXTENSION-OPERATOR (primitive-method)
  ((has-body
    :value
    (lambda (?psm)
      (in-environment
       ((?dm . (role-value ?psm 'has-design-model))
        (?focus . (role-value ?psm 'has-design-focus))
        (?value . (apply-design-extension-operator
                   ?focus ?dm
                   (role-value ?psm 'has-design-operator))))
       (if (not (= ?value :nothing))
         (achieve-generic-subtask
          ?psm new-design-state
          has-design-model (cons
                            (cons ?focus ?value)
                            ?dm)))))))
  :own-slots ((tackles-task-type try-design-operator)
              (applicability-condition
               (kappa
                (?task5)
                (design-extension-operator
                 (role-value
                  ?task5 has-design-operator))))))

(def-function apply-design-extension-operator (?param ?dm ?op)
  :constraint (and (parameter ?param)
                   (design-model ?dm)
                   (design-extension-operator ?op))
  :body (call (the ?body
                (has-body ?op ?body))
              ?param
              ?dm))

(def-class SELECT-DESIGN-OPERATOR (goal-specification-task) ?task
  ((has-input-role :value has-design-operators
                   :value has-design-focus)
   (has-output-role :value has-selected-operator)
   (has-design-operators :type list)
   (has-selected-operator :type design-operator)
   (has-design-focus :type design-focus)
   (has-goal-expression
    :value (kappa (?task ?op)
                  (and (design-operator ?op)
                       (has-selected-operator ?task ?op))))))

(def-class DEFAULT-OPERATOR-SELECTION (primitive-method) ?psm
  ((has-body
    :value (lambda (?psm)
             (first (role-value ?psm
                                'has-design-operators)))))
   :own-slots ((tackles-task-type  select-design-operator)))

(def-class COLLECT-FOCUS-OPERATORS (goal-specification-task) ?task
  ((has-input-role :value has-design-focus)
   (has-design-focus :type design-focus)))
```

```
(def-class DEFAULT-OPERATOR-COLLECTION (primitive-method) ?psm
  ((has-body
    :value (lambda (?psm)
             (setofall ?op
                       (and (design-operator
                             ?op
                             applicable-to-parameters ?l)
                            (member (role-value
                                     ?psm 'has-design-focus)
                                    (eval ?l)))))))
  :own-slots ((tackles-task-type collect-focus-operators)
              (applicability-condition
               (kappa
                (?task)
                (and (= :extend
                        (role-value
                         ?task 'has-design-context))
                     (parameter
                      (role-value
                       ?task 'has-design-focus)))))))

(def-class SORT-DESIGN-OPERATORS (primitive-task) ?task
  ((has-input-role :value has-design-operators
                   :value has-operator-order-relation)
   (has-design-operators :type list)
   (has-operator-order-relation :default-value design-operator-order)
   (has-body
    :value (lambda (?task)
             (sort (role-value
                    ?task has-design-operators)
                   (role-value ?task has-operator-order-relation))))))

(def-class SELECT-DESIGN-FOCUS (goal-specification-task) ?task
  ((has-input-role :value has-design-foci)
   (has-output-role :value has-design-focus)
   (has-design-foci :type list)
   (has-design-focus :type design-focus)
   (has-goal-expression
    :value (kappa (?task ?focus)
             (has-design-focus ?task ?focus))))
```

```
(def-class DEFAULT-PARAMETER-SELECTION (primitive-method)
  ((has-input-role
    :value has-design-focus-order-relation
    :value has-possible-values-relation)
   (has-design-focus-order-relation
    :default-value design-focus-order)
   (has-possible-values-relation
    :default-value possible-value)
   (has-body
    :value (lambda (?psm)
             (if  (= ?foci (role-value ?psm has-design-foci))
               (select-most-preferred-focus
                (collect-most-restricted-parameters
                 ?foci
                 (role-value ?psm
                             has-possible-values-relation))
                (role-value ?psm
                            has-design-focus-order-relation))))))
  :own-slots ((tackles-task-type select-design-focus)
              (applicability-condition
               (kappa (?task)
                      (every (the ?foci
                                  (has-design-foci
                                   ?task ?foci))
                             parameter)))))

;;;USE-METHOD STATEMENTS
(tell (use-method consistent-max-state-selection
                  select-design-state
                  gen-design-psm))
```