Enhancing Knowledge Acquisition

with

Constraint Technology

A Thesis Presented for the Degree of

Doctor of Philosophy

at the University of Aberdeen

Simon White

BSc Computer Science and Mathematics (York)

MSc Applied Artificial Intelligence (Aberdeen)

2000

Declaration

I declare that this thesis has been composed by myself and describes my own work. It has not been accepted in any previous application for a degree. All verbatim extracts have been distinguished by quotation marks, and all sources of knowledge have been specifically acknowledged.

Simon White

March 30, 2000

Department of Computing Science,

University of Aberdeen,

King's College,

Aberdeen

To Timi

Acknowledgements

This dissertation would not have been possible without the support of many colleagues, friends, and family. My sincerest thanks go to them all. In particular, I should like to thank my supervisor, Derek Sleeman, for countless valuable discussions, and his continued enthusiasm for my work. I also gratefully acknowledge the contributions of Nicolas Graner and the MLT project, who laid the early foundations for my work. I thank Ken Brown for introducing me to the technology of constraints, and some useful comments on my progress with SCREAMER and SCREAMER+.

I am very grateful to the EPSRC for funding my studies, and to the computing officers and secretarial staff of the University of Aberdeen Computing Science Department for the invaluable services they provide. Although not directly related to my PhD work, I am also grateful to those former colleagues with whom I gained the experience that made me a more ‘plausible’ PhD candidate. Among those, I should like to mention Richard Göbel, Dietmar Pfahl, Kim Jungfer, Robert Rodosek, and Hans Weber.

Finally, I should like to express my gratitude for the unerring emotional support of my family. I regard myself as very lucky that my wife was willing to follow me to a new and very foreign country, share the frustrations of my research, and tolerate the hardships associated with being a student family. Szeretlek, Timi.

Simon White
March 30, 2000

Abstract

This dissertation introduces the notion of *fitness-for-purpose* applied to a set of knowledge bases, a problem solver, and a problem-solving goal. Fitness-for-purpose addresses the issue of goal achievability when the knowledge bases are allocated roles and used as the problem solver's inputs. In particular, we seek to determine whether the goal is achievable without running the problem solver, and acquire knowledge bases for particular input roles such that the goal is achievable. The dissertation argues that constraint technology is well-suited for specifying and examining these fitness-for-purpose requirements.

This thesis is directly supported by two pieces of work. Firstly, I demonstrate the use of constraint technology to *specify* necessary conditions for achieving a problem-solving goal; and *detect* configurations of a problem solver and knowledge bases which do not meet those criteria. Elimination of the configurations which *will not* solve the goal is significant progress towards the discovery of configurations which *will* solve the goal. Secondly, I demonstrate that constraint-based fitness-for-purpose specifications can be interpreted by a computer and used as the basis for a domain-independent knowledge acquisition tool. The implementation described uses a constraint-augmented grammar as the notation for specifying required knowledge. The tool examines knowledge as it is acquired and can react to acquired knowledge in a way that attempts to meet fitness-for-purpose requirements.

These ideas were implemented using a LISP-based constraints package, called SCREAMER. Shortcomings in the package's expressiveness led to the development of SCREAMER+: an improved version which extends the original in three major directions. Firstly, it improves the capability to assert constraints on lists and their subparts; secondly, it provides constraint-based versions of higher-order functions[1] such as **`some`**, **`every`** and **`mapcar`**; and, thirdly, it integrates SCREAMER constraints with CLOS, the object-oriented part of Common LISP.

1. A higher-order function is a function which takes another function as an argument.

Contents

Title i

Declaration ii

Acknowledgements iv

Abstract v

Contents vi

1 Introduction 1

1.1 Background and Motivation 1

1.2 Problem Statement and Thesis 3

1.3 Overview of Accomplishments 6

1.4 Dissertation Layout 7

2 A Brief Review of Knowledge Acquisition 10

2.1 What is Knowledge Acquisition? 10

2.2 Knowledge Elicitation 15

2.2.1 Interviewing 18

2.2.2 Sorting 18

2.2.3 Laddering 19

2.2.4 Repertory Grid 19

2.2.5 Automated Knowledge Elicitation 21

2.3 Machine Learning 22

2.3.1 Inductive Learning 23

2.3.2 Analytical Learning 25

2.3.3 Genetic Algorithms 27

2.3.4 Connectionist Learning 29

2.4 Knowledge Base Refinement 31
2.4.1 TEIRESIAS 33
2.4.2 KRUST / STALKER 33
2.5 Trends and Current Research Issues 34
2.5.1 The Rise of Knowledge Level Modelling 35
2.5.2 The Promise of Knowledge Sharing and Reuse 40
2.5.3 The Vision of Knowledge Management 41
2.5.4 Other Research Issues 42

3 A Brief Review of Constraint Technology 44

3.1 What is Constraint Technology? 44
3.2 Methods for Solving Constraint Satisfaction Problems 47
3.2.1 Problem Reduction Methods 49
3.2.1.1 Achieving Node Consistency 49
3.2.1.2 Achieving Arc Consistency 50
3.2.1.3 Achieving Path Consistency 52
3.2.2 Constructive Search Methods 52
3.2.2.1 Variable Ordering Strategies 53
3.2.2.2 Consistency Maintenance 55
3.2.2.3 Backjumping 57
3.2.2.4 Other Constructive Search Methods 58
3.2.3 Repair-Driven Search Methods 59
3.2.3.1 Min-Conflicts 59
3.2.3.2 Propose-and-Revise 59
3.3 Constraint Programming Systems 60
3.3.1 Formative Systems 61
3.3.1.1 Steele’s Constraint System 61
3.3.1.2 CLP(R) 62
3.3.2 Consolidated Systems 62
3.3.2.1 CHIP 62
3.3.2.2 ILOG Solver 65

3.3.2.3 IF/PROLOG 66
3.3.2.4 ECLIPSE 67
3.3.3 Progressive Systems 69
3.3.3.1 SCREAMER 70
3.3.3.2 CLAIRE / ECLAIR 70
3.3.3.3 Oz 72
3.3.4 Other Systems 74

4 MUSKRAT and the Problem of Fitness for Purpose 76

4.1 Motivation and Context 76
4.2 The Past: The Machine Learning Toolbox 78
4.2.1 The Learning Tools of MLT 78
4.2.2 The Common Knowledge Representation Language 79
4.2.2.1 Concepts, Properties and Instances 81
4.2.2.2 Sorts 82
4.2.2.3 Relations and Facts 83
4.2.2.4 Rules 83
4.2.3 The MLT Consultant 84
4.3 The Future: The MUSKRAT Vision 86
4.3.1 Critiquing the MLT Consultant 87
4.3.2 The Scope of MUSKRAT 87
4.3.3 The MUSKRAT Advisor 89
4.4 The Present: Addressing Fitness for Purpose 91
4.4.1 Three Actions Concerning Fitness for Purpose 93
4.4.2 Determining Fitness for Purpose 93
4.4.3 Acquiring Knowledge which is Fit for a Purpose 98

5 Constraint Handling in Common LISP 100

5.1 Why LISP is Almost Wonderful 100
5.2 The SCREAMER Constraint Solving Package 104

5.2.1 Nondeterminism using SCREAMER 104
5.2.2 Constraint Handling using SCREAMER 106
5.2.2.1 Creating Constraint Variables 107
5.2.2.2 Asserting Constraints 108
5.2.2.3 Searching for Solutions 109
5.2.3 How the Constraints Package of SCREAMER Works 110
5.2.3.1 Problem Set-up 110
5.2.3.2 Searching 112
5.3 The Extension to SCREAMER 115
5.3.1 Constraints on Lists 120
5.3.2 Constraints on Higher-Order Functions 121
5.3.3 Constraints on Objects 123
5.3.4 Other SCREAMER+ Constraint Functions 125
5.3.5 How SCREAMER+ Works 127
5.4 Evaluation 136
5.4.1 A Logic Problem 136
5.4.2 The Mastermind Game 140
5.4.3 Crossnumber Puzzle 142
5.4.4 Self Referential Quiz 145
5.4.5 The Car Sequencing Problem 147
5.5 Summary 154

6 Determining Fitness for Purpose 156

6.1 Introduction 156
6.2 The Plausibility Approximation 160
6.2.1 Approximating Fitness for Purpose 160
6.2.2 Approximation in the Context of Problem Solving 162
6.2.3 Verifying Fitness for Purpose 166
6.2.3.1 Building a Meta–Problem-Solver 169
6.2.3.2 Testing a Goal 174
6.3 Examples 176

6.3.1 Set Pruning 176
6.3.1.1 Problem Solver 176
6.3.1.2 Meta–problem-solver 176
6.3.1.3 Goal 180
6.3.2 Meal Design and Scheduling 183
6.3.2.1 Problem Solvers 184
6.3.2.2 Meta–problem-solvers 187
6.3.2.3 Goal 193
6.4 Discussion 194
6.4.1 Desirable Approximations 194
6.4.2 Plausibility as a Binary Decision Process 197
6.4.3 Soundness and Completeness 198
6.4.4 Functional and Dynamic Properties 199

7 Acquiring Knowledge which is Fit for a Purpose 201

7.1 Introduction and Motivation 201
7.2 Formal Grammars 202
7.2.1 Backus-Nauer Form (BNF) 205
7.2.2 Extended Backus-Nauer Form (EBNF) 206
7.3 Grammar-Driven Knowledge Elicitation 206
7.3.1 Representing a BNF Grammar 207
7.3.2 Representing an EBNF Grammar 208
7.3.3 Adding Questions and Comments 209
7.3.4 Postprocessing of Acquired Knowledge 210
7.4 Constraint-Augmented Grammars 211
7.4.1 Concise Specifications 212
7.4.2 Single-Input Property Checking 213
7.4.3 Multiple-Input Property Checking 215
7.4.4 Reactive User-Interfaces 216
7.5 Evaluation 217
7.5.1 Acquiring a Breakfast Order 217

7.5.2 Acquiring a Simple Fact Base 221
7.5.3 Acquiring a Case Base of Oil Well Drilling 223
7.5.4 Acquiring a Set of Simple Rules 225
7.6 Discussion 227
7.6.1 Related Work 230
7.6.2 Limitations 232

8 Conclusions and Further Work 235

8.1 Summary 235
8.2 Contributions 237
8.3 Further Work 241
8.3.1 Constraint Handling in Common LISP 242
8.3.2 Determining Fitness for Purpose 245
8.3.3 Acquiring Knowledge which is Fit for a Purpose 252
8.3.4 Extending the MUSKRAT Vision 253
8.4 Conclusions 255

References 256

Appendix A: The Extension to SCREAMER 268

A.1 Type Restrictions 268
A.2 Boolean Values 270
A.3 Expressions 271
A.4 Lists and Sequences 273
A.5 Sets and Bags 282
A.6 Arrays 286
A.7 Objects 287
A.8 Higher Order Functions 294
A.9 Miscellaneous 300

Appendix B: Details of Meal Design & Scheduling 303

B.1 The CKRL Knowledge Bases 303

B.1.1 Background.ckrl 303

B.1.2 Derivation.ckrl 304

B.1.3 Dish-atts.ckrl 304

B.1.4 Dish-Components.ckrl 305

B.1.5 Dishes.ckrl 306

B.1.6 Preferences.ckrl 308

B.1.7 Recipes.ckrl 308

B.1.8 Resources.ckrl 311

B.1.9 Task.ckrl 311

B.2 Translating CKRL to LISP 311

B.2.1 Concepts and Properties 312

B.2.2 Instances 314

B.2.3 Sorts 315

B.2.4 Relations and Facts 316

B.2.5 Rules 317

B.3 The Meta–Problem-Solver 319

Appendix C: Monitoring Expensive Computations 327

Appendix D: The COCKATOO Application Programmer's Interface 330

D.1 Grammars 330

D.2 Clauses 333

D.3 Constraining Clauses 337

D.4 File Naming Convention 340

Subject Index 341

Chapter 1

Introduction

'To forget one's purpose is the commonest form of stupidity.'

Friedrich Nietzsche

Chapter Summary

This chapter introduces the main topic of the dissertation, that of *fitness for purpose*. The motivation for the study of fitness for purpose originated from lessons learned during the MLT project, and the desire to build MUSKRAT, a goal-driven problem solving and knowledge acquisition toolbox. To represent fitness for purpose, the adopted approach applied the technology of constraints to the specification of knowledge. The technology provided mechanisms for propagating knowledge from one part of a knowledge base to another, and testing the consistency of available knowledge against a problem-solving goal. This chapter outlines this constraint-based approach, lists the accomplishments of the work, and explains the dissertation layout.

1.1 Background and Motivation

Knowledge Acquisition is a field that was borne out of the difficulties encountered when trying to capture the knowledge of human experts for entering into a knowledge-based system. Although the aims of the field have remained essentially the same, its interests have diversified considerably, such that it now seeks to *classify* knowledge in terms of domain, task and method; *explore* ways in which different types of knowledge can be acquired and combined (including the techniques of knowledge elicitation, ma-

chine learning and knowledge base refinement); and *investigate* profitable contexts for gathering knowledge, such as through observation of existing business processes, or collating knowledge which is distributed over the Internet.

This dissertation concerns some of the benefits that can be gained by knowledge acquisition with **Constraint Technology**. *Constraint Technology* is the set of notations, methods and software packages developed for solving problems which are naturally expressed using constraints. A classic problem of this type is the eight queens problem, in which eight queens must be placed on an 8×8 chessboard such that no two queens are placed on the same row, column, or diagonal. The restriction on the positioning of the queens is the main feature of the problem. This dissertation aims to demonstrate that the same technology can be applied to a problem whose main feature is the set of constraints on existing, or required, knowledge bases.

The origins of this work can be traced back to the Machine Learning Toolbox (MLT) project (Kodratoff et al., 1992). The aim of that project was to facilitate the application of machine learning to industrial problems by building a collection of machine learning tools, together with an advisory system, the *Consultant*. An important observation was that the Consultant made recommendations on the success of learning algorithms *without reference to any problem solver which might use the learned knowledge*. Since the recommendations were made outwith the problem-solving context, I believe their usefulness for solving problems is questionable.

The experiences of the Machine Learning Toolbox suggested that knowledge acquisition and problem solving should not be performed in mutual isolation. Furthermore, knowledge should not be acquired without forethought as to how it will be applied – knowledge acquisition should be **goal-driven**. This apparently facile statement is not as obvious as it might appear. In the past, knowledge engineers believed that knowledge could be first acquired in a relatively ad hoc manner, then later adapted to suit a specific purpose. In practice, they often found significant mismatches between the acquired knowledge and the knowledge needed for the task. This can happen when essential parts of the knowledge are missing, or the available knowledge has a significantly different representation to that required. Both cases imply rework. A *goal-driven* approach to knowledge acquisition aims to ameliorate this aspect by hav-

ing a clear picture of the task to be achieved right from the outset. At the same time, I believe it is prudent to be watchful for situations in which knowledge can be fruitfully *reused*, so that frequently used knowledge is not repeatedly acquired.

The vehicle for my investigations into goal-driven knowledge acquisition and problem solving has been MUSKRAT, a toolbox which was specified to include both problem solvers and knowledge acquisition tools (Graner & Sleeman, 1992). In this toolbox, the problem solvers can act as the "targets" of knowledge acquisition. Additionally, knowledge which has already been acquired using one of the knowledge acquisition tools could be tested for its "compatibility" with one of the problem solvers. If the knowledge passes such a test, it is a candidate for reuse. Note that although the MUSKRAT system was specified in 1992, it was never built. Moreover, the specification left the most difficult question unanswered: *how can knowledge be tested computationally for its suitability to solve a given problem?*

1.2 Problem Statement and Thesis

To make the notion of "compatibility" more concrete, I have defined the **fitness for purpose** property of a knowledge base. A knowledge base is said to be *fit for the purpose* of satisfying a problem-solving goal if it can contribute to the solution when used *without modification* with a particular problem solver. (Later, I give a more accurate definition which also takes the **role** of the knowledge base into account.) Conversely, if a modification is necessary before the knowledge base can be used, then it is *not* fit for that purpose.

The term "fitness for purpose" has been borrowed from the legal profession:

> 'Where the buyer expressly or by implication makes known to the seller the purpose for which the goods are required, so as to show that the buyer is relying on the seller's skill or judgment, and the goods are of the sort which it is the seller's business to supply, there will be an implied condition that they are *fit for the purpose*'. (Australian Contract Law[1])

1. See http://www.anu.edu.au/law/pub/edinst/anu/contract/MIMPLIED-TERMS.html#M

This definition speaks of “buyers” and “sellers”, but the exchange of money is of little concern to the context of MUSKRAT. My interest lies in the expression of a *purpose* or *goal* for which goods are acquired, and the conditions which fulfil that purpose (see Figure 1).

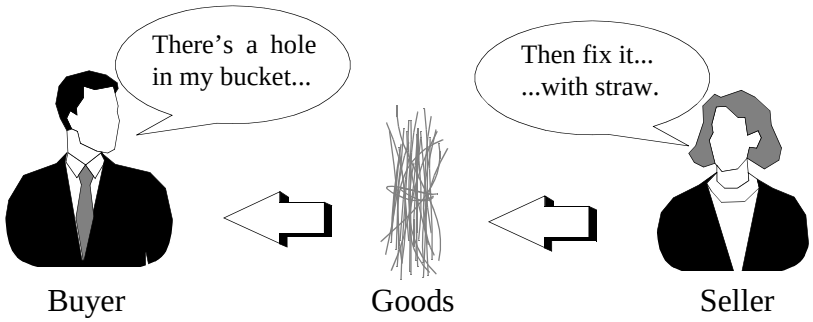


Figure 1: Fitness for Purpose in Contract Law

I believe there is a close enough parallel between the scenario of contract law and the context of MUSKRAT to justify the use of the term (see Figure 2). Perhaps the greatest difference is that, in Figure 1, the buyer and seller are both *active* agents in a fitness-for-purpose transaction, whereas in MUSKRAT, a problem solver is usually *passively* involved. That is, a problem solver would not actively declare its knowledge requirements because it would not be able to identify gaps in its own knowledge. This task is performed instead by the MUSKRAT infrastructure. In the future, problem solving agents may engage actively in fitness-for-purpose exchanges with advisory agents.

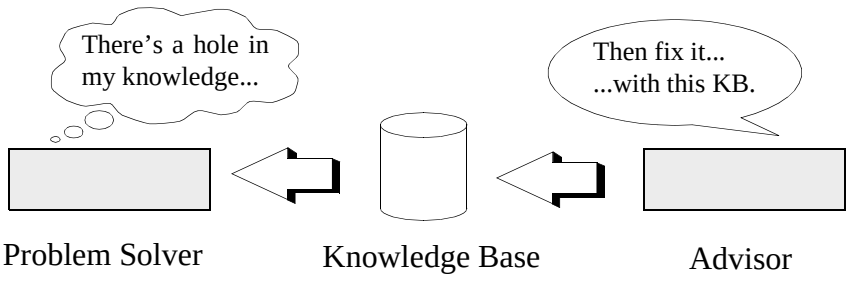


Figure 2: Fitness for Purpose in MUSKRAT

As an aside, *fitness for purpose* is also frequently cited as a practical definition of 'quality'. This applies particularly to engineering disciplines, where the alternative definition of quality as a 'degree of excellence' is too subjective to be very useful.

Now that I have introduced the notion of fitness for purpose, I can state the main thesis of this dissertation as follows.

> *The fitness-for-purpose requirements of a problem solver can be both specified and computationally examined by employing constraint technology.*

This is an interesting statement for the following reasons.

- *Fitness for purpose* is a new notion in knowledge acquisition. It is different from the notion of *competence*, a term already used in the field (e.g., Wielinga, Akkermans & Schreiber, 1998), because competence is a property that can be evaluated without knowing the problem solver's inputs. In contrast, fitness-for-purpose is a kind of *conditional competence*. It addresses the question of whether a particular problem can be solved by a given problem solver when its inputs (and their roles) are known.
- There is an important issue of tractability to be resolved, since a thorough computational examination of fitness-for-purpose requirements would encounter the Halting Problem. I approach this issue by introducing a tractable *approximation* to fitness-for-purpose.
- The thesis proposes a technology for dealing with fitness-for-purpose.
- Fitness-for-purpose represents a new application for constraint technology.

In the dissertation, the thesis is directly supported by two pieces of work. Firstly, I demonstrate the use of *constraint technology* to specify necessary conditions for achieving a problem-solving goal, and detect configurations of a problem solver and knowledge bases which do not meet those criteria. Elimination of the configurations which *will not* solve the goal is significant progress towards the discovery of configurations which *will* solve the goal. Secondly, I demonstrate that constraint-based fitness-for-purpose specifications can be interpreted by a computer and used as the basis for a domain-independent knowledge acquisition tool.

1.3 Overview of Accomplishments

The research described by this dissertation accomplished the following tasks:

- Identified the requirement that the MUSKRAT advisor (Chapter 4) be able to recognise *fitness for purpose*.
- Introduced an approximation (plausibility) to overcome the tractability problems associated with a thorough examination of *fitness for purpose* (Chapters 4 & 6).
- Investigated and chose constraint technology as the implementation approach to recognising *fitness for purpose* (Chapters 5 & 6).
- Adopted SCREAMER as the constraints package because it is LISP-based. Lack of expressiveness in SCREAMER naturally led to the development of an extended version, SCREAMER+ (Chapter 5).
- The suitability of SCREAMER+ for determining the fitness for purpose of knowledge bases was demonstrated for a simple problem solver, and in the context of a more complex problem-solving scenario (Chapter 6).
- The suitability of SCREAMER+ for specifying knowledge was exploited with the development of a knowledge acquisition tool that interprets a constraint-augmented grammar (Chapter 7).

With regard to the research methodology used, it is worth noting that some research is strongly motivated by a particular concrete application, whilst other research is more theoretical and inquisitive in nature. Most computing science research, this dissertation included, combines both theoretical and application-driven elements. The research is *application-driven* because the issue of fitness for purpose arose from a desire to build the Multistrategy Knowledge Refinement and Acquisition Toolbox, MUSKRAT. The research is also theoretical-inquisitive ("blue skies research") because it identifies and tackles a fundamentally difficult problem. I would therefore place this thesis roughly halfway between being application-driven and theoretical-inquisitive on the "motivation dimension" of AI research.

1.4 Dissertation Layout

Dissertations usually adopt a structure in which they first provide background material to the field(s) of research, then explain the main problem(s) or issue(s) tackled, before presenting and discussing the proposed solution or new approach. This dissertation is no exception. The logical structure and flow of the dissertation is summarised by Figure 3.

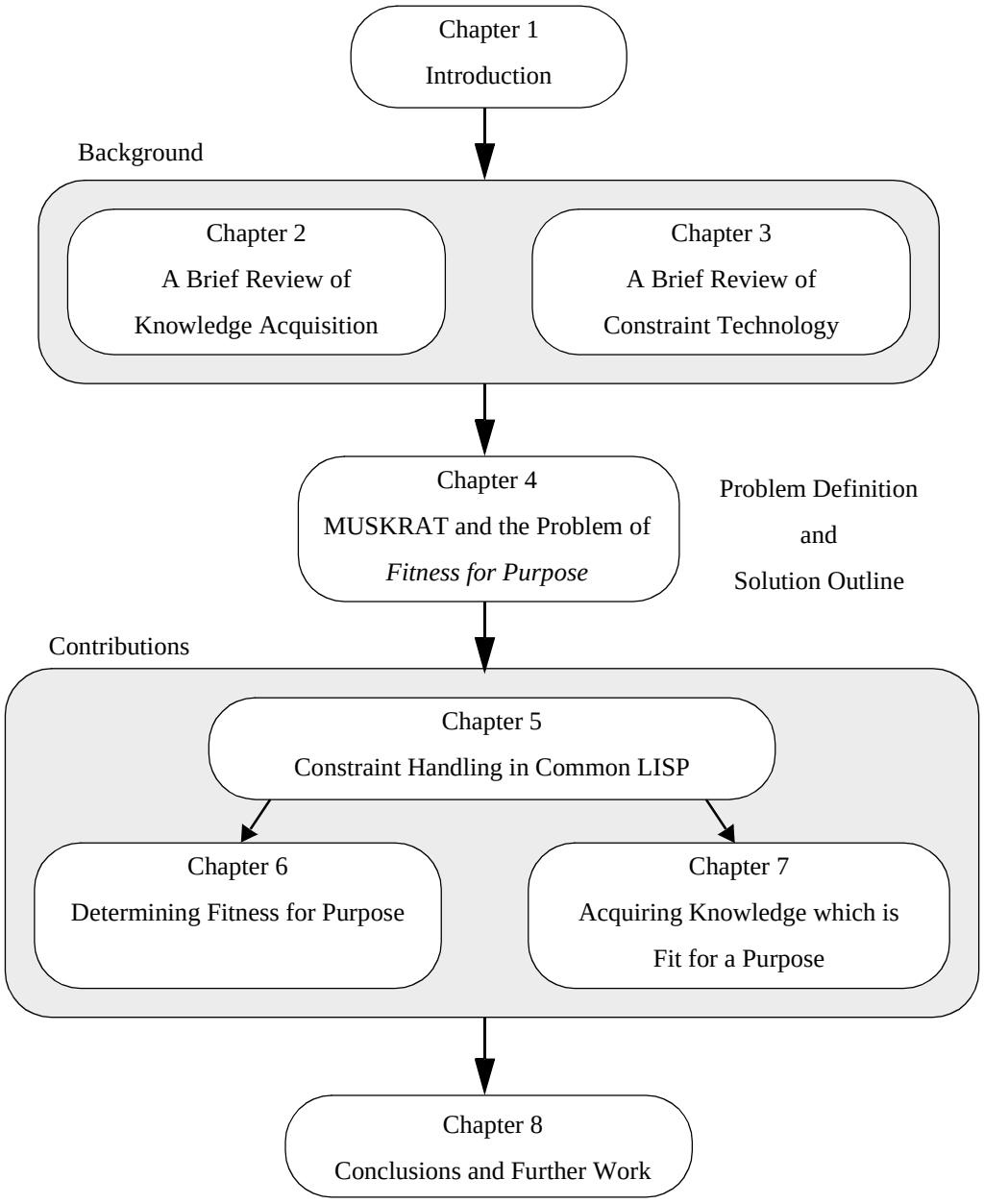


Figure 3: Flow of the Dissertation

In Chapter 1, I have sketched the nature of the problems tackled, and set the scene for the rest of the dissertation. The following two chapters provide necessary background material on the principal fields of interest, namely *knowledge acquisition* and *constraint technology*. Those readers familiar with one, or both, of these fields may choose to skip the appropriate material. Chapter 4 first provides further background which is specific to the central problem of the dissertation, that of fitness for purpose, then observes some sub-problems related to that topic, before outlining the approach taken to these problems in the later chapters. Central to this approach is the adoption of *constraint technology*, and, in particular, the LISP-based package SCREAMER. Chapter 5 (together with the more detailed descriptions in Appendix A) reports on a new extension to SCREAMER which greatly enhances its expressiveness. The enhanced constraint library, called SCREAMER+, is the technology which supports solutions to the issues discussed in chapters 6 and 7. Chapter 6 concerns an operational procedure for the automatic determination of fitness for purpose. Chapter 7 presents a new, configurable knowledge acquisition tool (COCKATOO) which is able to acquire knowledge such that it is already fit for a given purpose. Finally, in chapter 8, I discuss the contributions which this work has made, and several possible future directions.

Four appendices are also provided. Appendix A supplements the material of chapter 5 by documenting all the functions of the SCREAMER extension. Appendix B provides details of the meal design and scheduling example given in chapter 6, including the mechanism employed to translate MUSKRAT knowledge bases into LISP data structures. Appendix C provides supplementary material to support the idea of monitoring the progress of expensive computations, discussed in chapter 6. Finally, Appendix D documents the functions available to the programmer when building an application with the knowledge acquisition tool COCKATOO.

In writing the dissertation, one of my aims was to make it readable by those with a good general grounding in computing (e.g., a postgraduate computer science student specialising in another field), but little experience of knowledge acquisition and constraint technology. Even in a work of this size, however, it is not possible to elaborate on every detail for such an audience. It would be impractical, for example, to provide an exten-

sive tutorial on Common LISP to support chapter 5. Nevertheless, I hope that the dissertation is found to be “fit for the purpose” of its intended readership.

Note that some of the work presented in this dissertation has already been published. An extended abstract sketching some of the motivational aspects of the MUSKRAT project is in the proceedings of the Nineteenth Annual Conference of the Cognitive Science Society, a meeting held at Stanford University (Sleeman & White, 1997). A paper outlining the intentions of the MUSKRAT project, with similar material to Chapter 4 of this dissertation, was first published as a University of Aberdeen technical report (Sleeman & White, 1996), then later accepted in an updated form for the proceedings of the Eleventh Knowledge Acquisition Workshop in Banff, Canada (White & Sleeman, 1998a). The material on constraint handling in Common LISP, covered by Chapter 5 and Appendix A of this dissertation, was published in a more concise form as a technical report at the University of Aberdeen (White & Sleeman, 1998b). More recently, a paper introducing the abstraction/plausibility approach to the description of competence, covered in more detail by Chapter 6, was published in the proceedings of the Eleventh European Workshop on Knowledge Acquisition, Modelling, and Management (White & Sleeman, 1999).

Chapter 2

A Brief Review of Knowledge Acquisition

'Solving problems is our business.'
Bob Wielinga

Chapter Summary

In the 1970s and 1980s, knowledge acquisition was seen as the transfer of problem solving expertise from a human expert to a knowledge-based system. Since manual methods for this process were found to be very difficult and expensive, knowledge acquisition was famously dubbed the 'bottleneck' to building expert systems. To break the bottleneck, several different approaches have been tried, including automated knowledge elicitation, machine learning, and knowledge base refinement. This chapter presents illustrative examples from each of these approaches to knowledge acquisition, and explains the rising profile of knowledge level modelling in recent years.

2.1 What is Knowledge Acquisition?

In the early 1970s, a collaboration between medical and artificial intelligence researchers at Stanford University resulted in the development of one of the first **Knowledge-Based Systems**, called **MYCIN**. Knowledge-based systems (also called **Expert Systems**) are designed to solve problems in specific areas of expertise (their **domain**) by representing and applying the relevant problem-solving knowledge. MYCIN's area of expertise was in the diagnosis and treatment of blood infections; knowledge-based systems (KBSs) have also been applied to many other domains. For example, **DENDRAL**

was an early knowledge-based system devised to determine the molecular structure of unknown organic compounds by interpreting their mass spectrograms, a task normally accomplished by specialised chemists. **R1** (McDermott, 1982), also called **XCON**, was a knowledge-based system developed jointly by Carnegie-Mellon University and the Digital Equipment Corporation to configure the installation of mainframe computers, a time-consuming task normally performed by experienced engineers.

The domain knowledge of MYCIN was represented as a set of heuristic rules of the form:

IF (condition_1 **AND** condition_2 **AND** ... **AND** condition_m)
THEN conclude_1; conclude_2; ... ; conclude_n;

The truth value of each condition in a rule was found by a combination of testing against known data and requesting additional information from a physician. When interpreting the rule set, the program 'knew' what overall conclusion it would like to draw (its top-level *goal*), and sought out a combination of rules which might lead to that conclusion. This is called a **backward-chaining** control structure, because the program assumes the desirable goal state is achievable and looks backwards from the goal for a sequence of inferences that would link it to the current knowledge state. In contrast, a **forward-chaining** control structure searches forward from the current knowledge state to the goal (See Figure 4).

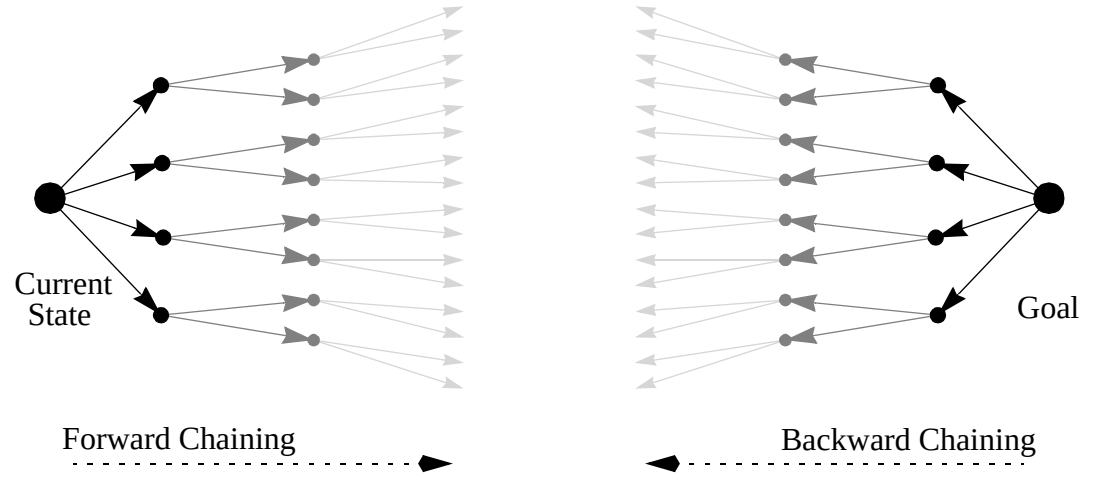


Figure 4: Forward and Backward Chaining in Knowledge-Based Systems

Later, a domain independent version of MYCIN was developed, called **EMYCIN**. This contained the same control mechanisms as MYCIN, but knew nothing of blood infections. The idea was that the same system could be applied to a *different* domain by supplying a new set of (domain-dependent) rules. This was one of the first attempts to separate the domain knowledge from the control structure. Knowledge Acquisition was borne out of this separation, since the question was then *how* to acquire the domain knowledge. Later, attention was also brought to the importance of the control structure: it became clear that EMYCIN was *not* a general-purpose problem solver, but *was* well-suited to diagnostic tasks for which large amounts of data are available.

Having explained the background and purpose of the knowledge acquisition task, I can now provide a definition.

Definition 1: Knowledge Acquisition

> 'The transfer and transformation of potential problem solving expertise from some knowledge source to a program.' (Buchanan et al., 1983)

During the 1970s and 1980s, most researchers and practitioners believed that knowledge should be acquired in the following way. A person, called a **knowledge engineer**, first interrogates a **domain expert**, then organises and ultimately formalises the extracted knowledge such that it is suitable for processing by a knowledge-based system. Afterwards, the performance of the extracted knowledge can be tested by presenting example problems to the knowledge-based system. If the system does not perform as well as expected, a refinement process attempts to improve performance. There were thus seen to be three basic stages to knowledge acquisition, namely **elicitation** (the interaction between the knowledge engineer and the domain expert), **formalisation** (also called **encoding**), and **refinement**. Early attempts to acquire knowledge in this way proved to be so unfruitful that knowledge acquisition was labelled the '**bottleneck**' to building knowledge-based systems (Feigenbaum, 1977). It was estimated that this 'traditional' approach (also called the '**transfer approach**') to knowledge acquisition typically produced between 2 and 5 units of knowledge (heuristic rules) per day (Jackson, 1990).

The difficulties arose for many reasons, including the following (Jackson, 1990).

- The jargon of some specialist fields of knowledge may prove to be a barrier to the understanding of the knowledge engineer.
- The facts and principles underlying many domains of interest cannot easily be characterised in the precise mathematical/logical way necessary for subsequent processing and inference by a computer program.
- Experts need to know more than just the shallow factual knowledge of a domain in order to solve problems. For example, an expert might know what kinds of information are relevant to which kinds of judgement, how reliable different information sources are, and how to split hard problems up into easier subproblems.
- Human problem solving expertise often relies on ‘common sense’ knowledge about the everyday world. Such knowledge is so deeply-rooted into our experiences as humans that we may not even realise what we know, or what knowledge we are using in our reasoning. The existence of this **tacit knowledge** can make the knowledge elicitation task formidable.

In addition, there may be pragmatic concerns such as the willingness of the domain expert to cooperate in the knowledge engineer’s task. A busy domain expert might assign a low priority to the knowledge acquisition task, with an associated risk of infrequent and unproductive meetings. In extreme cases, the domain expert might even resent the acquisition of her knowledge. The recognition of these problems led to several different research directions, each attempting to reduce the **knowledge acquisition bottleneck**. The most important of these are *knowledge elicitation*, *machine learning*, and *knowledge-base refinement*.

Knowledge elicitation borrows techniques from clinical psychology in an attempt to make the interactions between the domain expert and the knowledge engineer more productive. **Automated knowledge elicitation** advances this idea by incorporating the same psychological techniques directly into a computer program. Such a program can then be used to assist, or perhaps even replace, the knowledge engineer.

The techniques of **machine learning** do not interact directly with a human expert, but instead with data descriptions of the problem to be solved. The most common machine learning paradigm involves presenting a computer program with a 'training set' of example problem descriptions, together with the known solution in each case. The machine learning program is then able to infer a pattern to the example/solution pairs such that it is afterwards able to solve problems which it has not seen before.

The techniques of **knowledge base refinement** assume that a knowledge base already exists, but typically use knowledge of the inference procedures of the knowledge base to improve its performance. The techniques often help to identify pieces of knowledge which are *incorrect*, *incomplete*, or *inconsistent*. There may also be *redundancy* in the knowledge, which is sometimes an indication of a mistake in the encoding.

I recognise that not all researchers subscribe to the view that *knowledge elicitation*, *machine learning* and *knowledge base refinement* should all be categorised under the same "umbrella" of *knowledge acquisition*. Some of those in the machine learning community, for example, might argue that theirs is a separate discipline, with many applications other than building knowledge-based systems. Many of those interested in knowledge base refinement, on the other hand, might see their task as a specialised form of machine learning. Lastly, perhaps because of the historical roots of knowledge acquisition, many AI researchers might view the terms 'knowledge acquisition' and 'knowledge elicitation' as synonyms.

For each of the three sub-disciplines, however, there is a sense in which their associated techniques *acquire* knowledge. Furthermore, each of the sub-disciplines can be seen as distinct because they deal with different kinds of knowledge source: knowledge elicitation acquires knowledge directly from the *domain expert*; machine learning acquires knowledge from *structured data*; and knowledge base refinement acquires knowledge by improving one or more aspects (e.g., structure, content, performance) of an *existing knowledge base*. There have also been attempts to acquire knowledge from less structured data sources, such as natural language texts (e.g., Wilks, 1997; Biebow & Szulman, 1999). When one thinks of knowledge acquisition, one should therefore consider both the type of agent which acquires the knowledge (a person or a machine), as well as the nature of the source of the knowledge (another person, structured/

unstructured data, a knowledge base). Table 1 summarises the possible interactions, and provides examples of situations where knowledge acquisition of each type occurs.

Acquisition Agent	Knowledge Source	Examples
Person	Person	Student/Teacher relationship; 'traditional' knowledge elicitation techniques.
Person	Unstructured Data	Knowledge engineer reads a book on the subject of interest.
Person	Structured Data	Knowledge engineer interprets a table of numerical values.
Person	Knowledge Base	Knowledge engineer refines a knowledge base 'manually'; i.e., without the assistance of any automated reasoning.
Program	Person	Automated Knowledge Elicitation
Program	Unstructured Data	Knowledge acquisition from texts, diagrams, speech.
Program	Structured Data	Most machine learning techniques
Program	Knowledge Base	Knowledge Base Refinement

Table 1: Types of Knowledge Acquisition According to Acquisition Agent and Knowledge Source

In the rest of this chapter, I provide an overview of *knowledge acquisition* by describing some representative techniques from each of *knowledge elicitation*, *machine learning*, and *knowledge base refinement*. I then report on some important current research themes.

2.2 Knowledge Elicitation

Whereas the term *knowledge acquisition* asserts nothing about the source of the acquired knowledge, *knowledge elicitation* concerns specifically the acquisition of knowledge from a human expert (see table 1). Since the main consideration of this dissertation is *knowledge acquisition for knowledge-based systems*, I adopt the following definition, proposed by Firlej and Hellens.

Definition 2: Knowledge Elicitation

> '(Knowledge) elicitation is ... the process of drawing forth, or evoking a reply from the human expert for the purpose of building a system.' (Firlej and Hellens, 1991).

There are many established techniques for knowledge elicitation, including *interviewing*, *sorting*, *laddering* and the *repertory grid method* (Diaper, 1989). Before describing the mechanics of these methods, I shall mention some important issues related to the application of a method. When planning an elicitation session, the knowledge engineer should consider:

- whether some or all of the knowledge is already documented;
- whether an expert is available and willing to take part in the session;
- whether the expert understands the project objectives;
- how much control can be exercised over the content of the knowledge elicited, and any associated bias in the control procedure;
- the type of knowledge required, and the type of knowledge usually elicited by the proposed method(s);
- the extents to which the elicited knowledge is *certain* (referring to the reliability of an inference), *contentious* (referring to the level of agreement among experts), and *dynamic* (changeable over time);
- whether the knowledge requires some special form of representation, such as graphs, maps, sketches, or sounds (other than speech);
- the effect of the environment on the performance of the expert (environmental cues can significantly affect the expert's recall ability);
- the likely effects of the knowledge-based system, once implemented, on the people and processes at the site of installation.

An important distinction when considering which elicitation method to apply is whether it is **contrived** or **uncontrived**. An **uncontrived method** seeks to observe an expert during problem solving without interfering in the problem solving process. In a **contrived method**, the knowledge engineer interacts directly with the domain expert, and can therefore steer the knowledge acquisition process towards the topics of partic-

ular interest. The main disadvantage of contrived methods is that the domain expert can no longer be viewed in isolation; indeed, the behaviour of the knowledge engineer plays a large part in the effectiveness of the acquisition exercise, and can even harm the experiment by introducing an unwanted bias. For example, when interviewing a domain expert, the language used by the knowledge engineer can carry connotations which influence the domain expert's answers. (Consider the question "How fast was the sports car going when it crashed into the wall?" as opposed to "What was the speed of the vehicle at the time of impact?".)

When choosing a method, there should clearly be a good match between the type of knowledge required and the type generally produced by the method under consideration. For example, can the proposed method elicit class hierarchies, causal knowledge, examples, constraints, facts, goals, explanations, justifications, preferences, procedures, or relations? Cordingley, in (Diaper, 1989) presents a matrix containing knowledge acquisition techniques and their suitabilities for a variety of knowledge types. As an example, it states that although *interviewing* is a good technique for eliciting conceptual structures, facts, and causal knowledge, its efficacy for eliciting rules and assessments of weight of evidence is questionable. Similarly, the *repertory grid method* (see page 19) is good for eliciting conceptual structures, rules and weights of evidence, but bad for eliciting causal knowledge, procedures, and an expert's problem solving strategy.

It is important that elicitation sessions are recorded, so that the expert's performance can be later analysed and a tangible account of events is established for future reference. A verbatim written account of a knowledge engineer's encounter with a domain expert is referred to as a **protocol**. This is usually acquired by recording the meeting onto audio cassette, then later transcribing the discourse. During the meeting, the expert is usually[1] asked to think aloud while solving a particular problem. This reasoning is recorded and becomes part of the protocol. When analysing the protocol, the knowledge engineer identifies the most important concepts and reasoning processes, so that they can later be reproduced by a knowledge-based system. Although some researchers expressed concern that verbalisation of problem solving might interfere

1. Diaper (1989) discusses other approaches.

with the problem solving itself, investigations have shown that protocols acquired in this way are of value (Ericsson & Simon, 1984).

Sometimes, more than one medium might be required to record the problem solving behaviour. For example, when eliciting a technique for solving the Rubik's Cube™, a camera would be needed as well as an audio soundtrack. In any case, it is very important that after the recording session has ended and the expert has left the scene, the knowledge engineer is able to reproduce the expert's problem solving behaviour.

2.2.1 Interviewing

A clinical interview of the domain expert by the knowledge engineer is a common knowledge acquisition technique. There are several different types of interview (Diaper, 1989). In an **unstructured interview**, the knowledge engineer asks probing questions and records the responses. The style of interviewing is flexible, however, so that the domain expert's reaction can be pursued if the direction looks fruitful. One alternative is a **focussed interview**, which concentrates on a single aspect of problem-solving, and covers it in great depth. Another approach is a **structured interview**, in which the knowledge engineer keeps strictly to an agenda, and is prepared with a list of specific questions relating to the domain.

When interviewing a domain expert using any of these techniques, the knowledge engineer is making the assumption that the domain expert's knowledge can easily be articulated. Unfortunately, this is not always the case. For example, an expert historian might have evolved an informal taxonomy of historical events, which she had never needed to articulate. Some other knowledge elicitation techniques assist the knowledge engineer in eliciting domain knowledge which might not otherwise be forthcoming. Two such methods are *sorting* and *laddering*.

2.2.2 Sorting

Sorting is a specialised technique, used for eliciting further knowledge about a *preselected* set of concepts. When sorting, each concept of interest is written on a card, and the domain expert is asked to divide the pack of cards into separate, but meaningful,

piles. The knowledge engineer records the separation and asks the domain expert to explain it. Then the process is repeated, and the domain expert is requested to provide a new logical separation. This continues until the domain expert can think of no more ways to separate the concepts logically. Often, a *sorting* will elicit the kind of knowledge which can be modelled by a concept attribute.

2.2.3 Laddering

Laddering is a technique for eliciting hierarchical and clustering knowledge about a preselected set of concepts. The knowledge engineer starts with a so-called *seed concept* and poses questions such that the domain expert justifies the position of the concept in a hierarchy, and at the same time offers further knowledge. For example, given the concept *sedimentary rock*, one might ask 'Can you give examples of sedimentary rocks?'. This should elicit concepts which are lower in the hierarchy. It is also possible to elicit concepts at the same level in the hierarchy by asking for alternatives, e.g., 'What other rocks are there which are not sedimentary?'; or concepts higher in the hierarchy by asking for commonalities, e.g., 'What do basalt and pumice have in common?'.

2.2.4 Repertory Grid

The **repertory grid** is a technique based on **personal construct theory** (Kelly, 1955). The theory defines a **construct** as a notion which describes some concept and construes it as being alike or different from other concepts. For example, a car's *size*, *running costs*, and *prestige-level* may differentiate it from other cars. Each construct has two associated extreme values, called **poles**, and several allowable intermediate values. Accordingly, a construct called *size* might range from *small* to *large*, a construct called *running costs* might range from *economical* to *expensive*, and *prestige-level* might range from *low* to *high*. The ordinal set of allowable values for each construct is called the **range of convenience**. Note that a construct is not always applicable to a concept. One would not normally speak of the *prestige-level* of a toothbrush, for example, or the *running costs* of a friendship (!).

A concept characterised by constructs in the way described is called an **element**. In my cars example, one might wish to characterise a *Vectra*, an *Escort* and a *Mini*. Figure 5 shows the result of an elicitation. For each element, every construct has been given one of five possible values within the range of convenience.

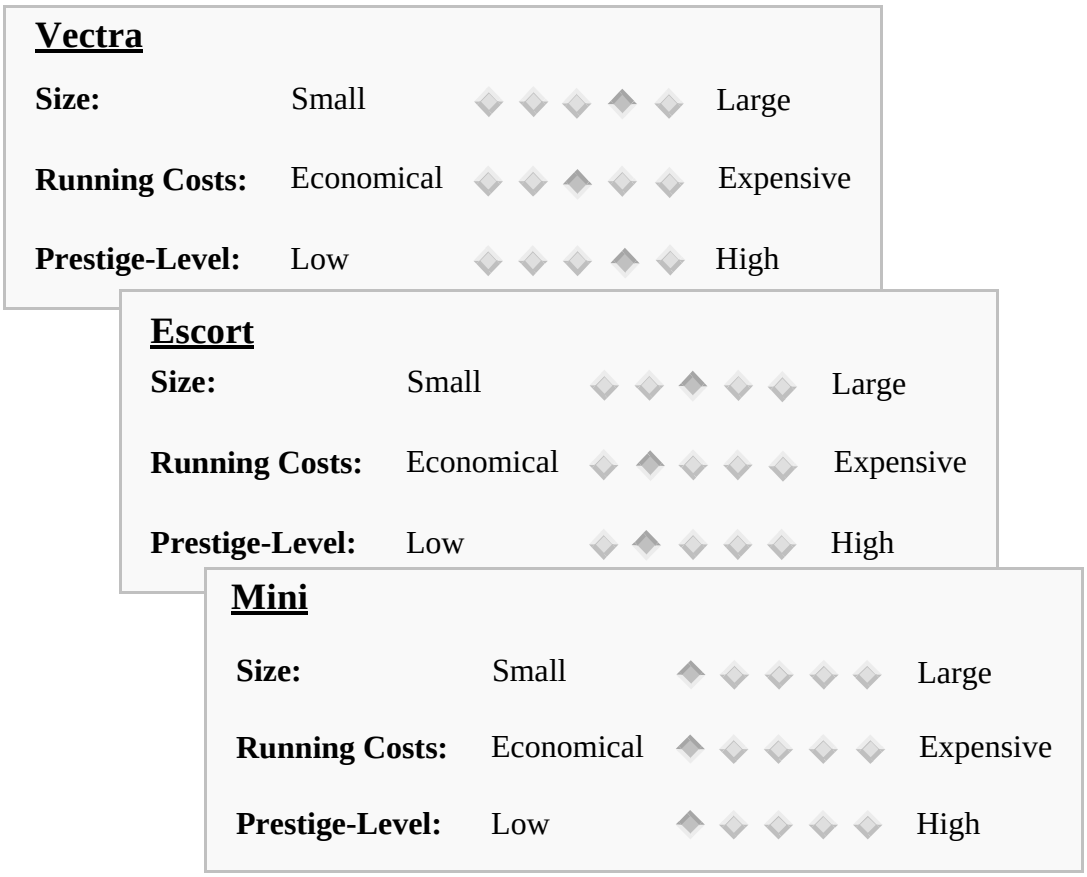


Figure 5: Example Repertory Grids for three different cars

If the meanings of each *construct* and their *poles* are well-defined and used consistently during the elicitation, then the knowledge thus elicited can form the basis for a knowledge-based system. **ETS** and **AQUINAS**, both computerised extensions of the repertory grid method, have been used to derive 'hundreds' of small and medium-sized knowledge-based systems (Boose, 1990).

2.2.5 Automated Knowledge Elicitation

As we have seen, some of the elicitation methods, such as the *Repertory Grid*, have been incorporated directly into computer programs. This was a very natural step, since such methods can be time consuming and laborious to apply manually, and the underlying algorithms are often straightforward (Reichgelt and Shadbolt, 1992). **Automated elicitation methods** were usually implemented initially as separate, stand-alone tools, usually in a domain-independent way, but often assuming the application of a particular problem solving method on the knowledge acquired. For example, **SALT** (Marcus, 1988) was an automated knowledge acquisition tool that acquired knowledge for systems that use the *Propose-and-Revise* method (see "Propose-and-Revise" on page 59). Some researchers focussed on improving the functionality of such tools, or applying those tools in previously unforeseen ways. *Repertory grids*, for example, were enhanced by the **AQUINAS** system by including the notion of hierarchical knowledge for the first time (Boose, 1990), and also used in a specialised form to elicit *operational* knowledge[2] (Wolf, 1999).

Other researchers have concentrated on harnessing the synergy of the different methods by building a (computerised) *workbench* for knowledge elicitation. One of the first of these was a research prototype called **ProtoKEW** (Reichgelt & Shadbolt, 1992). This was subsequently re-implemented and marketed as a commercial product, called **PC-Pack**. It encompassed a suite of integrated knowledge elicitation tools such as a repertory grid, and tools for sorting and laddering. Moreover, unlike many other knowledge acquisition systems, *PC-Pack* is easily portable[3] as it runs on modern laptop computers.

Researchers have also recognised the importance of the elicitation of **consensual knowledge**, that is, knowledge which is acquired from multiple experts. Ideally, the accrued knowledge should also be approved by all the experts, but this can be problematic, particularly since the experts are often geographically distant. Typically, a knowl-

2. Operational knowledge is knowledge which describes actions; i.e., knowledge about change of state.
3. In the sense that the knowledge engineers can take the system with them. This is important because it means that the knowledge engineer can visit domain experts in their own environment, rather than the domain expert having to come to the knowledge engineer.

edge engineer would have to acquire piecemeal accounts of the domain from the experts in separate sessions and then attempt to merge the results. Furthermore, any disagreements among the experts can lead to internal inconsistencies in the acquired knowledge. Software tools can ameliorate these problems. For example, Puuronen & Terziyan (Puuronen & Terziyan, 1999) describe a method for selecting the 'most supported' knowledge from a group of experts; and researchers have now also started to exploit the potential of the Internet for **distributed knowledge acquisition**. For example, there are now implementations of the *Repertory Grid* available on the World Wide Web[4], as well as more sophisticated tools, such as **WebOnto** (Domingue, 1998) for remotely devising domain models.

2.3 Machine Learning

Before describing *machine learning*, we should be clear what is meant by **learning**. Carbonell gives the following definition.

Definition 3: Learning

> 'The ability to perform new tasks that could not be performed before or perform old tasks better (faster, more accurately, etc.) as a result of changes produced by the learning process.' (Carbonell, 1990)

Some researchers see the development of **machine learning** as a promising approach to the reduction of the knowledge acquisition bottleneck. It is now such a large and well-established AI research area that I attempt here only to illustrate the major paradigms which have emerged; more comprehensive introductions to the field are widely available (e.g., Hutchinson, 1994). Carbonell identified four major paradigms of machine learning (Carbonell, 1990): **inductive learning** (e.g., acquiring concepts from sets of positive and negative examples); **analytical learning** (e.g., explanation-based learning and certain forms of analogical and case-based learning); **genetic algorithms**; and **connectionist learning** methods.

4. See, for example, `http://www.csd.abdn.ac.uk/~swhite/repgrid/repgrid.html` and `http://Tiger.cpsc.ucalgary.ca/WebGrid/WebGrid.html`

Definition 4: Machine Learning

When learning is achieved by an automaton[5], rather than a person, it is called *machine learning*.

2.3.1 Inductive Learning

Inductive learning seeks to find a general description of a concept from a set of known examples and (usually) counterexamples. It is justifiably called 'learning' because the resulting description distinguishes not only between the known examples of the concept (a recall task), but can also be used to make predictions about unknown examples. There is an assumption, of course, that the known examples are representative of the whole space of possibilities.

Probably the best-known inductive learning algorithms are those of the **TDIDT** (Top-Down Induction of Decision Trees) family (Quinlan, 1986), which are used for classification tasks. An algorithm of this kind outputs a **decision tree**, which is a piece of procedural knowledge that can assign unseen objects to one of several disjoint classes. In order to do this, the algorithm must first be *trained* with a set of objects which are each of a known class. An example **training set** is given in Table 2. It provides a simple description of four weather attributes taken every Saturday morning over a period of fourteen weeks, together with a classification as to whether or not it rained on that day. The occasions when it rained have been labelled as **positive instances** (P) of the concept to be learned, whilst the occasions when it did not rain have been labelled as **negative instances** (N). The induction task is to use the *training set* to devise a classification scheme for any object of the same type. As an example, by consideration of the data in Table 2, would we expect it to rain if there was a sunny outlook, a cool temperature, normal humidity, and it was windy?

Figure 6 presents a *decision tree* which correctly classifies each example in Table 2, and additionally provides a decision mechanism for previously unseen examples. It

5. 'A machine, robot, or formal system designed to follow a precise sequence of instructions.' (Free On-Line Dictionary of Computing, `http://foldoc.doc.ic.ac.uk`)

Example Number	Attributes				Class
	Outlook	Temperature	Humidity	Windy	
1	sunny	hot	high	false	N
2	sunny	hot	high	true	N
3	overcast	hot	high	false	P
4	rain	mild	high	false	P
5	rain	cool	normal	false	P
6	rain	cool	normal	true	N
7	overcast	cool	normal	true	P
8	sunny	mild	high	false	N
9	sunny	cool	normal	false	P
10	rain	mild	normal	false	P
11	sunny	mild	normal	true	P
12	overcast	mild	high	true	P
13	overcast	hot	normal	false	P
14	rain	mild	high	true	N

Table 2: The Weather Training Set

was produced by an algorithm called **ID3** (Quinlan, 1983), which orders the attributes in the tree such that those which provide the most class-discriminating information are considered first[6]. Note that such a tree cannot be produced if the training set contains examples which have exactly the same set of attribute values, but are of different classes. In such cases the attributes are *inadequate* for the training set; this knowledge can be used to stimulate revision of the training set, such as the addition of a new, discriminating attribute.

Since the TDIDT algorithms must always be told the classification of each example in the training set, it is called **supervised learning**. Some concept learning algorithms are not provided with this class information; these are the **unsupervised learning** algorithms.

6. ID3 regards a decision tree as an information source that generates a “message” (the class of the object). It uses an information-theoretic measure (called information gain) to select the attribute at each choice point.

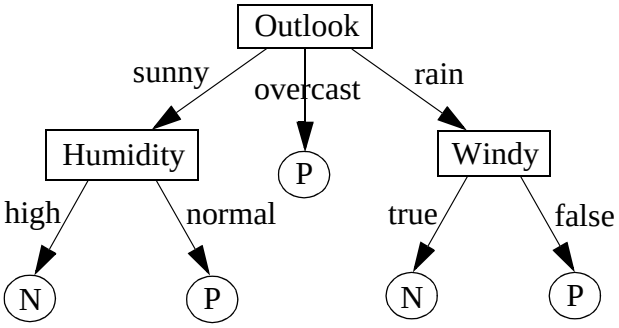


Figure 6: Decision Tree for Weather Prediction

2.3.2 Analytical Learning

I have shown that inductive learners abstract common properties from the many examples provided in a training set. In contrast, **analytical learners** acquire knowledge by applying background knowledge to very few examples (often only one). One of the most common analytical methods is that of **explanation-based learning** (EBL). Explanation-based learners (Mitchell, Keller & Kedar-Cabelli, 1986; DeJong & Mooney, 1986; Minton et al., 1990) generalise from a single example by explaining *why* that example is an instance of the concept. The explanation takes the form of a *proof* that the instance is an example of the concept.

EBG (Mitchell, Keller & Kedar-Cabelli, 1986) is a method for Explanation-Based Generalisation[7] that requires four pieces of knowledge:

- a **goal concept** to be learned;
- a **training example**, which is a positive example of the goal concept;
- a **domain theory**, which provides the means to explain how training examples are members of the goal concept; and,
- an **operationality criterion**, which defines the terms in which the output concept must be expressed.

7. The term ‘Explanation-Based Learning’ was adopted later than ‘Explanation-Based Generalisation’, following the realisation that the EBG approach can be equally applied to specialisation tasks (DeJong & Mooney, 1986).

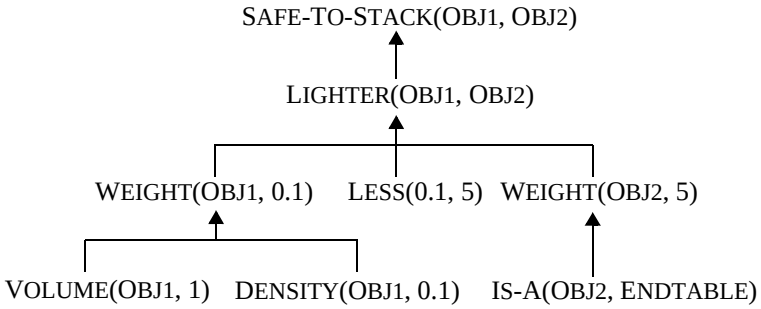
Goal Concept	Pair of objects <x,y> such that SAFE-TO-STACK(x,y) where SAFE-TO-STACK(x,y) ⇔ NOT(FRAGILE(y)) ∨ LIGHTER(x,y)
Training Example	ON (OBJ1, OBJ2) IS-A(OBJ1, BOX) IS-A(OBJ2, ENDTABLE) COLOUR(OBJ1, RED) COLOUR(OBJ2, BLUE)
Domain Theory	VOLUME(p1, v1) ∧ DENSITY(p1, d1) ⇒ WEIGHT(p1, v1 × d1) WEIGHT(p1, w1) ∧ WEIGHT(p1, w2) ∧ LESS(w1, w2) ⇒ LIGHTER(p1, p2) IS-A(p1, ENDTABLE) ⇒ WEIGHT(p1, 5) (by default) LESS(0.1, 5)
Operationality Criterion	The concept definition must be expressed in terms of the predicates used to describe examples (e.g., VOLUME, COLOUR, DENSITY) or other selected, easily evaluated, predicates from the domain theory (e.g., LESS).
Proof (tree)	
Learned Concept	VOLUME(x, v1) ∧ DENSITY(x, d1) ∧ LESS(v1 × d1, 5) ∧ IS-A(y, ENDTABLE) ⇒ SAFE-TO-STACK(x, y)

Table 3: An Example of Explanation-Based Generalisation. The top four rows of the table represent the inputs to the algorithm, and the bottom two rows represent the outputs.

It is assumed that the goal concept fails to satisfy the *operationality criterion* (otherwise the concept description would already be acceptable, and there would be no work to do!). Given these four elements, the task is to find a generalisation of the training example that satisfies the operationality criterion and is a sufficient condition for membership of the goal concept.

Table 3 provides an example of explanation-based generalisation in which the goal concept, SAFE-TO-STACK, predicts whether one object will support another. A posi-

tive example is provided (OBJ1 is stacked on top of OBJ2), as well as a *domain theory* which relates *weights* to *volumes* and *densities*. The task is to explain why the example satisfies the goal concept SAFE-TO-STACK in terms of the predicates used in the examples and domain theory. This has the effect of determining which attributes are relevant to the *goal concept*. The **proof tree** (also in Table 3) explains that OBJ1 is safe to stack on OBJ2 because OBJ1 is lighter than OBJ2. The weight of OBJ1 is derived from its volume and density, whereas the weight of OBJ2 is known through a default rule about ENDTABLEs. Once such a proof has been derived, it is generalised such that new examples might also be similarly explained (see bottom row of Table 3).

2.3.3 Genetic Algorithms

In nature, the population of a species evolves according to principles of natural selection, or the "survival of the fittest"[8]. According to this scheme, the characteristics of an individual which is *fit* (and therefore good at competing for resources) are more likely to be passed to the next generation than the characteristics of an *unfit* individual. **Genetic algorithms** (GAs) attempt to find solutions to problems by mimicking this evolutionary process (Booker, Goldberg & Holland, 1990; Beasley, Bull & Martin, 1993). Such an algorithm supports a population of individuals, each one representing a possible solution to the problem of interest.

To illustrate the idea, I now describe how a basic genetic algorithm might work. The features characterising each individual are represented by **binary attributes**, whose value (either 0 or 1) indicates whether the corresponding feature is present or absent in the individual. A **fitness function** assigns each individual a *score* to indicate how good a solution it is to the problem. These scores are used to help select the best individuals from which to generate 'offspring'[9]. The next generation of the population is produced by a mixture of two operations on individuals from the current population (see pseudo-code in figure 7). The first of these operations is called **cross-breeding**, which is used

8. See Darwin, C., (1859), "On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life", now at `http://www.literature.org/Works/Charles-Darwin/origin/`, and `http://www.bbc.co.uk/education/darwin`

9. A subset of all individuals is chosen, with the selection usually biased in favour of those with a high fitness score.

to create two 'children' from two selected 'parents'. A randomly generated **crossover point** splits the attribute sequences of the parents and decides which 'genes' (attributes) are to be transferred to which of the two offspring (see Figure 8). The second mechanism, called **mutation**, also occurs randomly, but with a low probability. When activated, it simply flips the state of a gene from 1 to 0, or vice versa. Desirable characteristics are thus carried from one generation to the next, with the result that the most promising parts of the search space of possible solutions are explored.

```
BEGIN /* genetic algorithm */
   generate initial population
   compute fitness of each individual

   WHILE NOT finished DO
   BEGIN /* produce new generation */

      FOR (population_size / 2) DO
      BEGIN /* reproductive cycle */
         select two individuals from old generation for mating
         /* biased in favour of the fitter ones */
         recombine the two individuals to give two offspring
         compute fitness of the two offspring
         insert offspring into new generation
      END

      IF population has converged THEN
         finished := TRUE

   END

END
```

Figure 7: Pseudo-code for a Genetic Algorithm;
taken from (Beasley, Bull & Martin, 1993).

Genetic algorithms can be seen as learners whenever the individuals of the population represent solutions to some problem solving task (such as classification). Then the creation of a new population generation corresponds to a modification of the problem solving behaviour, and the application of the fitness function provides feedback on the quality of the current solution. GAs have been used successfully for many tasks, including the prediction of company profitability, deciding how to bet in poker, predicting letter sequences in natural language texts, and learning the past tense of Norwegian verbs (Booker, Goldberg & Holland, 1990). Although proponents of genetic algorithms recognise that the approach does not always deliver the optimal solution to

a problem, they argue that the approach can deliver a “good” solution “acceptably” quickly.

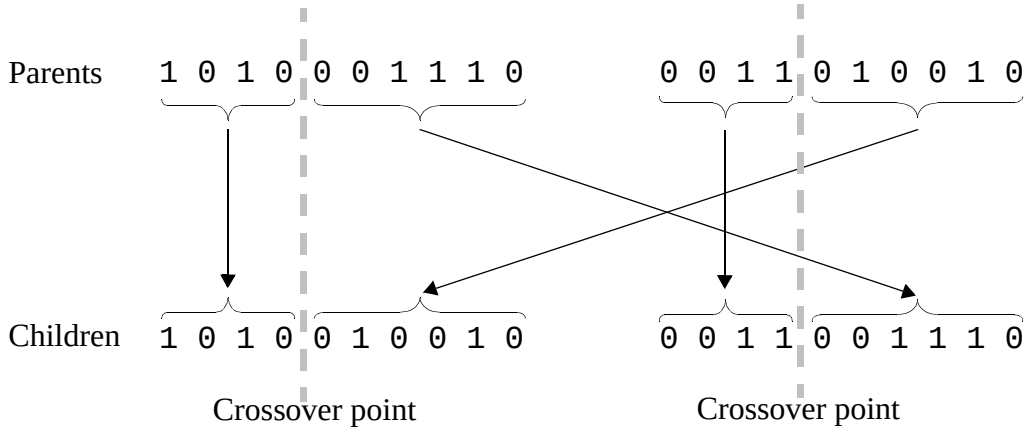


Figure 8: “Breeding” Solutions with a Genetic Algorithm

2.3.4 Connectionist Learning

Many proponents of **connectionist learners** (also called **Neural Nets**) believe that this form of learning is the nearest to that of the human brain. This section considers one of the simplest connectionist algorithms, that of the **simple perceptron** (also called a **simple threshold unit**), and briefly explains the enhancements which led to more complex structures such as the multi-layer perceptron and the backpropagation algorithm.

The *simple perceptron* can be considered to function in a similar way to the light sensitive cells in the retina of the eye. When light falls on one of these cells, it is said to *fire*, meaning that it emits a small electrical charge. This charge might also be the stimulus for other connected neurons to fire. In its simplest form, each cell on the retina is assumed to be either ‘on’ or ‘off’, and the value of each cell is used as an input to a single connected neuron, or node (see Figure 9). The node computes a weighted sum of its inputs and yields an output by comparing this sum to an internal threshold value *t*. In particular, if the inputs are $s_1, s_2, \ldots, s_n$ where $s_i \in \{0, 1\}$ for $1 \le i \le n$, and the weights are $w_1, w_2, \ldots, w_n$, where $w_i \in \mathbb{R}$ for $1 \le i \le n$, then the node’s output is 1 if $\sum_{\ldots} (w_i \times s_i) > t$, and 0 otherwise.

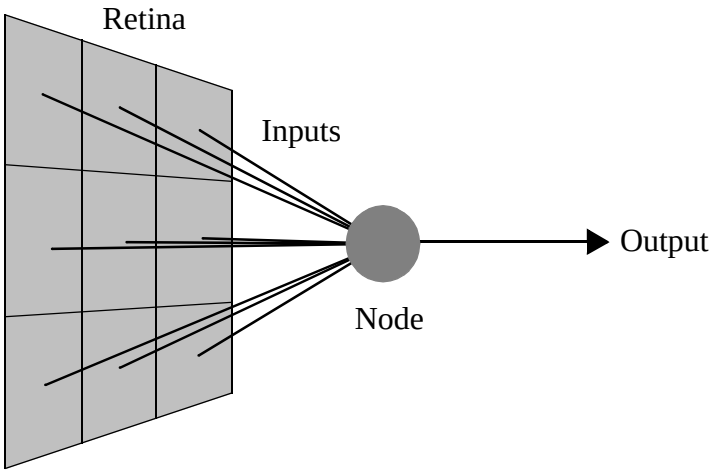


Figure 9: A Simple Perceptron

A perceptron can learn to *recognise* a class C, based on the values of its inputs. If a set of inputs describe a positive instance of C, the perceptron outputs 1; if the inputs describe a negative instance, the perceptron outputs 0. Given a training set of 'images' on the retina, a perceptron can learn by modifying its weights according to the following algorithm:

```
/* Simple Perceptron learning to recognise class C using N inputs */

FOR image ∈ Training_Set DO
BEGIN
   IF image ∈ C AND output(image) = 0       /* a misclassification       */
   THEN
      FOR j = 1 TO N DO
      BEGIN
         wj := wj + (ε × sj);               /* increase weight           */
      END
   ELSE IF image ∉ C AND output(image) = 1  /* another misclassification */
   THEN
      FOR j = 1 TO N DO
      BEGIN
         wj := wj - (ε × sj);               /* decrease weight           */
      END
   ELSE                                     /* correctly classified...   */
      ;                                     /* ...so do nothing          */
END
```

Here, ε is called the **learning rate**, because it dictates how quickly the process converges to the set of weights with the smallest error.

As might be expected, a *simple perceptron* is somewhat limited in what it can learn[10]. For example it cannot learn to recognise a single dot on the retina. A **multi-layer perceptron** is more powerful; it has many nodes which are arranged in layers such that the outputs of some nodes are used as inputs to others. In such networks, each input is usually allowed to take a real value between 0 and 1, instead of just 0 or 1. **Backpropagation** (Rumelhart, Hinton & Williams, 1986) uses two-layer networks in which every node in a layer receives an input from every node in the layer below. The term *backpropagation* refers to the procedure for updating weights across the layers. Starting from the final output, blame for the difference between the actual and desired output (the *error*) is apportioned to the directly contributing nodes. These nodes, in turn, apportion blame to their inputs, with the resulting effect that weight changes are propagated backwards through the network. *Backpropagation* has been successfully applied to many domains, including robot control (Hutchinson, 1994), the mapping of text to speech (Dietterich, Hild & Bakiri, 1990), and speech recognition (Hinton, 1990).

2.4 Knowledge Base Refinement

During the late 1980s, there was also a growing realisation that knowledge acquisition is a *cyclic* process (see Figure 10). That is, the first knowledge base acquired seldom possesses all the required characteristics in terms of accuracy, scope, flexibility, efficiency, and so on. As a result, the knowledge may need to be reorganised as part of a refinement process, or, perhaps more critically, knowledge gaps may be identified which can only be filled by another meeting with the domain expert. Moreover, the knowledge acquisition process needs also to allow for the possibility of *change* in the knowledge that it models. Otherwise a knowledge base in a dynamic domain could very quickly become obsolete. Such considerations gave rise to the field of **knowledge base refinement** (KBR).

10. Since there is only a single threshold value, it can only learn concepts which are *linearly separable.* That is, if a multi-dimensional space of input values is constructed, then a single hyperplane must be able to separate the positive instances from the negative instances of the concept.

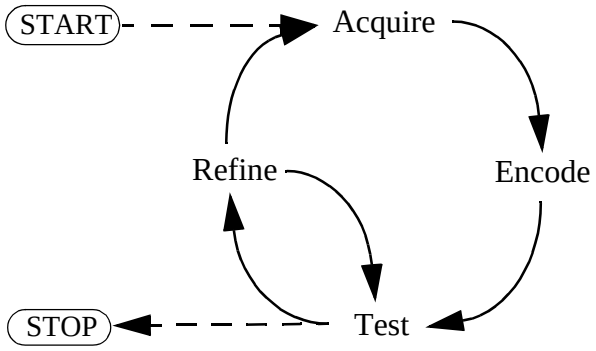


Figure 10: The Cyclic Nature of the Knowledge Acquisition Process

Definition 5: Knowledge Base Refinement

Knowledge Base Refinement is an algorithmic process through which a knowledge engineer improves an existing knowledge base. The nature of the improvement (i.e., whether it concerns accuracy, scope, simplicity, flexibility, etc.) is not predetermined, but will suit the knowledge engineer's purpose.

Since the days of MYCIN and the early expert system shells, knowledge bases have 'traditionally' been represented as sets of rules. For example, the following rules could be used to classify a tree as either oak, beech or silver birch, given its bark colour and texture.

```
IF BarkColour is silvery_white
THEN TreeName is silver_birch;

IF BarkTexture is smooth AND BarkColour is grey
THEN TreeName is beech;

IF BarkTexture is fissured AND BarkColour is brown
THEN TreeName is oak;
```

It is therefore not surprising that most research in knowledge base refinement has concentrated on the revision of such rule sets. A revision is usually undertaken to increase the *classification accuracy* of such knowledge bases, but this is not necessarily the case. Firstly, not all knowledge bases support classification tasks: for example, a knowledge base used in planning could be revised so that the desired goal is achieved

earlier. Secondly, there are other criteria that can be used in the assessment of a knowledge base. A knowledge engineer might consider the simplicity and intelligibility of a knowledge base to be more important than its accuracy.

I now describe some illustrative examples of KBR systems.

2.4.1 TEIRESIAS[11]

TEIRESIAS was developed by Randall Davis as a subsystem of the MYCIN project at Stanford University, and was one of the earliest aides to knowledge base refinement (Davis & Lenat, 1982; Barr & Feigenbaum, 1982; Jackson, 1990). An existing MYCIN knowledge base was tested on a sample problem through Teiresias. If the MYCIN rules provided the same diagnosis as the expert, then their behaviour was satisfactory for that problem. If MYCIN's solution differed from that of the expert, however, TEIRESIAS intervened and used *rule models* to help it identify where the source of the problem might be. These models were effectively **meta-knowledge** of MYCIN's inference engine, which enabled TEIRESIAS to trace through the reasoning used, and explain both *how* and *why* an incorrect diagnosis had been obtained. TEIRESIAS could also help when entering a new rule into the knowledge base, because its rule models contain *associative knowledge* about rule parts. For example, MYCIN rules which identify a class of bacterium tend to comprise three conditions: the *site* and *stain* of the culture, as well as the probable *portal of entry* of the bacteria (Barr & Feigenbaum, 1982).

2.4.2 KRUST / STALKER

KRUST (Knowledge Refinement Using Semantic Trees) (Craw, 1991) is a knowledge base refinement system for the improvement of backward chaining sets of production rules. It assumes that the initial knowledge base is close to the desired knowledge base, and therefore only minor modifications are necessary. Whenever a training example is

11. Teiresias was a blind soothsayer from Thebes, whose advice was sought by Odysseus in Hades. Odysseus, trying to make his way home after the Trojan War, had been told by the enchantress Circe that he would never return safely without consulting Teiresias. Odysseus journeyed to the far western edge of the earth and, on entering Hades, dug a pit and filled it with sacrificial blood to summon the ghosts of the dead. Teiresias was among them, and he gave the needed advice. (See the *Illustrated Encyclopedia of Greek Mythology*; `http://www.mythweb.com/`).

presented for which the conclusions of the knowledge base and the human expert differ, KRUST generates a *set* of possible modifications which improve the behaviour of the knowledge base. Each of the refined knowledge bases is then tested for consistency with other training examples, so that the refinement which performs best over the whole training set can be chosen. This approach is less committed than most other refinement systems, which automatically choose a single refinement based on heuristics coded into the algorithm. On the other hand, testing many alternative refinements can be time-consuming. **STALKER** (Carbonara & Sleeman, 1996a; Carbonara & Sleeman, 1996b) is an improved version of KRUST which addresses this efficiency issue. More recent work has applied KRUST's technology to the pharmaceutical task of tablet formulation (Boswell, Craw & Rowe, 1997), and also moved towards the application of KRUST to knowledge bases implemented in commercially available expert system shells such as **CLIPS**[12].

2.5 Trends and Current Research Issues

To summarise this chapter so far, I have illustrated the diversity of different approaches to the knowledge acquisition task. *Knowledge elicitation* has borrowed some established techniques from clinical psychology, and assisted the knowledge engineer by incorporating these techniques into software tools. The field of *machine learning* has grown dramatically in recent years, developing many different paradigms and approaches, now supported by several separate research communities. The refinement of rule sets no longer has the prominence that it enjoyed in the early 1990s, but the objectives of *knowledge base refinement* are still implicit in the area of *ontological reengineering* (Gómez-Pérez, 1999). In the meantime, there has also been an important change of emphasis within the knowledge acquisition community. It is now widely recognised that to build an intelligent system, knowledge must be very carefully *modelled*.

12. See `http://www.scms.rgu.ac.uk/research/kbs/krustworks/`

2.5.1 The Rise of Knowledge Level Modelling

The **knowledge level** was introduced by Newell (Newell, 1982) as a reaction to the observation that researchers were more involved with the (relatively shallow) representational schemas of AI systems (such as production rules) than with the (deeper) properties of knowledge. He developed a simple model of knowledge-level interaction in which each agent processed *knowledge* in order to perform *actions* and achieve *goals*. In this model, an agent behaves according to the **principle of rationality**, which states that an agent's actions are selected with the objective of attaining the desired goals. Under this scheme, knowledge is defined as 'whatever can be ascribed to an agent, such that its behaviour can be computed according to the principle of rationality' (Newell, 1982). The importance of the knowledge level lay not in the details of Newell's model as such, but in the sentiment that AI research should concentrate its efforts more on describing how things are and why they behave as they do, instead of engineering somewhat superficial (and therefore brittle) simulations of human expertise.

The notion had a profound effect which is still much in evidence in the KA community. In fact, the modelling approach to knowledge acquisition can be seen as a reaction to the introduction of the knowledge level. It differs from knowledge elicitation in that it attempts to represent the most important problem solving concepts (e.g., domain objects, task components) thoroughly, under the assumption that an improved understanding of these elements will aid the reasoning power and maintenance of a KBS. The approach does *not* subscribe to the *transfer view* for building knowledge-based systems (see section 2.1), as explained below by Schreiber, Wielinga and Breuker.

> 'A KBS is not a container filled with knowledge extracted from an expert, but an operational model that exhibits some desired behaviour observed or specified in terms of real-world phenomena. This desired behaviour can coincide with some behaviour as exhibited by an expert. If one wants to construct a KBS that performs medical diagnosis, the behaviour of a physician in asking questions and explaining the problem of a patient may be a good starting point for a description of the intended problem-solving be-

haviour of the KBS. However, a KBS is hardly ever the functional and behavioural equivalent of an expert.' (Schreiber, Wielinga & Breuker, 1993)

Note in particular that a KBS does not have to use the same reasoning process as the human expert whose behaviour it mimics. Without doubt, the reasoning used by the expert could be very difficult to elicit, but, equally importantly, it might be possible to use the advantages of information technology to devise a better (e.g., faster, more reliable, or more general) way of arriving at a solution.

Definition 6: Knowledge Level Model

A *knowledge level model* is 'a model constructed in a manner whereby no specific attention is paid to implementation issues and decisions. Typically such a model (1) expresses some portion of the knowledge required by one or more agents to achieve an existing or required level of problem solving competence; or (2) will be made during an early phase of KBS construction and serve to specify the requirements for subsequent design and implementation.' (Uschold, 1998)

In summary, a **knowledge-level model** of a domain is an implementation-independent description of the objects it contains. The implementation independence of the model frees the knowledge engineer from low-level (e.g., computer language dependent) representation issues until they really matter. On the other hand, since the model is constructed with the intention of building a computer system, it is not constrained to emulate the reasoning process of a human expert.

There have been many attempts to provide a framework for knowledge-level modelling. Some of these concern the modelling *process* itself, and some of them provide specific formalisms for the *output* of the modelling process. The best-known framework, **KADS** (Knowledge Analysis and Design System) (Schreiber, Wielinga & Breuker, 1993), provides guidelines for the modelling process as well as a set of formalisms for its output. One of the aims of KADS was to provide a framework for the construction of knowledge-based systems by applying a divide and conquer strategy to the underlying engineering process. Several *models* were introduced, each of which

emphasises certain aspects of the system under construction. While building one of the models, a developer could temporarily neglect certain other aspects of the system, thus reducing the complexity of the engineering task (Schreiber, Wielinga & Breuker, 1993). KADS comprises the following six different models.

Organisational Model - Provides an analysis of the *environment* in which the KBS will have to function, and attempts to predict how its introduction will affect the organisation and the people working in it.

Application Model - Defines the *function* of the proposed system in the organisation; also states the functional/operational requirements of the KBS such as its speed and hardware platform.

Task Model - Specifies *how* the function of the system (as given by the application model) will be achieved in terms of system tasks.

Model of Cooperation - If the function of the system has been decomposed into subtasks and assigned to several independent 'agents', then this model describes *how the agents interact* when performing those subtasks.

Model of Expertise - Specifies the *expertise* required to perform the problem solving tasks of the system.

Design Model - Describes the *design decisions* which specify how the expertise and cooperation models can be realised.

The main difference between KADS and other mainstream software engineering methods lies in the *model of expertise*. This model would not be needed for the development of most conventional data processing systems, but assumes a prominent role in the development of knowledge-based systems. Several different formalisms have been suggested for representing the knowledge of this model; for example **(ML)^2** (Van Harmelen & Balder, 1993), **CML** (Schreiber et al., 1994), and **KARL**[13] (Fensel, Angele & Studer, 1995). More recently, **UPML**, the Unified Problem-solving Method

13. The Knowledge Acquisition and Representation Language within the MIKE (Model-Based and Incremental Knowledge Engineering) methodology closely followed the KADS model of expertise.

description Language (Fensel et al., 1999) has married the experiences of many modelling languages for knowledge-based systems.

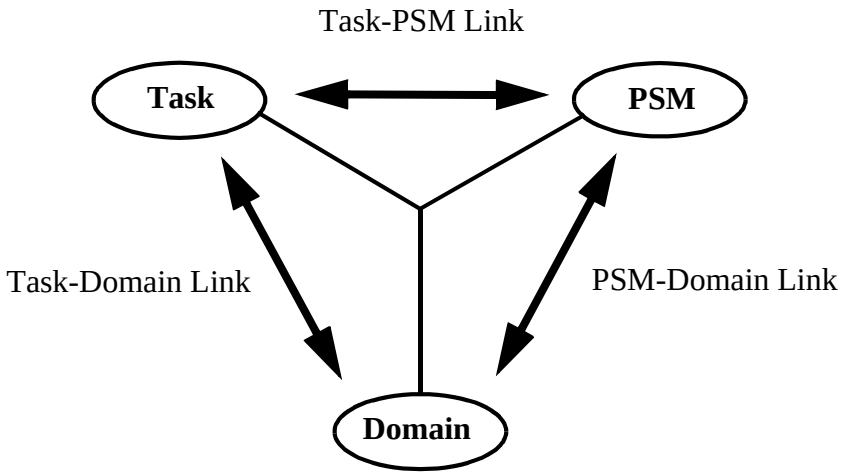


Figure 11: Knowledge Components of a Knowledge-Based System

Since the first separation of control structure and domain knowledge in EMYCIN (see page 12), much has been learned about the automation of the problem solving process. The KA community now distinguishes between *domain knowledge, tasks,* and *problem solving methods* (see figure 11). A **domain** defines an application area, such as cardiology, mass spectroscopy, or linguistics. A **task** defines a problem *type*, such as diagnosis, design, monitoring, or configuration. In this dissertation, I refer to a *particular* task as a **task instance**. For example, a physician who identifies human diseases given knowledge of patients' symptoms performs a diagnosis *task*; but when the diagnosis is of a *particular* patient, it is a *task instance*. A **problem solving method** (PSM) is an inference strategy for solving a task. Some *problem solving methods* can be applied to a range of tasks, but others are more restricted. There have been several attempts to identify and characterise PSMs. Early work (e.g., Buchanan et al., 1983; Chandrasekaran, 1983; McDermott, 1988) identified a small number of PSMs in the hope that these would cover (at least) a large number of tasks. More recently, researchers have recognised that a small number of PSMs is not sufficient for full task coverage, and instead *adapt* existing methods to suit their tasks (e.g., Fensel, 1997; Gennari et al., 1998).

Examples of PSMs are **heuristic classification** (Clancey, 1985), **propose-and-revise** (e.g., Marcus, 1988), and **cover-and-differentiate** (e.g., Eshelman, 1988).

For problem solving to be successful in a KBS, the realisations of these three knowledge components should be mutually compatible (at least) at the knowledge level. If they are compatible at the knowledge level, but not also compatible at the symbol level, then some additional program components can be used to bridge the gaps. For example, a simple problem solver which computes the straight-line distance between two points in a plane might accept only (x,y) coordinate pairs as input. If our 'domain knowledge' specifies points in the plane by polar, rather than Cartesian, coordinates, then the two components are compatible at the knowledge level, but not at the symbol level. In this case, a 'bridge' component would convert polar coordinates to their Cartesian equivalents.

Characterising the relationships between tasks, domains, and PSMs is one of the most challenging issues of current KA research. The Task-Domain link (see Figure 11) is perhaps the least problematic because the experts in any given field already know which tasks they can (and would like to) perform. Therefore, given a suitable taxonomy of tasks, the Task-Domain link can be characterised by a set of ⟨*task instance*, *domain instance*⟩ pairs. One of the earliest attempts to characterise the Task-PSM link was due to McDermott (McDermott, 1988). His research concentrated on the development of knowledge acquisition tools for the so-called **strong methods** (also called *role-limiting methods*). These methods can be applied only to a very limited range of tasks, but the narrow focus also means that strong guidance can be given during the acquisition of knowledge targeted at these methods. For example **SALT** (Marcus, 1988) acquires knowledge for systems that use the *Propose-and-Revise* method, but could not acquire knowledge for systems that use a different method. Likewise **MOLE** (Eshelman, 1988) acquires knowledge only for *Cover-and-Differentiate* systems. Characterising the Task-PSM link for **weak methods** is more difficult because *weak methods*, by their very nature, can be applied to a wide range of tasks. More recently, the PSM-Domain link has been more closely characterised through the explicit statement of *applicability conditions*, or *assumptions*. It has been argued that making the

assumptions of a PSM explicit is a necessary step for their reuse (Fensel, 1995b; Benjamins & Pierret-Golbreich, 1996).

2.5.2 The Promise of Knowledge Sharing and Reuse

Throughout the history of computing, programmers, software engineers and other computer practitioners have wished for components and mechanisms that enable their work to be reused in the same way that mathematicians reuse and apply theorems to different domains, civil engineers modify and reuse building designs, and mechanics reuse physical components such as nuts and bolts. These considerations have pervaded the whole field of computing science, from the design of programming languages, through the architecture of software applications and their subcomponents, to the representation of the data exchanged between separate systems. In the fields of Artificial Intelligence and, in particular, Knowledge Acquisition, much has been spoken about the reuse of domain knowledge and reasoning components. The gains from effective reuse in the field of Knowledge Acquisition are potentially very great, given the amount of effort needed to acquire knowledge from a human expert (consider the ‘knowledge acquisition bottleneck’, described on page 12).

There are two main themes of reuse:

- Reuse of *existing domain knowledge*, either
  - for the same task as originally envisaged, or
  - for a different task.
- Reuse of *existing problem solvers*, or PSMs, in unforeseen problem-solving contexts.

Ontologies have been suggested as a way of tackling the first of these themes. Although there is still much discussion amongst AI researchers on the nature of an ontology, a commonly cited definition is ‘an explicit specification of a conceptualisation’; that is, a formal description of the terms and concepts used to describe a domain. Some researchers take the philosophical viewpoint that an ontology should model some fundamental aspect of the world around us, such as the nature of time, or the mathematical properties of different kinds of numbers. As a builder of information sys-

tems, I take the view that an ontology is a *practical* solution to the problems of reuse. Like a dictionary, it specifies the terms used to describe the domain, and the relationships between those terms. To exchange knowledge between heterogeneous systems, however, the ontology itself must be in a format which is understood by all of those systems. The **Knowledge Interchange Format** (KIF) (Genesereth & Fikes, 1992) is probably the best-known format. It is based on first-order logic, in contrast to the frame-based approach of the **Common Knowledge Representation Language** (CKRL) (Morik, Causse & Boswell, 1991), devised for sharing knowledge amongst components of the Machine Learning Toolbox project (see chapter 4). More recently, the mark-up language **XML**[14] has provided a means for defining simple ontologies and exchanging knowledge between independent systems.

The reuse of existing problem solvers/problem solving methods is a topic which is attracting increased interest, and is also a theme of this thesis (see chapter 6). Many KA researchers speak of the value of a library of problem solving methods, or a toolbox of problem solvers, and some claim to have already built such a library. However, I believe that the libraries which currently exist are probably only of immediate practical use to those who built them, or at most to other KA researchers. There are still many difficulties associated with providing domain experts with the necessary support to select, retrieve, and adapt problem solving methods to solve problems in their domain. But this should not discourage us from trying. The **IBROW** project (Benjamins *et al.*, 1998) is an ambitious attempt to provide a broker for problem solving methods on the World-Wide Web; and, at the 11th European Workshop on Knowledge Acquisition, Modelling and Management (EKAW '99), an informal agreement was made by many of those present to collaborate and provide a simple web-based repository of problem-solving methods. This would be an invaluable resource for the whole community.

2.5.3 The Vision of Knowledge Management

I have already indicated that the research field of Knowledge Acquisition evolved as a side-effect of the difficulties encountered while building and maintaining knowledge-based systems. Initially, KA was seen as the transfer of knowledge from a human

14. See `http://www.w3.org/XML/`

expert to a program. Later, the modelling paradigm of KA gained greater acceptance. In recent years, KA Researchers have realised that the methods they have developed could be applied to activities *other than* building knowledge-based systems. The same techniques can also be applied to **knowledge management** tasks such as acquiring, modelling and distributing the knowledge of individuals and organisations. Some researchers believe that this "third phase" of Knowledge Acquisition will prove to be even more important for the methods it has developed than its original *raison d'être* (Fensel & Studer, 1999).

2.5.4 Other Research Issues

An EPSRC sponsored event which took place in 1996 identified the following issues as important areas of further research in (software assisted) knowledge acquisition (Sleeman & Shadbolt, 1996):

Case studies - There is a need for a series of case studies with current tools and methodologies to assess their full capabilities.

Knowledge reuse - Effective reuse through shared ontologies and libraries of expert systems components.

Diverse knowledge resources - Integration of multiple knowledge sources using formalised approaches like ontologies but being mindful of the constraints imposed by real-world data repositories.

Data mining - the relationship between data mining and knowledge acquisition should be explored.

Acquisition and maintenance methods - tools to enable *knowledge providers* (rather than knowledge engineers) to both create and subsequently maintain knowledge bases.

Emergent KA - KA arising as a side-effect of other routine clerical activity.

Graphical and perceptual KA - KA from graphical and perceptual knowledge.

Distributed KA - Distributed methods on Internet and intranet.

Temporal KA - Methods for the acquisition of dynamic and time-varying knowledge.

Upstream KA - Development of KA methods to support upstream activities such as requirements acquisition, and business process modelling.

Standards - Implementation standards so that a number of disparate tools can cooperate on aspects of a common KA task.

Software dissemination - Packaging of tools such that they work on platforms that have wide industrial uptake.

Another good indicator of current research trends is the list of tracks at an established conference or workshop in the field of interest. The themes of the 12th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop (held in November, 1999) were:

- Knowledge Acquisition for Requirements Engineering;
- Knowledge Management and Knowledge Distribution through the Internet;
- Knowledge Engineering for Intelligent Agent and Internet Applications;
- Cost-Effective Development and Use of Ontologies and Problem-Solving Methods;
- Ontologies and Metadata for Knowledge Retrieval;
- Evaluation of Knowledge Acquisition Methodologies;
- Tools for Adding Domain Knowledge to a Knowledge Base.

Chapter 3

A Brief Review of Constraint Technology

'At the heart of any constraint system
...is a satisfaction mechanism.'
James Gosling

Chapter Summary

This chapter provides some background material on constraint technology in preparation for more detailed discussions in later chapters. It introduces the notions of a *constraint* and a *constraint satisfaction problem*, and describes some of the most common methods for solving constraint satisfaction problems. It also provides a brief survey of some of the most recent and influential constraint satisfaction programming languages. This survey should be seen as further background to my choice of the SCREAMER package for implementing the research ideas discussed in chapters 5, 6 and 7.

3.1 What is Constraint Technology?

Constraint technology concerns the declarative expression of **constraints**, and the methods employed to solve **constraint satisfaction problems** (CSPs). Note that I distinguish between *constraint satisfaction problems* and **constraint programming**, since the latter represents a particular technique for solving the former. In this section, I define these notions, and list some of the important application areas of the technology.

Informally, a *constraint* is a relation which restricts the values of a set of variables. For example, one might constrain the variable *a* to be the arithmetic sum of *b* and *c*; or, when representing two persons with the variables *p* and *q*, one might constrain *p* to be the father of *q*. A *constraint* states the condition that must hold among the variables without implying any *direction of interpretation*: for example, when asserting *a* to be the sum of *b* and *c*, not only are the allowable values of *a* restricted by the values of *b* and *c*, but also the allowable values of *b* and *c* are each restricted by the values of the other two variables. In an implementation of a constraint solving system this becomes very important, since the value of the third variable in such a constraint can often be computed as soon as the values of *any* two of the three variables become bound[1].

Formally, a *constraint* is defined as follows:

Definition 7: Constraint

A *constraint C* is a pair $\langle R, S \rangle$ where *R* is a *k*-ary relation and *S* is a *k*-tuple of variables (Rodosek, 1997).

For example, $\langle Father, (p, q) \rangle$ expresses the constraint that *p* is the father of *q*. The set of variables, *S*, is called the **scope** of the constraint. Constraints with a *scope* of exactly two variables are called **binary constraints**, and those with a scope of three are called **ternary constraints**. Usually, a variable involved in a constraint has a set of known possible values associated with it, called its **domain**. For instance, if $p \in \{fred, george, mary\}$, then the *domain* of *p* is $\{fred, george, mary\}$. Likewise, if a constraint variable *v* is known to belong to the set of all integers, then the domain of *v* is that set. (When writing computer programs to solve constraint problems, domains of infinite size, such as the set of integers, can often be represented by a *type* of the programming language. You can think of the enumeration of domain values as an *extensional* definition of the domain, and the use of a type as an *intensional* definition.)

1. This is not true for all relations. For example, it is not true for addition under modulo arithmetic.

Definition 8: Constraint Satisfaction Problem

A *constraint satisfaction problem* (CSP) is defined by a set of *variables*, each of which has a known domain, and a set of *constraints* on some or all of these variables. The *solution* to a constraint satisfaction problem is a set of variable–value assignments which satisfy the constraints.

A **finite domain CSP** is a CSP in which each variable of the problem has a finite domain. A common example expressed as a finite domain constraint satisfaction problem is that of **cryptarithmetic**. Such problems are usually given by a simple arithmetic equation in decimal integers, except that each *digit* in the equation has been replaced with a single letter. So, for example, the equation 2 + 2 = 4 might be represented by 'X + X = Y'; and 12 × 32 = 384 by 'IN × ON = OUT'. Given only the algebraic version of such an equation, the task is to find a value for each of the letters such that the equation holds. Note that when a digit occurs more than once in the equation, it is always represented by the same letter. Also, since integers are not written with leading zeroes in conventional arithmetic, the left-most digit of an integer has the range 1-9 instead of 0-9.

```
  S E N D
+ M O R E
---------
M O N E Y
```

Figure 12: An Example of a Constraint Satisfaction Problem

A well-known example of a cryptarithmetic puzzle is shown in Figure 12. This problem is sometimes used as a benchmark, and has a unique solution: 9567 + 1085 = 10652. A naive *generate-and-test* approach to finding this solution would assign a value to each of the unknowns and then test to see whether all of the constraints are satisfied. Six of the eight variables have a domain size of ten, and the other two have a domain size of nine, so there are $10^6 \times 9^2$ = 81 million combinations. Constraint

programming techniques are able to significantly reduce the domains of variables *before* values are assigned, making the search for solutions much more efficient. Using such techniques on a modern computer, the above problem can be solved in well under a second[2].

Constraint satisfaction problems occur in many domains, and the techniques used for solving them have very wide applicability. Important application areas include:

- planning and scheduling; i.e., deciding *what* processes to perform and *when* to carry them out;
- (parametric) design tasks, i.e., the construction of some artefact according to some predefined template of the solution;
- spatial layout problems (an important special case of design problems), such as the arrangement of components on VLSI circuit boards subject to design constraints, or the positioning of elements in a computer-generated diagram subject to aesthetic constraints;
- configuration of, for example, electrical components in some system such that the required functionality/performance is achieved;
- the interpretation of textual or visual material.

Note that CSPs can also be used to tackle optimisation problems. This leads to numerous important applications such as cost minimisation in operations research, or finding the quickest way of manufacturing some product as an instance of the job-shop scheduling problem.

3.2 Methods for Solving Constraint Satisfaction Problems

This section reviews some of the best-known methods for solving finite domain CSPs[3], observing first of all that the solution of a constraint satisfaction problem is often seen

2. When repeated 100 times, a quad processor Sun UltraSPARC Enterprise 450 running Allegro Common LISP 4.3 and the SCREAMER constraint satisfaction package solved the problem in an average CPU time of 0.25 seconds.
3. The discussion in this chapter is restricted to methods for solving *finite-domain* problems; later I also discuss the specification of *infinite* domains (see, for example, the discussion concerning the definition of user-defined types from page 134).

as a **search problem**. This is significant because the study of AI has much to do with the investigation of search problems[4].

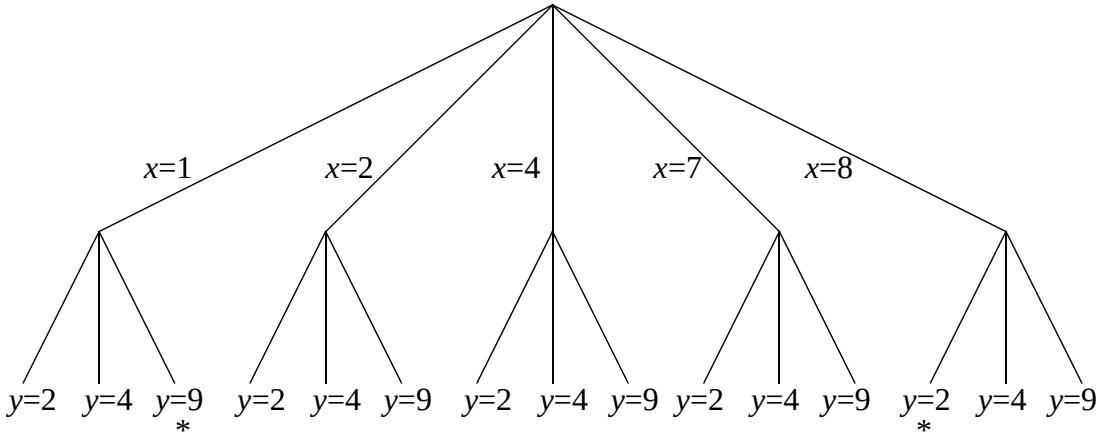


Figure 13: Search Tree for a Simple CSP

It is easy to see how the view of 'constraint satisfaction as search' has emerged. If each variable in a CSP has a finite domain, then perhaps the most obvious way to find a solution is to try out all the possible combinations of variable-value assignments until all constraints are satisfied. So, for example, suppose the sum $x + y$ is constrained to equal 10, and it is already known that $x \in \{1, 2, 4, 7, 8\}$ and $y \in \{2, 4, 9\}$. Then one might provisionally assign x the value 1, and y the value 2. Now the sum $x + y$ conflicts with the given constraint, so one might then retract the value given to y, and try the value 4 instead. This still fails, so one might try $y = 9$. This assignment satisfies the constraint, so the point $x = 1, y = 9$ is a solution. If more solutions are desired, then a different value for x must be tried (since the values for y are exhausted), and then retry the values for y. The approach described is called **backtracking search**, and is a fundamental technique for constraint satisfaction. Figure 13 shows the search tree for this problem, in which every leaf node represents a potential solution to the problem. In fact there are only two solutions, each indicated in the figure with an asterisk.

The simple backtracking method described above is a reliable way[5] of constructing a solution, but unfortunately can be very slow for more difficult problems. Many

4. The following non-trivial tasks have been solved by searching: finding a proof for a theorem in logic or mathematics, generating a possible synthesis path for organic chemistry, parsing natural language sentences, finding the shortest path connecting a set of non-equidistant points (the "travelling salesman problem"), and determining moves in chess (Gardner, 1981; Sleeman, 1992).

methods have therefore been developed which improve upon simple backtracking. In the following sections, I describe several of these improved methods, dividing my account into descriptions of three basic method types: *problem reduction*, *constructive search*, and *repair-driven search*.

3.2.1 Problem Reduction Methods

Problem reduction methods *preprocess* the problem before embarking on a search, with the aim of reducing the original problem to an easier problem with the same set of solutions. This is usually achieved by removing redundant values from variable domains. Of course one needs to invest computational effort into the recognition of redundancy, but the reduction in the size of the search tree is often, though not always, a very good pay-back.

3.2.1.1 Achieving Node Consistency

The simplest form of problem reduction is to achieve **node consistency**, defined as follows.

Definition 9: Node Consistency

A variable is said to be *node consistent* if, and only if, all values in its domain satisfy the unary constraints on that variable.

A CSP is *node consistent* if, and only if, every variable of the problem is node-consistent. The idea is to eliminate those domain values which are inconsistent with the unary constraints of their variables. Suppose, for example, that in the $\langle Father, (p, q)\rangle$ problem, in which the domain of p is $\{fred, george, mary\}$, the unary constraint that p must be male is added. Since Mary is (presumably) not male, achieving node consistency would mean removing her from the domain of p. Such changes require relatively little computational effort[6], but can be very influential in reducing the size of some

5. It is both *sound* and *complete*. That is, all returned values are solutions to the problem, and, given enough time, all solutions are returned.
6. For each variable, the amount of effort is directly proportional to the size of its domain.

problems. Furthermore, it is very easy to devise an algorithm that achieves node consistency in a CSP. One must simply consider each variable in turn, and delete any domain values which do not satisfy all the unary constraints on that variable.

3.2.1.2 Achieving Arc Consistency

A more effective, but slightly more costly approach, is to consider binary constraints, and perform a similar form of problem reduction. Consider the two constraint variables x and y. To achieve **arc consistency**, one removes from the domain of x all domain values which are inconsistent with the constraint on x and y. Similarly, one removes from the domain of y values which are inconsistent with the same constraint. The formal definitions for the *arc consistency* of a single arc, and of a CSP, are given below.

Definition 10: Arc Consistency

An arc in a constraint network is said to be *consistent* if, and only if, for every value a in the domain of x which satisfies the constraint on x, there exists a value in the domain of y such that $\langle x = a, y = b \rangle$ satisfies the constraint on x and y.

A CSP is *arc consistent* if, and only if, every arc in its constraint network is *arc consistent*.

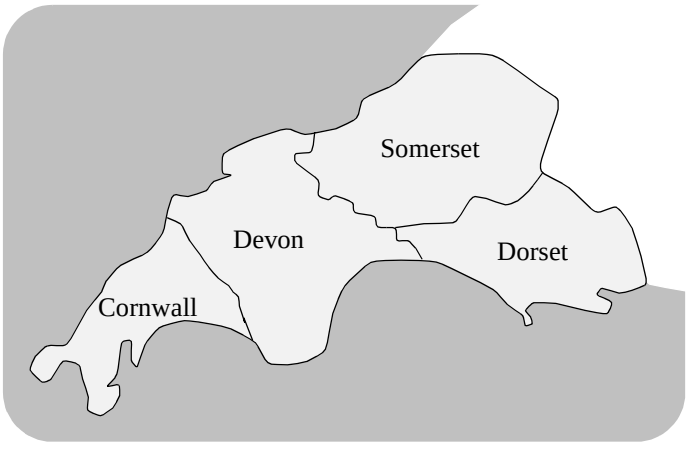


Figure 14: A Map Colouring Problem

The arc-consistency reduction technique can be illustrated by considering the *3-colour map colouring problem* with the four counties of the South-west peninsula of England: Cornwall, Devon, Somerset and Dorset (see Figure 14). The objective of the problem is to colour the map's regions with three different colours such that no two regions sharing a frontier are the same colour.

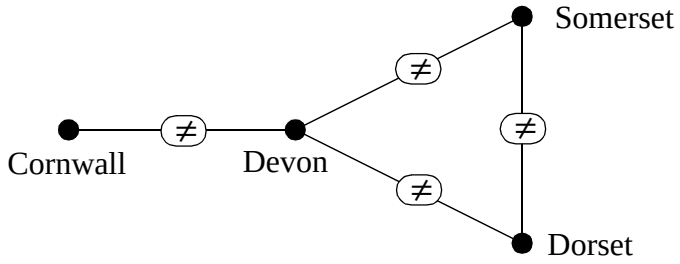


Figure 15: A Constraint Network of the Map Colouring Problem

When the problem is viewed as a CSP, each county represents a constraint variable which must be assigned a value (its colour). The problem can be represented as a **constraint network**, which portrays the variables of the problem as the nodes of a graph (see Figure 15). In the graph, arcs are drawn between variables which share a constraint. I have also labelled the arc with the nature of the constraint (in this case, all arcs are labelled with the disequality relation). In the example, suppose that the three available colours are {`red`, `green`, `blue`}, and that we would like *Dorset* to be coloured `blue`. Then to achieve *arc-consistency*, `blue` would be removed from the domains of *Somerset* and *Devon*, because their colour values must be different to that of *Dorset*.

Several algorithms have been published which achieve *arc consistency*. One of the simplest, called **AC-1**, is a relatively straightforward algorithm, but has a disappointing time complexity. If n is the number of variables in the problem, d is the maximum domain size of the variables, and a is the number of arcs in the constraint network, then the AC-1 algorithm has a time complexity of $O(d^3na)$ (Tsang, 1993). A later algorithm, called **AC-4** in the literature, contains more complex ideas and achieves a time

complexity in $O(ad^2)$. Unfortunately, AC-4 requires a large amount of space to record and maintain the relationships between variables as it proceeds. As a result it has an asymptotic *space* complexity of $O(ad^2)$ (Tsang, 1993).

Note that arc consistency does not imply node consistency. A CSP which is both arc consistent and node consistent is said to be *strongly arc consistent*.

3.2.1.3 Achieving Path Consistency

A **path** through a constraint network is an ordered list of constraint variables, such that each adjacent pair of variables in the list share a binary constraint (i.e., each adjacent pair of variables are joined by an arc in the network). The **length** of a path refers to the number of variables contained in the list. The consistency of a path is given by a property, called **k-consistency**, which requires that for every path of length *k*, each combination of $k-1$ variables leaves at least one consistent value for the remaining variable in that path. Note that *arc consistency* is a special case of *k-consistency* for which $k = 2$, and *node consistency* is a special case of *k-consistency* for which $k = 1$.

Analogous algorithms to AC-1 and AC-4 exist for achieving *path consistency*, but unfortunately they have prohibitive time complexities. PC-4, an algorithm for achieving *path consistency* that is analogous to AC-4, has a time complexity in $O(d^3n^3)$, and a space complexity also in $O(d^3n^3)$ (Tsang, 1993). In practice, therefore, one usually undertakes to achieve no more than *arc consistency* when tackling real problems.

3.2.2 Constructive Search Methods

Constructive search methods begin with an empty set of value-variable assignments. The set of assignments is *constructed* as the search proceeds. At each step, one more variable is assigned a value, and the constraints are checked. As soon as this occurs for the last of the variables without a constraint violation, one can be sure that a solution has been found.

3.2.2.1 Variable Ordering Strategies

The description of backtracking search given above was a straightforward *generate-and-test* approach in which all possible values for each of the variables in the problem are tried in turn until either a solution is found, or the choices of values have been exhausted. No attention was paid to the order in which values were tested. However, significant gains can be made if the order of provisional assignment is chosen carefully. The technique employed for making this choice is called a **variable ordering strategy**, and there are two basic types: **static orderings**, and **dynamic orderings**. A *static ordering strategy* analyses the variables and their associated constraints before starting the search, and arranges the choice of variables into the order which looks most promising. A *dynamic ordering strategy* reconsiders the ordering of variables as it searches the tree. This entails more computational effort than a static ordering strategy, but also has more scope for pruning the search tree.

Two basic ideas related to the ordering of a set of variables are the **position** of a variable in an ordering, and the **distance** between two variables. In an ordering of *n* variables $\langle x_1, x_2, x_3, \ldots, x_n \rangle$, the *position* of a variable refers to its location index within the ordering. So, for example, the position of x_3 in the given ordering is 3, and the position of x_n is n. The *distance* between two variables in an ordering is the absolute value of the differences of their positions. So, for example, if x_i has position *i*, and x_j has position *j*, then the distance between those two variables is given by $|i - j|$.

Minimal Bandwidth Ordering

The **bandwidth of a variable** in an ordering of variables is the maximum distance from it to any of the variables connected to it in the constraint network. The **bandwidth of an ordering** is the maximum bandwidth of its variables. **Minimal bandwidth ordering** is a *static* ordering strategy which minimises the distances between constrained variables. The idea is that when following an ordering in which these distances are minimised, one would not expect so much backtracking after a failure. For example, consider again the map colouring problem of figures 14 and 15. If the counties were ordered such that the colour of Cornwall were chosen first, and the

colour of Devon last, then impossible assignments for Somerset and Dorset might be unnecessarily explored.

Minimum Width Ordering

The **width of a variable** is the number of connected variables that precede it in a variable ordering. The *width of an ordering* is the maximum width of the variables in that ordering. The minimum width ordering of a CSP is the ordering with the smallest width. *Minimum Width Ordering* is a static ordering strategy which places the most connected variables near the front of the ordering. The idea is that the most connected variables are likely to be the most difficult to instantiate, so they should receive values first. In this way, it is hoped that most of the backtracking occurs in the top part of the search tree.

An algorithm which finds the minimal width ordering of a CSP in $O(n^2)$, due to Freuder, is given by Tsang (1993).

Maximum Degree Ordering

A similar strategy which requires less computation is the **maximum-degree ordering**. The principle is the same – put the most heavily constrained variables at the start of the ordering. This time, however, the strategy demands that the variables are ordered according to the *degrees* of their corresponding nodes in the network, with the variable of maximum degree to be chosen first.

The Fail-First Strategy

The **fail-first strategy** is a dynamic ordering strategy whose justification is similar to those for the minimum width ordering and maximum-degree ordering. The strategy is to order the variables in increasing order of *domain size*. When used with simple backtracking, this can be done in advance as a static ordering, since the domain sizes of variables do not change during the search. In general, however, the approach can be combined with other domain reduction methods such as **forward checking** (see section 3.2.2.2). In such cases, the reordering of variables occurs at every choice point, and has great potential for improving the efficiency of the search.

3.2.2.2 Consistency Maintenance

Suppose we wish to solve the "Eight Queens Problem"[7]. The objective of this problem is to place eight queens onto a chess board so that no queen "attacks" any other. A queen is said to *attack* another queen when it lies on the same row, column or diagonal of the chess board. (In chess terminology, "rows" are known as *ranks*, and "columns" are known as *files*.) The problem can also be generalised, such that the objective is to place n queens on an $n \times n$ board ($n \geq 4$), with no queen attacking any other. The "n-queens problem" has become a standard benchmark for testing constraint solving algorithms (Minton et al., 1992).

Forward Checking

The eight queens problem can be solved with a simple backtracking approach. I might start by placing a queen on the first square of the first rank. I know that no more queens can be placed on the same rank, so I look for the next possible space on the second rank. The third queen would be placed on the first possible square on the third rank, and so on. If no spaces are left for one of the later queens, I must backtrack and make a different choice for one of the earlier queens.

This method finds the solutions, but is wasteful because it checks combinations which I should already know cannot be solutions. One way of eliminating such combinations before they are tested is called **forward checking**. As soon as the first queen has been placed on the first square of the first rank, I know not only that no other queen can be placed on the same rank, but also that no queen can be placed on the first file or the long diagonal. The corresponding domain values of the other variables should therefore be eliminated immediately to reflect this knowledge. Likewise, when the second queen is placed, further domain values can be removed from the other variables. In general, every time a choice is made, I can immediately remove any values inconsistent with that choice from the domains of the remaining variables. Furthermore, whenever a choice is made for a variable, it is guaranteed to be consistent with any previous variable choices, because the inconsistent values were removed from the domain. The

7. The Eight Queens Problem was first published in 1848 in the German chess magazine *Schach*. The problem was generalised to n-queens by Netto in 1901 (Russell & Norvig, 1995).

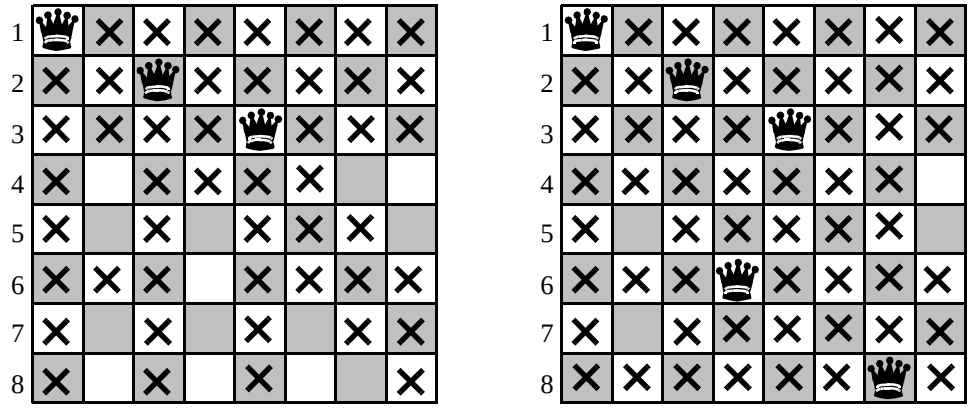


Figure 16: Forward Checking (i) and Generalised Forward Checking (ii) in the Eight Queens Problem

forward checking procedure does represent some extra work, but it will very often save computational effort over the whole problem because of the extent to which it reduces the search space.

After placing just three queens in the Eight Queens Problem, the remaining search space has already more than halved in size, and the domain size of the variable representing the queen on the third rank has been reduced to just one (see Figure 16(i)).

Generalised Forward Checking

An improved version of *forward checking* is called **generalised forward checking**. This behaves the same as *forward checking*, except that whenever the domain size of a variable becomes one, that variable is assigned the single remaining domain value immediately. Any values which can be eliminated from the domains of other variables as a result of this assignment are also immediately removed. In Figure 16(i), for example, a queen would immediately be placed on the sixth rank, and the newly attacked squares would be removed from the domains of the other variables. This leaves only a single domain value for the queen on the eighth rank, so the *generalised forward checking* procedure would again be applied, placing the queen on the eighth rank and eliminating more squares (see Figure 16(ii)). The elimination procedure continues until a solution is found, the elimination runs to exhaustion without finding a solution (in which case the algorithm continues with its search), or a failure is gener-

ated (upon which the algorithm backtracks to the most recent choice point to try a different value).

Maintaining Arc Consistency

A procedure called **maintaining arc consistency** reduces the search space even further than *generalised forward checking*. Maintaining arc consistency is similar to generalised forward checking, except that propagation occurs not only when the domain of a variable is reduced to a single value, but *whenever* the domain of a variable is reduced. This can reduce the search space quite considerably, but it also means that much time is spent maintaining consistency rather than searching. For this reason, researchers believed that *maintaining arc consistency* was overkill, and usually favoured some form of *forward checking*. In 1994, however, the supposed superiority of *forward checking* was challenged when some problems were discovered that were solved much more quickly by *maintaining arc consistency* than by *forward checking* (Sabin and Freuder, 1994). It appears that *maintaining arc consistency* outperforms *forward checking* for hard problems, when the search space is very sparsely populated with solutions.

3.2.2.3 Backjumping

The *forward checking* techniques described above improved upon the behaviour of *simple backtracking* by making better forward moves in the search tree. It is also possible to improve the basic algorithm by refining the backtracking procedure.

Consider, for example, the queen placements shown in Figure 17. After these five queen placements there is no valid square left for the next queen on the sixth rank. Now, instead of backtracking blindly to the most recent choice point, one looks to see which of the recent choice points might rectify the current failure. Upon inspection, we realise that *regardless* of the position of the queen on the fifth rank, the position of a new queen on the sixth rank will *always* fail. One can therefore jump straight back to choose a new position for the queen on the fourth rank without any fear of missing solutions.

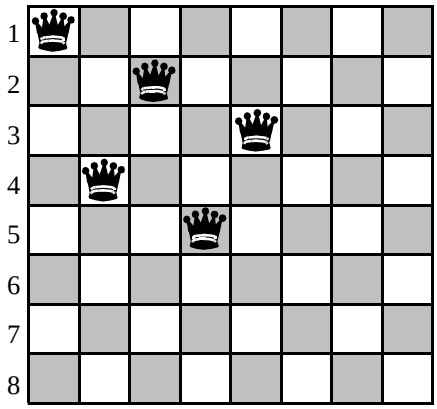


Figure 17: Backjumping in the Eight Queens Problem

To implement this method, one must simply remember the deepest variable that was checked against when generating the current failure.

3.2.2.4 Other Constructive Search Methods

I have described several of the most important constructive methods for constraint satisfaction. I acknowledge that there are many other methods which cannot be covered in this short review. You should refer to (Kumar, 1992) and (Dechter & Frost, 1998) for a more complete survey of established methods[8]. An important observation is that different techniques can be combined, sometimes giving results that are better than either of the two algorithms alone. For example, one such "hybrid" algorithm is **BJ-FC**, a combination of backjumping and forward-checking proposed by Prosser (Prosser, 1993).

Note that there is an important distinction between *systematic* and *non-systematic* constructive search algorithms. The algorithms described so far are all **systematic search procedures** because, given enough time, they are guaranteed to find all solutions to a problem from a single run of the algorithm. **Non-systematic constructive search procedures** search a smaller portion of the tree and therefore execute faster than their systematic counterparts, but run the risk of missing solutions. **Limited discrepancy search** and **iterative sampling** (Harvey, 1995) are two such algorithms.

8. Henz & Müller give a less comprehensive, but well-written overview of finite domain constraint programming (Henz & Müller, 2000).

3.2.3 Repair-Driven Search Methods

The **repair-driven search methods** differ from the constructive search methods in that instead of beginning with an empty set of variable-value assignments and iteratively *adding* new assignments, they start with a complete set of assignments which do not necessarily satisfy the constraints. They then iteratively *modify* the assignment set until either a solution is found or the algorithm concedes that it cannot find a solution. The process can still be seen as a search procedure, but one whose search operators are modification-based.

3.2.3.1 Min-Conflicts

The **Minimum-Conflicts** algorithm (Minton et al., 1992) starts with a complete set of variable-value assignments. (The algorithm generally requires a good set of initial assignments. If no set of assignments is available, heuristics may be applied in an attempt to generate an initial state which is close to a solution.) The algorithm then randomly selects one of the variables whose value violates a constraint, inspects all possible instantiations of the variable, and chooses the value which results in the fewest conflicts. If two or more values result in the same (minimal) number of conflicts, the choice between these options is made randomly. For some problems, this simple algorithm is very good at finding a solution. It has been used, for example, to solve the 1000000-queens problem. The method is best applied to large problems with many solutions. (Small problems can already be solved with more systematic search procedures, and the algorithm may not find solutions to a tightly constrained problem.)

3.2.3.2 Propose-and-Revise

Unlike the other methods described in this chapter, which used *domain-independent* heuristics, the **Propose-and-Revise** method (Marcus, 1988) uses domain dependent heuristics to help direct it towards a solution to the problem. As the name suggests. the method consists of two parts. The *propose* part uses domain knowledge to conjecture a solution to the problem. If constraints are violated, the method then applies domain-dependent *fixes* (heuristic-based modifications) to revise the solution approximation.

Note that the conjectured solution must normally be a good approximation to the final solution for the method to work.

Although both the propose and revise parts apply domain-dependent knowledge, this knowledge can in practice be separated from the domain-independent method which applies it. As a result of this separation, the method can be customised to a new application domain by eliciting domain-dependent problem-solving knowledge from domain experts in a piecemeal fashion. This was the premise behind the **SALT** knowledge acquisition tool (Marcus, 1988).

3.3 Constraint Programming Systems

Constraint programming is the explicit expression and application of constraints within some programming language. The programming language can be of any type, for example, an imperative language like C, a functional language such as LISP, or a logical language like PROLOG. It is commonly argued that *constraint programming* is a declarative paradigm, since constraints are usually a closer expression of *what* the problem is, rather than *how* to solve it. This, combined with the fact that constraint programming can tackle hard combinatorial problems through efficient search techniques, makes the paradigm very attractive:

> ‘Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.’ (Freuder, 1997)

This section provides a representative overview of some of the systems devised for constraint programming and the solution of constraint satisfaction problems. Note in particular that some of the systems have been developed as research projects, and others are commercially available packages. I present the survey in approximate chronological order, and have divided my account accordingly into three generations: the *formative* systems, the *consolidated* systems, and the *progressive* systems.

3.3.1 Formative Systems

The **formative systems** were ground-breaking because they demonstrated new ways of solving problems computationally, and had major influences on the directions of later work.

3.3.1.1 Steele's Constraint System

Guy L. Steele, Jr. provided one of the earliest contributions to constraint programming with research carried out at the MIT AI Research Laboratory (Steele, 1980). In his PhD thesis, he likened a constraint program to a network of devices connected by wires. Data values may flow along the wires, and computation is performed by the devices using only locally available data values. A device can then place new values on some attached wires, with the result that computed values are *propagated* through the network. For example, consider the relationship between temperatures given in degrees Fahrenheit and Celsius, namely $Centigrade = \frac{5 \times (Fahrenheit - 32)}{9}$. This relationship can be expressed by a constraint network, as given by Figure 18. Now suppose that centigrade has the value –40. Then the device $MULT_2$ computes the result $9 \times -40 = -360$, and propagates this along the network as the value of the constraint variable B. Since B is also equal to $5 \times A$, $MULT_1$ computes the value of A to be –72, and propagates this further to the device $ADDER_1$. $ADDER_1$ sums –72 and 32 and posts the equivalent Fahrenheit temperature, –40 [9].

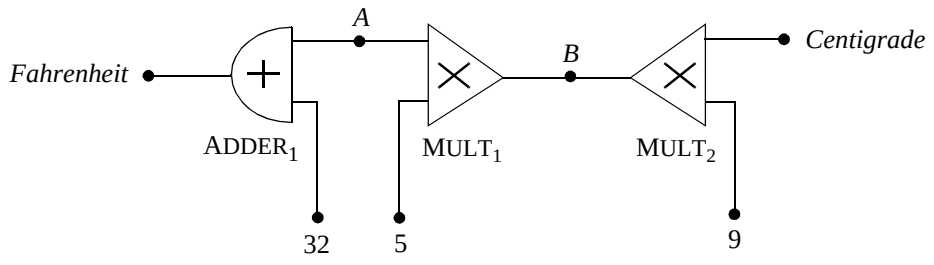


Figure 18: A Constraint Network for Converting Temperatures[10]

9. This is not a mistake: the temperature -40 degrees Celsius is also -40 degrees Fahrenheit.
10. Adapted from (Steele, 1980)

Steele went on to implement a system of local propagation on a LISP machine. Steele's goal was to provide a complete programming system which would implicitly support the constraint paradigm to the same extent that LISP, say, supports automatic storage management. Although he did not fully achieve his goal, he laid foundations for much of the later work on constraint programming.

3.3.1.2 CLP(*R*)

Some other important early work on constraint logic programming was carried out at Monash University in Melbourne, Australia, by Joxan Jaffar and Jean-Louis Lassez around 1987. They defined a scheme for constraint logic programming over any given constraint store, and designed it as an extension of logic programming. They gave it the generic name **CLP(*X*)**, where the X defines the domain of computation. For example, a system implementing the scheme for finite domain variables is called CLP(*FD*), and a similar system dealing with *Booleans*, *naturals* and *reals* is called CLP(*BNR*). The pioneering system of Jaffar and Lassez solved constraints over the domain of real linear arithmetic, and was called CLP(*R*). The constraint solver dealt only with linear constraints, but the system also contained a mechanism for delaying non-linear constraints until they became linear. This meant that even non-linear constraints could be dealt with in some situations.

3.3.2 Consolidated Systems

The **consolidated systems** did not introduce so many new ideas, but implemented the ideas inherited from earlier systems efficiently and reliably. Through these systems, constraint technology became more widely available, and was applied to real-world problems.

3.3.2.1 CHIP

CHIP (Constraint Handling in PROLOG) was one of the most influential constraint logic programming systems, originally developed at ECRC[11] in Munich between 1985

11. The European Computer-Industry Research Centre, jointly funded by Bull, Siemens, and ICL

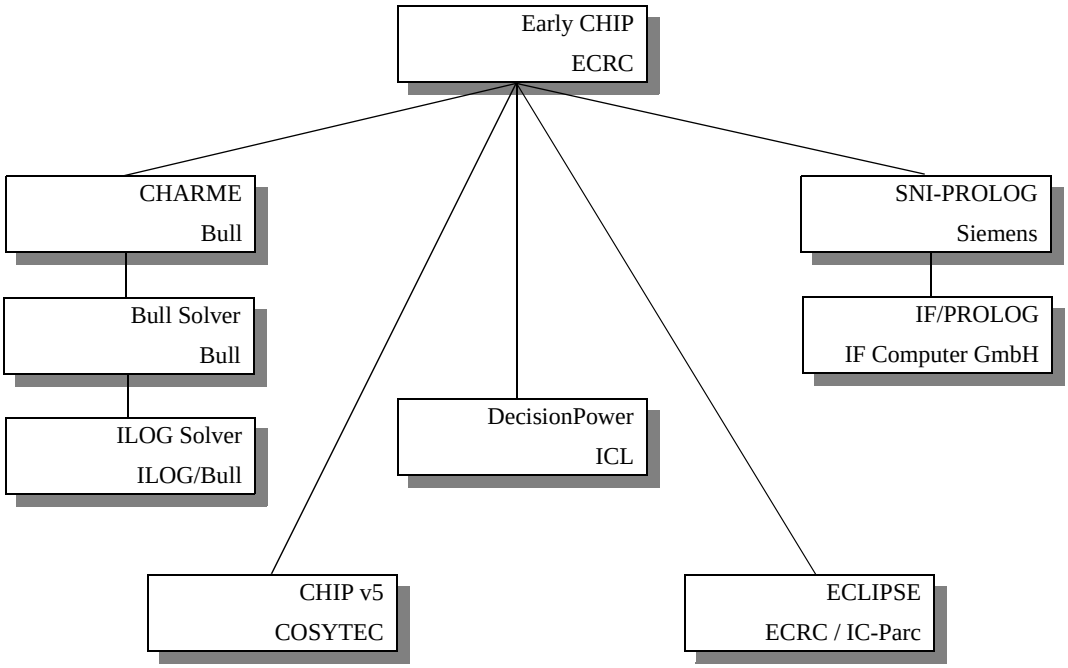


Figure 19: The CHIP Family Tree

and 1990 (Simonis, 1995). The early versions of CHIP also lead to a number of related systems by ECRC funding companies. In 1989, Bull re-engineered the system to produce a constraint based system with a C-like syntax, called **CHARME**. ICL also took inspiration from CHIP to develop a system called **DecisionPower**, and Siemens used constraints in their own implementation of PROLOG, **SNI-PROLOG** (later sold as IF/PROLOG). In 1990 the team which had designed CHIP left the ECRC and set up a company called COSYTEC, based in Paris, where they have continued to develop CHIP. CHIP v5 has recently been released, and is available not only in PROLOG, but also in C and C++ versions. The ECRC's own successor to CHIP is called **ECLIPSE**, now being maintained and further developed by a university-industry collaboration at London's Imperial College, called IC-Parc. See Figure 19 for an overview of CHIP and its derivatives.

Since CHIP is based on PROLOG, a backtracking search for the solution of a set of constraints can be seamlessly integrated with conventional PROLOG terms. More specifically, CHIP extends the host language with a convention for the representation of constraints and the functionalities of several specialised constraint solvers. The

finite domain solver was at the core of the original version of CHIP. It offered basic arithmetic constraints and some symbolic constraints. This was later augmented by a *Boolean solver*, which can solve constraint problems expressed in propositional calculus, and a *complete rational solver* for linear arithmetic terms, which uses Gaussian elimination and an incremental Simplex method to solve equations and inequalities over a continuous domain (Simonis, 1995).

```
money([S,E,N,D,M,O,R,Y]) :-
    ranges([S,E,N,D,M,O,R,Y]),
    state_constraint([S,E,N,D,M,O,R,Y]),
    labelling([S,E,N,D,M,O,R,Y]).

state_constraint([S,E,N,D,M,O,R,Y]) :-
    alldifferent([S,E,N,D,M,O,R,Y]),
    S ## 0,
    M ## 0,
                  1000*S + 100*E + 10*N + D
                + 1000*M + 100*O + 10*R + E
    #= 10000*M + 1000*O + 100*N + 10*E + Y.

ranges([]).
ranges([X|Xs]) :- X :: 0..9, ranges(Xs).

labelling([]).
labelling([H|T]) :-
    delete(V,[H|T],R,0,first_fail),
    indomain(V),
    labelling(R).
```

Figure 20: SEND + MORE = MONEY problem coded in CHIP

Figure 20 shows how the cryptarithmetic example can be coded using CHIP[12]. Here, the symbol ## is a constraint meaning “not equal to”. Note that the goal `delete(V, [H|T], R, 0, first_fail)` uses the **fail-first heuristic** for variable ordering. The idea behind this heuristic is to assign values to more constrained variables first, since an early commitment to highly constrained variables can significantly reduce backtracking during search. The program splits the list `[H|T]` into the most constrained variable (`V`) and the rest of the list (`R`).

12. reproduced from `http://www.aiai.ed.ac.uk/~timd/constraints/csptools/csptools.html`

3.3.2.2 ILOG Solver

The developers of the **ILOG Solver** recognised that embedding a constraint logic programming language into an object-oriented host language could provide advantages, such as greater software extensibility and improved modelling of the problem (Puget, 1994). Their first implementation, called **PECOS**, demonstrated the feasibility of the idea, and was written in an object-oriented dialect of LISP. The technical success of PECOS lead to the development of the current commercially available version of ILOG Solver, which is a constraint satisfaction library for C++.

The ILOG Solver maintains constraint variables as C++ objects which have several hidden slots. For example, an object which represents a logical variable contains slots for the value of the variable (if it is known), a representation of the domain of the variable (if the value is unknown), and a list of constraints posted on that variable. Constraints are themselves represented as objects, so the assertion of a constraint implies firstly, the creation of a new constraint object and, secondly, its addition to some variable's list of posted constraints.

The object-oriented approach to constraint handling also makes it possible to define C++ classes which already have constraints posted on them as part of their definition. Such *class constraints* will then be satisfied by all instances of the class. The representation of variables as objects also assists memory management through a module called the memory *allocator*. This unit relieves the programmer of many memory allocation considerations: it automatically reclaims the memory used by objects created in a branch of the search tree when that branch is abandoned.

Figure 21 illustrates the use of the ILOG Solver to find a solution to the SEND+MORE=MONEY cryptarithmetic problem[13]. The solution strategy is similar to the CHIP version, i.e.: domain variables are first declared, a single constraint is asserted to represent the sum, and then values are generated using the fail first heuristic.

13. also reproduced from `http://www.aiai.ed.ac.uk/~timd/constraints/csptools/csptools.html`

```
#include <fstream.h>
#include <ilsolver/ctint.h>

CtInt dummy = CtInit();

CtIntVar S(1, 9), E(0, 9), N(0, 9), D(0, 9),
         M(1, 9), O(0, 9), R(0, 9), Y(0, 9);
CtIntVar* AllVars[]={&S, &E, &N, &D, &M, &O, &R, &Y};

int main(int, char**) {
    CtAllNeq(8, AllVars);
    CtEq(          1000*S + 100*E + 10*N + D
                 + 1000*M + 100*O + 10*R + E,
         10000*M + 1000*O + 100*N + 10*E + Y);
    CtSolve(CtGenerate(8, AllVars));

    for (CtInt i = 0; i < 8; i++)
        AllVars[i]->setName("");
    cout << "        " << S << E << N << D << endl;
    cout << " +      " << M << O << R << E << endl;
    cout << " = " << M << O << N << E << Y << endl ;
    CtEnd();
    return 0;
}
```

Figure 21: SEND + MORE = MONEY problem coded in the C++ version of the ILOG SOLVER

3.3.2.3 IF/PROLOG

IF/PROLOG is a commercially available ISO-compliant PROLOG which includes an optional ‘Constraint Technology Package’. The package solves systems of constraints over boolean, linear rational, and finite domain expressions. It also includes additional facilities for dealing with big integers and precise rationals. (A ‘big’ integer is one with up to 300000 digits, and a ‘precise’ rational is an uncancelled fraction of two big integers.) Interestingly, the package also incorporates a mechanism called ‘co-routines’, which are implemented as constraints and provide the programmer with more control than usual over the order in which PROLOG goals are evaluated. PROLOG normally proves sub-goals procedurally from left to right, which may result in cases which, from the declarative point of view, are incorrect. Consider, for example, the following interaction:

```
[user] ?- A \= a, A = b.
```

```
no
```

Here, the user has tested the variable `A` for inequality with the atom `a` *before* instantiating it with the value `b`. Since, at that stage, `A` is uninstantiated and therefore *can* be unified with `a`, the inequality fails and the answer `no` is given. This is very different if the two sub-goals are given in the opposite order:

```
[user] ?- A = b, A \= a.
          A = b
yes
```

In this case, `A` is first given the value `b`, and *afterwards* tested for inequality with `a`. This succeeds and the answer `yes` is generated.

Using a co-routine called `freeze`, it is possible to delay the proof of the inequality until after the variable A has been instantiated:

```
[user] ?- freeze(A, A \= a), A = b.
         A = b
yes
```

It is argued[14] that this leads to a more declarative style of PROLOG programming.

As well as the constraint technology package, IF/PROLOG also includes options for an interactive development environment with a graphical user interface tool-kit (for building GUIs directly in PROLOG), and an SQL relational database interface.

A solution to the SEND + MORE = MONEY problem is included in Figure 22 as an illustrative example of the use of the IF/PROLOG constraint technology package[15].

3.3.2.4 ECLIPSE

ECLIPSE is a recent PROLOG-based package (derived from CHIP) which has been designed to support some techniques from mathematical and stochastic programming as well as those of constraint logic programming (Wallace, Novello and Schimpf, 1997). The idea is to enable these techniques to be combined easily to produce *hybrid algorithms* capable of tackling complex industrial problems. Moreover, ECLIPSE

14. in the corporate literature of IF Computer, GmbH; see `http://www.ifcomputer.de/Products/IFProlog/Constraints/Specifications/home_en.html`
15. reproduced from `http://www.ifcomputer.de/Products/IFProlog/Constraints/ExamplePrograms/SendMoreMoney/home_en.html`

```
:-  import(const_domain).

send([[S,E,N,D],  [M,O,R,E],  [M,O,N,E,Y]])  :-
              Digits  =  [S,E,N,D,M,O,R,Y],
              Carries  =  [C1,C2,C3,C4],
              Digits  in  0..9,
              Carries  in  0..1,

              M                  ?=                C4,
              O  +  10  *  C4  ?=  M  +  S  +  C3,
              N  +  10  *  C3  ?=  O  +  E  +  C2,
              E  +  10  *  C2  ?=  R  +  N  +  C1,
              Y  +  10  *  C1  ?=  E  +  D,

              M  ?>=  1,
              S  ?>=  1,
              all_distinct(Digits),
              label(Digits).
```

Figure 22: The SEND + MORE = MONEY Problem coded in IF/PROLOG

supports *experimentation* with different hybrid algorithms, thereby assisting in the search for an appropriate algorithm to tackle a given problem.

ECLIPSE offers several libraries which together provide functionality for handling symbolic and numeric constraints in Boolean logic, finite domains, integer-valued arithmetic, and real-valued interval arithmetic (in which the variables are assigned upper and lower bounds). Unlike most other packages, however, it also integrates *repair methods* for solving CSPs. Such methods are incomplete, in the sense that they are not guaranteed to find a solution if it exists, but they often outperform complete methods, particularly when the search space is large.

Like IF/PROLOG's *co-routines*, ECLIPSE also makes provision for improved data control in PROLOG with a similar, but more general, mechanism, called **suspension**. This facility causes a goal to wait until some event occurs, when it will automatically 'awaken'. Such events could be, for example, the instantiation of a variable, the tightening of the lower bound of a finite domain variable, or any other domain reduction of a variable. Suspended goals can be re-activated, or 'killed', which means that they can never be woken. Furthermore, suspended goals can also be retrieved by the program itself *at run time*, enabling constraint handling behaviour to be programmed explicitly.

Although this provides a high degree of control, I question whether programs that use this facility can be truly declarative.

C and C++ programs can benefit from the power of ECLIPSE by being linked with an ECLIPSE library. Facilities are provided for passing data between the two environments, so that a close coupling is achieved. It is also possible for ECLIPSE programs to invoke C and C++ code.

The SEND + MORE = MONEY puzzle provides an illustration of the use of finite domains in ECLIPSE:

```
:- use_module(library(fd)).

send(List) :-
  List = [S, E, N, D, M, O, R, Y],
  List :: 0..9,
  alldistinct(List),
  1000*S+100*E+10*N+D + 1000*M+100*O+10*R+E #=
    10000*M+1000*O+100*N+10*E+Y,
  M ## 0,
  labeling(List).
```

Figure 23: SEND + MORE = MONEY problem coded in ECLIPSE

The problem is expressed as follows: the `send` clause first introduces the variables of the problem and their domains, then states the conditions that must hold for the involved variables, and finally invokes the default labelling procedure. The labelling procedure assigns values to each of the constrained variables.

3.3.3 Progressive Systems

The progressive systems represent more recent research directions. SCREAMER is unusual because it is one of the few systems which has been implemented in Common LISP, is portable and also extensible. The CLAIRE/ECLAIR system is interesting because it has been implemented as a *preprocessor* to the underlying language (C++), rather than being written in C++ itself. An interesting feature of Oz is a tool for visualising the search for solutions as it proceeds.

3.3.3.1 SCREAMER

SCREAMER is a portable extension of Common LISP which provides for nondeterminism (Siskind and McAllester, 1994) and constraint logic programming (Siskind and McAllester, 1993). To add nondeterminism to LISP, a choice point operator has been introduced, together with a failure function. These two new constructs provide the basis for automatic backtracking. For constraint handling, SCREAMER employs a number of local propagation techniques including binding propagation, Boolean constraint propagation, generalised forward checking, propagation of bounds, and unification (see “How the Constraints Package of SCREAMER Works” on page 110). SCREAMER differs from many other constraint packages in the range of constraints that it can handle. For example, unlike other packages, non-linear sets of numeric constraints can be solved by combining local propagation with a primitive function called *divide-and-conquer-force*. This function nondeterministically reduces the set of possible values of a variable by splitting its domain into two equally sized subsets and constraining the variable to take a value from either the first or the second subset. SCREAMER also enjoys limited abilities for symbolic constraint solving through generalised forward checking and unification. Siskind and McAllester cite three main advantages of SCREAMER over CLP(*R*) and CHIP: firstly, SCREAMER is portable to any Common LISP implementation; secondly, it can be easily modified and extended; and, thirdly, it can coexist and inter-operate with other current or future extensions to Common LISP.

Figure 24 lists a SCREAMER function which solves the SEND + MORE = MONEY cryptarithmetic problem. Notice the convention that SCREAMER functions ending with the letter ‘v’ are constraint-based counterparts of conventional Common LISP functions.

SCREAMER is described in more detail in chapter 5.

3.3.3.2 CLAIRE / ECLAIR

CLAIRE (‘Combining Logical Assertions, Inheritance, Relations and Entities’) is an object-oriented language designed ‘to solve advanced industrial problems (that require

```
(defun money ()
 (let ((s (an-integer-betweenv 1 9))
       (e (an-integer-betweenv 0 9))
       (n (an-integer-betweenv 0 9))
       (d (an-integer-betweenv 0 9))
       (m (an-integer-betweenv 1 9))
       (o (an-integer-betweenv 0 9))
       (r (an-integer-betweenv 0 9))
       (y (an-integer-betweenv 0 9)))

   (assert! (/=v s e n d m o r y))
   (assert! (=v (+v (*v 1000 s) (*v 100 e) (*v 10 n) d
                    (*v 1000 m) (*v 100 o) (*v 10 r) e)
                (+v (*v 10000 m)
                    (*v 1000 o) (*v 100 n) (*v 10 e) y)))

   (format t "~?~%" "s=~d e=~d n=~d d=~d m=~d o=~d r=~d y=~d"
           (one-value (solution
                       (list s e n d m o r y)
                      (static-ordering #'linear-force))))))
```

Figure 24: SEND + MORE = MONEY coded in SCREAMER

some form of reasoning or constraint solving)' (Caseau and Laburthe, 1996). It was designed to embrace much of the problem-solving ability of an earlier language, **LAURE**™, but with the additional qualities of *simplicity* and *C++ compliance*. Furthermore, the language designers aimed at a style which would be both *concise* and *easy to read*. The language is implemented as a C++ preprocessor, so that each compiled CLAIRE object is a C++ object. This means that although CLAIRE is a language in its own right, it can easily be combined with C++ code.

ECLAIR ('Expressing Constraints with a Library of Algorithms for Inference and Resolution') is an extension of CLAIRE which provides for the solution of finite domain constraint satisfaction problems (Laburthe et al., 1998). It is a small library of CLAIRE objects and methods, limited to the solution of sets of finite domain numeric constraints. Once the constraints have been set up, the *fail-first heuristic* is employed during the search for solutions. The package is intended mainly for the purposes of prototyping and research, and is more limited than commercial tools. It is an interesting approach, however, because of its C++ compliance and the preprocessing of the source as a step which generates C++ objects.

```
S :: Var(inf = 1, sup = 9)
E :: Var(inf = 0, sup = 9)
N :: Var(inf = 0, sup = 9)
D :: Var(inf = 0, sup = 9)
M :: Var(inf = 1, sup = 9)
O :: Var(inf = 0, sup = 9)
R :: Var(inf = 0, sup = 9)
Y :: Var(inf = 0, sup = 9)

Letters :: set(S,E,N,D,M,O,R,Y)
Word1 :: Var(sup = 1000000)
Word2 :: Var(sup = 1000000)
Word3 :: Var(sup = 1000000)

// solution (S,E,N,D,M,O,R,Y) = (9,5,6,7,1,0,8,2)
[send()
-> AllDifferent(Letters),
   list(1000,100,10,1) scalar list(S,E,N,D) == Word1,
   list(1000,100,10,1) scalar list(M,O,R,E) == Word2,
   list(10000,1000,100,10,1) scalar list(M,O,N,E,Y) == Word3,
   list(1,1) scalar list(Word1,Word2) == Word3,
   list(1000,91,1,10) scalar list(S,E,D,R) ==
                (list(90,1,9000,900) scalar list(N,Y,M,O))]
```

Figure 25: SEND + MORE = MONEY problem coded in ECLAIR

Figure 25 illustrates the use of ECLAIR for the solution of the SEND + MORE = MONEY cryptarithmetic problem[16].

3.3.3.3 Oz

Oz[17] is a symbolic processing language with built-in facilities for concurrent programming and constraint-based problem solving. It was designed and developed at the DFKI (Deutsches Forschungszentrum für Künstliche Intelligenz, or German AI Research Centre), and was first released in January 1994. Later versions improved on its syntax and efficiency[18]. Oz has been used for various applications, including multi-agent systems, simulations, scheduling, configuration, frequency allocation for mobile telecommunications, the development of graphical user interfaces, natural language processing, automated musical composition, and the programming of virtual reality (Müller and Smolka, 1996).

16. an example program provided with the ECLAIR installation
17. See `http://ps-www.dfki.uni-sb.de/oz/` for more information about Oz
18. For detailed performance figures, see `http://www.ps.uni-sb.de/oz2/features/efficiency.html`

```
proc {Money Root}
   S E N D M O R Y
in
   Root = sol(s:S e:E n:N d:D m:M o:O r:R y:Y) %1
   Root ::: 0#9                                %2
   {FD.distinct Root}                          %3
   S \=: 0                                     %4
   M \=: 0
                1000*S + 100*E + 10*N + D      %5
   +            1000*M + 100*O + 10*R + E
   =: 10000*M + 1000*O + 100*N + 10*E + Y
   {FD.distribute ff Root}
end
```

Figure 26: SEND + MORE = MONEY problem coded in OZ

Figure 26 contains an Oz script which solves the SEND + MORE = MONEY cryptarithmetic problem[19]. In this script, digits of the problem are represented by their corresponding letter. *Root* is a record which has a field for every letter (1), so, for example, the field *s* represents the value of the digit *S* in the problem. All the fields of Root are integers in the range 0..9 (2), and they are pairwise distinct (3). Since S and M are the leading digits of the three numbers in the sum, they cannot be zero (4). Lastly, the digits for the letters satisfy the equation SEND+MORE=MONEY (5). The line `{FD.distribute ff Root}` posts a so-called *distributor*, which orders the selection of fields within the *Root* record according to the fail-first (`ff`) strategy.

A tool for visualising search spaces explored by Oz, called the *Explorer*, is also available. It uses the search tree of a constraint problem as its main metaphor, which it enables a user to interactively explore. It is claimed that ‘first insights into the structure of the problem can be gained from the visualisation of the search tree: how are solutions distributed, how many solutions exist, how large are the parts of the tree explored before finding a solution’. The interactive nature of the user-interface supports the design of new heuristics and search strategies. In order to cope with different sizes of search spaces, the visualisation can be scaled by using a scrollbar, and branches of the

19. Reproduced from Schulte, C., Smolka, G., Würtz, J., (1998), “Finite Domain Constraint Programming in Oz: A Tutorial”, Oz Documentation Series, on-line at `http://www.ps.uni-sb.de/oz2/documentation/`

tree can be hidden. An example screenshot of the Explorer, taken when solving the cryptarithmetic problem, is given in Figure 27.

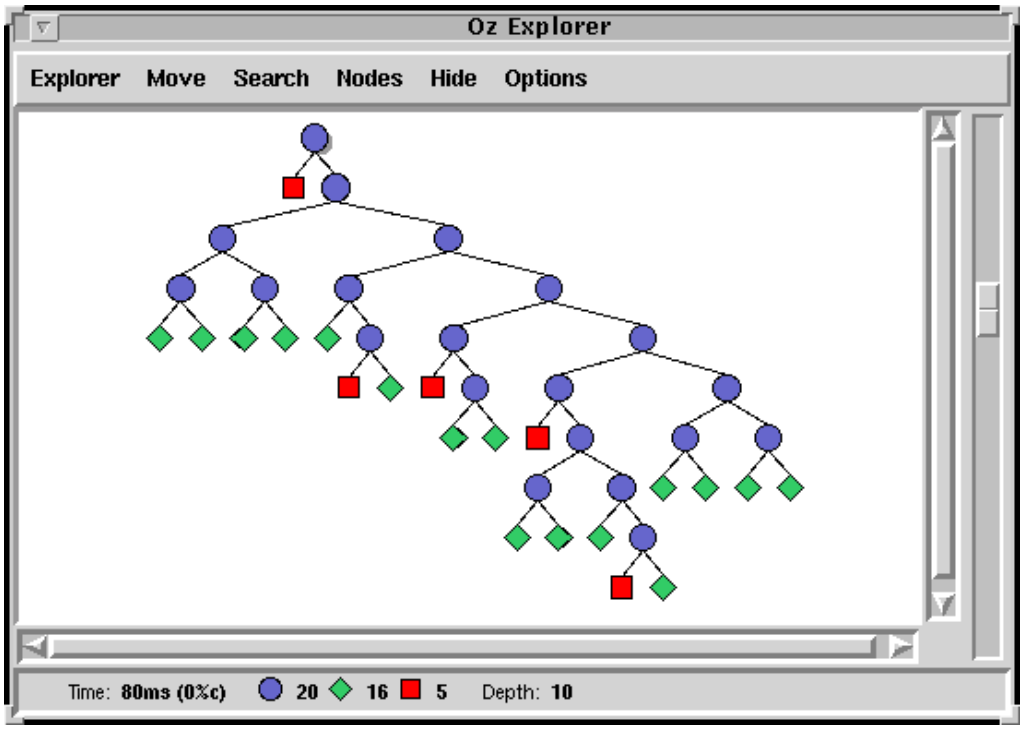


Figure 27: A Screenshot of the Oz Explorer searching for solutions to the cryptarithmetic problem

3.3.4 Other Systems

There are now so many systems for solving sets of constraints that I cannot describe them all in detail, nor even mention them all. Nevertheless, I mention here briefly some other interesting languages and systems. **SICStus PROLOG**[20], for example, is a commercially available UNIX PROLOG which includes facilities for finite domain constraint handling. The **Gödel** programming language[21], developed at the University of Bristol, claims to be more declarative than PROLOG, but also contains solvers for constraints over finite domains and linear rational arithmetic. Finally, **JCL** (Java Constraint Language)[22], developed at the Artificial Intelligence Laboratory of the

20. See `http://www.sics.se/sicstus.html`
21. See `http://www.cs.bris.ac.uk/~bowers/goedel.html`
22. See `http://liawww.epfl.ch/~torrens/Project/JCL/`

Swiss Federal Institute of Technology in Lausanne, Switzerland, provides application services related to binary constraints. This package is interesting because it enables constraint satisfaction problems to be solved by a Java *applet*, which runs on the client side of a network connection. You can find out more about these and other systems/ languages from the *constraints archive* on the World Wide Web[23].

I conclude this chapter by summarising the constraint systems discussed in the form of a table:

Name	Language(s)	Availability	Features
Steele's Constraint System	LISP	None.	Basic constraint propagation. Historical interest only
CLP(*R*)	Superset of PROLOG	free for academic and research purposes	Contains solver for sets of constraints in linear real-valued arithmetic
CHIP	PROLOG, C, and C++	commercial	Provides solvers for sets of constraints in Boolean- and finite domains, and linear rational arithmetic.
ILOG SOLVER	C++	commercial	Provides solvers for finite domain and linear arithmetic constraints, as well as specialised solvers for assignment and scheduling problems. Variable ordering strategies are user-definable. Object-orientation allows abstraction and reuse of constraint features.
IF/PROLOG	PROLOG	commercial	Solves systems of constraints over boolean, linear rational, and finite domain expressions. Provides co-routines for improved control over PROLOG goals, and can deal with big integers and precise rationals.
ECLIPSE	PROLOG	free to academic institutions; commercially available for other research	Provides many different solvers and enables the programmer to generate hybrid algorithms more easily.
SCREAMER	Common LISP	non-commercial	Adds non-determinism to Common LISP and provides for solving constraints over finite domain and bounded interval reals. Some provision for non-linear arithmetic and symbolic constraints. Portable and extensible.
ECLAIR	CLAIRE	non-commercial	Can solve finite domain CSPs in a specialised language that is 'C++ compliant'.
OZ	OZ	non-commercial	Specialised language for concurrent and constraint-based problem solving. The language combines ideas from logic-, functional-, and object-oriented programming. Includes a visualisation tool, the *Explorer.*

23. See `http://www.cirl.uoregon.edu/constraints/`

Chapter 4

MUSKRAT and the Problem of Fitness for Purpose

'Ramble On ... And now's the time, the time is now, to sing my song.'
Robert Plant, from the album Led Zeppelin II

Chapter Summary

This chapter introduces the core topic of my research, that of *fitness for purpose* applied to knowledge acquisition and problem solving. I describe my motivation for investigating this topic, and detail the relevant historical background. I then describe two specific problems related to fitness for purpose; namely that of determining whether available knowledge is fit for some given (problem solving) purpose, and acquiring new knowledge such that it is fit for some purpose. I explain why these problems are difficult, and sketch my solutions to these problems, which use constraint technology. The detail of the solutions is covered in later chapters.

4.1 Motivation and Context

As we have already seen in chapter 2, many knowledge acquisition tools and techniques have been developed in recent years. These include *knowledge elicitation methods* which acquire knowledge from a human expert, *machine learning algorithms* which infer knowledge from data, and *knowledge base refinement tools* which improve existing knowledge bases. Unfortunately, the increasing sophistication of these techniques and the large number of available tools makes it very difficult for many users (notably domain experts) to choose an appropriate tool for their particular application.

One approach to this problem is to include several different tools in an integrated toolbox, and provide an advisory system which comments on the suitability of the available tools for a particular task. This was the approach taken by the **MLT** (Machine Learning Toolbox) project, because

> '... it is not realistic to expect a user to find his way in the maze of the different algorithms', (Kodratoff et al., 1992).

An advisor of this kind was also foreseen as part of the specification of MUSKRAT (Graner & Sleeman, 1993). MUSKRAT (a Multistrategy Knowledge Refinement and Acquisition Toolbox) was conceived in the wake of the MLT project, adopting the alternative view that knowledge should not be acquired in isolation from the problem solving to which it contributes. For this reason, the MUSKRAT architecture included problem solvers as well as knowledge acquisition tools. The architecture should also be "open", in the sense that new tools can be added easily. Note that the *toolbox* itself places no restrictions on the tasks which it should solve; limitations in the range of tasks which can be solved emerge as a result of the tools and knowledge bases which it contains. The strength of MUSKRAT lies in the belief that descriptions of KA tools and problem solvers can be used to guide knowledge acquisition towards the effective use of a chosen problem solver. This style of KA is called **goal-driven knowledge acquisition** (Sleeman & White, 1997).

Although the MUSKRAT system is interesting in its own right, it is not the main subject of this thesis. Rather, MUSKRAT is the *vehicle* through which the deeper issues of *fitness for purpose* have been investigated. This distinction was already clear to one of the reviewers of my thesis proposal[1]:

> 'In my mind, the primary contribution of the student's dissertation will not be MUSKRAT, but rather this theory of KA-tool competency.' (Musen, 1998)

Although my initial concern was mostly for describing the fitness for purpose of *KA tools*, my subsequent research emphasised the investigation of *problem solvers*. This approach was consistent with my conjecture that:

1. The relationship between *fitness for purpose* and *competency* is discussed in detail on page 91.

- describing the fitness for purpose of KA tools could be harder than the same task for problem solvers, because problem solvers typically *search* and/or *transform* knowledge, but do not add *new* knowledge;
- it may be possible to show that methods used to describe the fitness for purpose of problem solvers can also be used to describe the fitness for purpose of KA tools.

The philosophy of MUSKRAT is to support the use/reuse of *existing* tools without changing them. Fitness for purpose descriptions were therefore originally conceived as value-adding supplements to the tools, rather than modifications of the tools themselves. As a side-effect of my main investigations, however, I was later able to show that the notion of fitness for purpose can also be *incorporated into* a knowledge acquisition tool, so that the knowledge acquired by the tool can be used directly, without further transformation. I call this “acquiring knowledge which is fit for a purpose”, a topic discussed later in this dissertation.

This work should be seen within the context of knowledge acquisition research. Not only does my work concern the description of *fitness for purpose*, which is an important issue in Knowledge Acquisition, it also contributes to the topic of *knowledge reuse*.

4.2 The Past: The Machine Learning Toolbox

The origins of my work can be traced back to the **MLT** (Machine Learning Toolbox) project (Kodratoff et al., 1992). This section recalls the aims of that project, describes the approach taken, and notes some limitations of the adopted approach. Observation of these limitations led to the specification of MUSKRAT (Graner & Sleeman, 1993).

4.2.1 The Learning Tools of MLT

The aim of the MLT project was to facilitate the application of machine learning to real industrial problems by building a collection of machine learning tools together with an advice-giving system. I note with interest that, in many cases, the inclusion of the tools within the toolbox also led to their improvement. Algorithms originally based on nu-

meric techniques were extended to include symbolic abilities (and vice versa), and the user-interfaces of all algorithms were improved during the project (Kodratoff, 1992). The toolbox contained ten learning tools, whose characteristics are summarised in Table 4 on page 80.

Note that most of the tools were algorithms for supervised learning, with the exceptions of the unsupervised learning methods of **DMP** and **SICLA**. Please refer to (Kodratoff et al., 1992) or the MLT Project Fact Sheets for more detailed information about the algorithms.

4.2.2 The Common Knowledge Representation Language

CKRL (the Common Knowledge Representation Language) was the knowledge interchange language of MLT. It can be seen as a predecessor to **KIF** (Genesereth & Fikes, 1992) and **XML**[2]. Unlike KIF and XML, however, it was initially developed with only one purpose in mind – that of enabling the exchange of information within the machine learning context of MLT. That said, the constructs it defines are largely of a generic nature: it provides for *concepts*, *properties*, *instances*, *sorts*, *relations*, *facts*, and *rules*.

Since I also used CKRL for some of the work on MUSKRAT, I now give a brief overview of its constructs. However, it contains many more features than I can explain in this short introduction; please refer to (Morik, Causse & Boswell, 1991; MLT Consortium, 1992a; MLT Consortium 1992b; MLT Consortium 1993) for more complete accounts.

2. See `http://www.w3.org/XML/`

Name & Origin	Input(s)	Output(s)	Comments
APT [ISoft SA, University of Coimbra, LRI.]	Incomplete and/or incorrect domain theory and problem solving rules	General problem decomposition rules	Detects and interactively corrects errors in the problem decomposition rules used as input. Then generalises these rules, again through interaction with an expert.
CIGOL [Turing Institute]	+ve and -ve examples, fact-based background knowledge	Rules (Horn clauses)	First-order rule induction
CN2 [Turing Institute]	Vectors of attribute values, which can be symbolic, integer or real. Missing values are permitted.	Ordered list of rules (a nested 'if-else'), or a set of rules.	Clark & Niblett's CN2 algorithm is a member of the AQ family of induction algorithms.
DMP [British Aerospace, plc.]	A directed graph, represented as a set of tuples.	Clusters of nodes within the graph.	A "Database Monitor Program", devised to cluster large quantities of noisy, low quality symbolic data.
KBG [Laboratoire de Recherche en Informatique (LRI), ISoft.]	A set of examples and a domain theory, expressed in first-order logic.	A graph of concepts clustering the examples and a set of rules to predict the values of predicates.	Clusters and generalises by applying a similarity measure to the set of examples
LASH [British Aerospace, plc.]	Prolog facts expressing either attribute-value pairs, or binary relationships.	Rules for describing the concept.	Supervised learning of rules from attribute-value pairs and relational information.
MAKEY [Institut National de la Recherche en Informatique et Automatique (INRIA)]	A set of concept descriptions, plus background knowledge	An "identification graph", or a set of rules	Induces discrimination rules of concepts from concept descriptions. Uses "typicality modifiers" which estimate the relative frequency of each value.
MOBAL [German National Research Centre for Computer Science (GMD)]	Model of an application domain; i.e. at least a set of facts describing domain structures or objects.	Inferred facts, taxonomy, Horn-clause rules.	Uses a first-order logic representation to model and revise concepts and rules.
NewID [Turing Institute]	Vectors of attribute-values	A decision tree	An implementation of Quinlan's ID3 algorithm. NewID can also evaluate a tree's accuracy.
SICLA [INRIA]	Vectors of attribute-values, and a 'distances matrix'.	A set of clusters	A set of algorithms that perform statistical symbolic and numerical data analysis. SICLA actually consisted of two main subsystems, SICLA-Clustering and SICLA-Disc.

Table 4: The Machine Learning Algorithms of MLT

4.2.2.1 Concepts, Properties and Instances

Concepts, **properties**, and **instances** together form a frame-based[3] knowledge representation language. A concept is defined using the `defconcept` construct, as in the following definition of a `dish`, which contains slots (properties) for its components (`main-component`, `carbohydrate-component` and `vegetable-component`), its `cost`, and the number of `dietary-points` associated with it.

```
defconcept dish
  relevant (main,
            carbohydrate,
            vegetable,
            cost,
            dietary-points);
```

The properties themselves are defined using the `defproperty` construct, which indicates the sort of information which the property contains. For example, the properties of a `dish` are given as:

```
defproperty main conceptref main-component;
defproperty carbohydrate conceptref carbohydrate-component;
defproperty vegetable conceptref vegetable-component;
defproperty cost sortref posint;
defproperty dietary-points sortref posint;
```

This says that the property `main` must be filled by a value which is an instance of the concept `main-component`. The properties `carbohydrate` and `vegetable` are similarly specified. The `cost` of a dish and the number of `dietary-points` it contains are both given by a value which is a positive integer (`posint`). Concepts can also be structured into a hierarchy, in which subconcepts inherit properties from their superconcepts. For example, the following CKRL specifies the hierarchy given in figure 28.

```
defconcept dish-component
  relevant (dietary-points,
            cost,
            preparation-time);

defconcept main-component superconcept dish-component;
defconcept meat-component superconcept main-component;
defconcept vegetarian-component superconcept main-component;
defconcept fish-component superconcept main-component;
```

3. Frames are data structures for representing stereotyped situations, but were also the main inspiration for modern *object-oriented* languages. They are an important tool of Artificial Intelligence and enable the knowledge engineer to define *classes*, *slots* (attributes), and their *fillers* (values). Classes may also inherit slots (and values) from other classes.

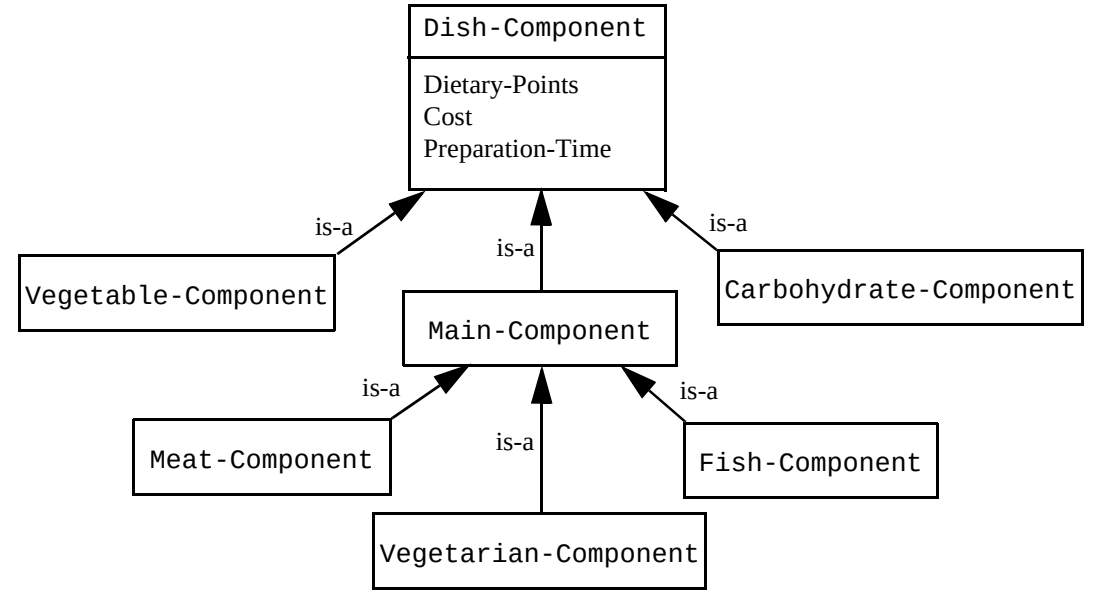


Figure 28: A Hierarchy of Dish Components

```
defconcept vegetable-component superconcept dish-component;
defconcept carbohydrate-component superconcept dish-component;
```

Instances of a concept are given using `definstance`. For example, the following defines `favourite` as an instance of the concept `dish`:

```
definstance favourite conceptref dish
    (main GAMMON-STEAK,
     carbohydrate FRENCH-FRIES,
     vegetable PEAS,
     cost 6,
     dietary-points 18);
```

4.2.2.2 Sorts

A **sort** behaves rather like a *type* in a conventional programming language. As you might expect, some common sorts are predefined by CKRL, such as `boolean` and `integer`. Other sorts can be defined within the language by using the `defsort` construct. For example, the sort `posint`, used above for the property `cost`, is defined as the set of non-negative integers in the following way:

```
defsort posint (range (integer (0:*)));
```

Sorts can also define ranges of nominal values:

```
defsort eating-habit
  range (nominal (vegan, vegetarian, omnivore));
```

4.2.2.3 Relations and Facts

Relational knowledge can be expressed by using a combination of **relations** and **facts**. The *relation* declares the concepts, or sorts, of the items involved in the relation, and *facts* instantiate those relations with concrete values. Relations are defined using the `defrelation` construct, and facts given using `deffacts`. For example, the following fragment defines the `uses` relation as a 3-tuple of a `task` concept, a `resource-type` concept, and a `posint`, which is a non-negative integer.

```
defrelation uses conceptref task,
                conceptref resource-type,
                sortref posint;
```

The relation can then be used to specify the number and type of resources which each task uses. The following `deffacts` statement provides three facts (tuples) which satisfy the `uses` relation. Note that `grill`, `friteuse`, and `saucepan` must be defined instances of the `resource-type` concept; and `grill-gammon`, `prepare-french-fries`, and `prepare-peas` must be instances of the `task` concept.

```
deffacts uses (grill-gammon, grill, 1),
         uses (prepare-french-fries, friteuse, 1),
         uses (prepare-peas, saucepan, 1);
```

4.2.2.4 Rules

Rules express simple inferencing capabilities that can be exchanged across implementation language barriers. Rules can be used to express the default behaviour of slots, or to assert further facts. For example, the following rule asserts a preference for dish components which can be prepared quickly (provided that the relation `like-component` is appropriately defined):

```
defrule like-quick
when (conceptref dish-component ?x)
  if (?x.preparation-time < 15)
  then (like-component(?x));
```

4.2.3 The MLT Consultant

This advisor, called the **Consultant**, was developed at the University of Aberdeen (Craw et al., 1992; Sleeman et al., 1995), and commented on the applicability of each tool for a particular learning task. In order to do this, it questioned its user about the task to be solved, and the data and background knowledge that could be provided. Three different modes of user-interaction were provided for this task description. In *fixed path mode*, questions relating to the same topic are asked together, thus maximising the apparent coherence to the user; in *intelligent mode*, the most discriminating question is asked at each stage, minimising the number of questions to be asked (at the risk of appearing incoherent); and in *browsing mode*, the user may choose the order in which to answer questions.

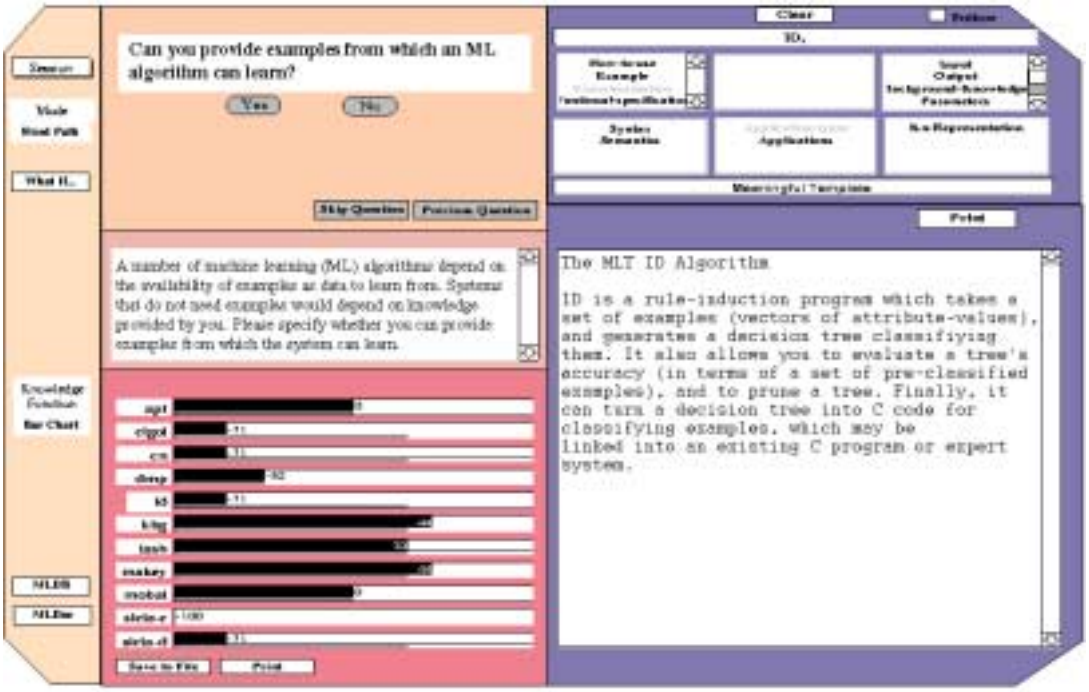


Figure 29: A Screenshot of the MLT Consultant

Figure 29 is a screenshot of the Consultant. The top left corner contains the interaction needed for the task description; the right side of the screen contains additional help information, and the bottom left corner maintains a bar chart of the certainty factors of

each of the toolbox algorithms. The bar chart is updated each time that additional task knowledge is acquired, so that users may begin to understand which answers are critical for the application of particular algorithms.

My investigation of the rules contained in MLT's knowledge base revealed that the 'most critical' rules for any algorithm were of an eliminative nature. That is, they could be roughly interpreted as saying

```
IF property_P IS_TRUE_OF learning_task_L
THEN don't_use algorithm_A.
```

The following small rule base is the pseudo-code of the "strong rules" contained in the Consultant-1 knowledge base of MLT. I define a "strong rule" to be one which has a relative effect on the certainty factor of an algorithm's applicability of ≥ 0.5. It was found that all such rules are of an eliminative nature[4].

```
IF user is not prepared to give a specific solution to problem
THEN don't use APT

IF the classes are not mutually exclusive (i.e. overlapping)
OR it is not possible to classify examples based only on the
provided attribute values
THEN don't use CN2

IF an output which consists of a generalisation, object match-
ings, intervals of values and links between values and objects
is not acceptable
THEN don't use KBG

IF each example is not described by a collection of predicates
OR the user cannot provide negative examples
OR user wants to select the most appropriate among a set of de-
cisions
OR the user's task is described by relations AND the number of
arguments > 2
THEN don't use LASH

IF user's concepts are organised as a hierarchy
OR the user has numeric data AND cannot specify numeric inter-
vals into which the values should be divided
OR user's input is not expressed as a set of concepts
OR the classes are not mutually exclusive (i.e. overlapping)
THEN don't use MAKEY

IF user has a lot of background knowledge
AND this knowledge can not be expressed as predicates
THEN don't use MOBAL
```

4. It is interesting that there are no rules for eliminating CIGOL or DMP, suggesting that either these algorithms have wide applicability, or there are some missing strong rules in the Consultant's knowledge base.

```
IF the classes are not mutually exclusive (i.e. overlapping)
OR it is not possible to classify examples based only on the
provided attribute values
THEN don't use NewID

IF the classes are not mutually exclusive (i.e. overlapping)
OR it is not possible to classify examples based only on the
provided attribute values
OR user is trying to recognise classes by the relation of sym-
bolic to numeric attributes
THEN don't use SICLA_DISC

IF there are missing or multiple values for numeric attributes
OR decisions don't depend on simple combinations of attributes
AND user's data is not a set of proximity/similarity measures
between entities
OR it is not useful to identify classes of similar entities
THEN don't use SICLA_CLUSTERING
```

Although it performed satisfactorily, the Consultant suffered from two major limitations. Firstly, since its recommendations were the result of *comparative* knowledge of the toolbox algorithms, it was difficult to extend with knowledge of other algorithms. This meant that whenever a new algorithm was added to the toolbox, the *whole* of the Consultant's knowledge base had to be revised, instead of just the knowledge pertaining to the new algorithm. Secondly, the Consultant had no understanding of the problem that the user wanted to solve in his application domain. Users were left to decide what knowledge was required to solve their problem by first defining a learning task; only afterwards could the Consultant help them with the choice of a suitable tool.

4.3 The Future: The MUSKRAT Vision

MUSKRAT was first described in 1993 (Graner & Sleeman, 1993), and conceived largely to overcome the observed limitations of MLT, and, in particular, the Consultant. It was unfortunate that, due to the personal circumstances of the student involved, these early ideas did not come to fruition sooner. My achievement has been to revive those ideas by understanding the issues involved, identifying a particular area of difficulty (that of 'fitness for purpose'), and developing techniques for dealing with it.

4.3.1 Critiquing the MLT Consultant

One of the criticisms levelled at MLT was that many of the algorithms it included were rather similar. The potential of the toolbox for real-world applications could be enhanced by widening its scope to include knowledge acquisition tools other than machine learning algorithms. Therefore the specification of MUSKRAT also included knowledge elicitation and knowledge-base refinement tools.

Perhaps the most important observation about the use of the Consultant is that recommendations on the success of learning algorithms are made *without reference to any problem solver which might use the learned knowledge*. Knowledge of a target problem solver would be very useful, since the mutual compatibility of the output of an arbitrary acquisition tool and the input requirements of a problem solver cannot be guaranteed. It was therefore decided that MUSKRAT should also have access to problem solvers, which will serve as the *targets* of the knowledge acquisition process. This is in contrast, for instance, with **KEW**, the Knowledge Engineering Workbench produced by the **ACKnowledge** project (Reichgelt and Shadbolt, 1992). KEW, which focuses on knowledge elicitation techniques, does not include problem solvers, and uses **Generalised Directive Models** (GDMs) to guide tool selection (van Heijst et al., 1992; Motta, O'Hara & Shadbolt, 1994). GDMs provide *knowledge engineers* with support when *building* knowledge-based systems. This differs from MUSKRAT, which aims to support the *end-user* in choosing among a set of *existing* tools and knowledge bases. The *generalised directive model* is encoded as a set of grammar rules. The knowledge engineer expands the grammar by making decisions about the nature of the task at hand. A full expansion (or interpretation) of the grammar provides a decomposition of the task to a level for which the associations between the task components and the knowledge elicitation tools are known. Appropriate knowledge elicitation tools can then be invoked to acquire the knowledge necessary for the task at hand.

4.3.2 The Scope of MUSKRAT

The MUSKRAT philosophy assumes that problem solving proceeds as follows:

Task Identification - A *task* is first identified; that is, a problem to be solved in a par-

ticular domain.

Problem Solver Selection - A suitable problem solver is selected to solve this task. If no single problem solver can be identified, it may be necessary to split the application task into sub-tasks that can each be solved by a problem solver.

Knowledge Specification - For each selected problem solver, the required data sets and knowledge bases are determined. This presupposes a knowledge-level analysis of each problem solver, which need only be done once since it does not depend on a particular application task.

Knowledge Gap Measurement - The available knowledge sources (human expert, examples, knowledge bases, etc.) are compared with the requirements of each selected problem solver. This may define one or more KA tasks.

KA Tool Selection - A tool is selected to solve each KA task; i.e., to bridge the gap between required and available knowledge. This presupposes a knowledge-level analysis of the available tools.

Knowledge Acquisition - The selected KA tools are applied.

Problem Solving - The selected problem solver(s) are applied.

These steps can be repeated in a cycle, especially if knowledge acquisition leads to the refinement of problem solver selection, or the knowledge acquisition step fails, or the user rejects the solution proposed in the final step.

MUSKRAT has been designed to support the steps of *knowledge specification*, *knowledge gap measurement*, and *KA tool selection*. It *assumes* that a problem solver has been selected for a particular task or sub-task, and directs the acquisition of knowledge for that problem solver. The system consists of any number of problem solvers, any number of KA tools, and a guidance module, the MUSKRAT advisor.

4.3.3 The MUSKRAT Advisor

The tool selection process starts with a description of the task and the subsequent selection of a corresponding problem solver. An advisory system may assist the user with this choice. This is not currently part of MUSKRAT, but a module similar to KEW's advice and guidance module (van Heijst et al., 1992) should be applicable.

Once a problem solver has been selected, MUSKRAT knows which KB(s) are *required*. This follows because each problem solver specifies the knowledge it requires in terms of its functionalities and representation. These requirements are expressed in a formalism which provides descriptors for both knowledge-level and symbol-level features.

The next step is to identify the available knowledge sources. I consider three broad categories of knowledge sources: **available knowledge** refers to knowledge that is already in the form required for a knowledge base, e.g., a set of rules. It may be directly usable or require transformation or refinement. Note that knowledge bases are seldom available initially, but when MUSKRAT is used iteratively as part of a problem solving cycle, *available knowledge* refers to that acquired during a previous iteration. **Available data** refers to a data set that is relevant to the problem and from which useful information could be extracted, even if it does not meet the immediate requirements of the KB. Typically, this may consist of past cases, i.e., previously solved problems similar to the one at hand, from which insights into the new problem can be gained. Alternatively, if the problem is to diagnose faults in a complex system, *available data* may refer to a model of the system which is useful (or perhaps necessary) to perform diagnosis. Note that the distinction between knowledge and data is not intrinsic, but depends on the problem solver's input requirements. For instance, a set of past cases is regarded as knowledge if it is to be used by a case-based reasoner that can use it directly, but only as data for a rule-based system which is unable to reason with cases. Finally, an **expert** is a person who can provide various forms of knowledge, possibly with the help of a knowledge elicitation tool and/or a knowledge engineer.

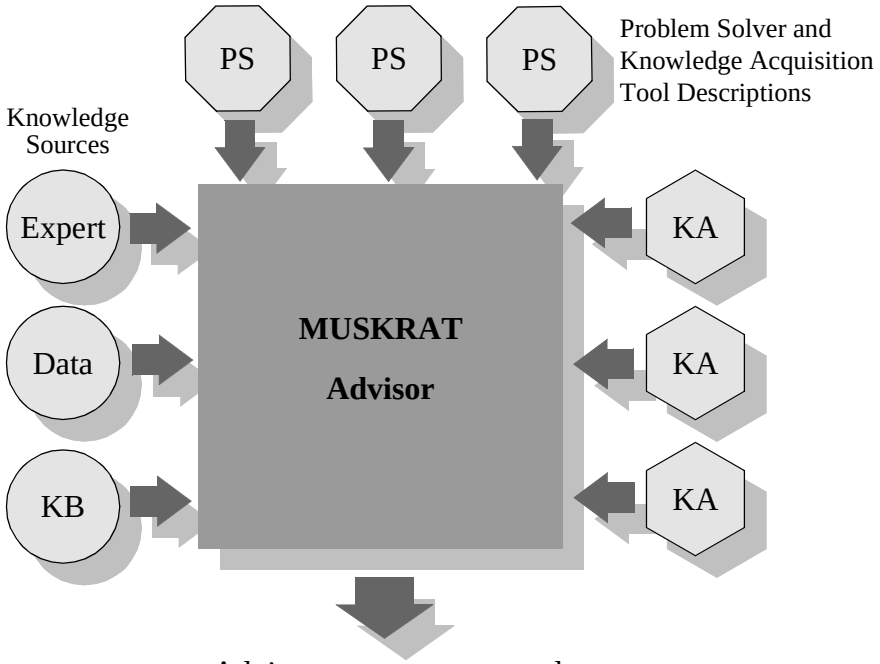


Figure 30: The Inputs and Outputs of the KA Selector/Advisor

The MUSKRAT Advisor is the central component of MUSKRAT (see Figure 30). Its most important function is to recognise reuse potential – a labour saving facility for the user. It does this at two levels; it endorses the *reuse of tools for achieving the user's problem solving goals* and the *reuse of knowledge bases in previously unseen problem solving contexts*. By recognising when reuse is appropriate, and generating advice for novice users, the toolbox becomes a more effective means for solving problems, and more amenable to those unfamiliar with the technology it contains.

In the next section, I outline my adopted approach to the mechanics of the advisor, which uses the technology of constraints. I also describe how this approach led to the development of a generic knowledge acquisition tool based on the same technology. The details of both the technology and the solutions are covered in later chapters.

4.4 The Present: Addressing Fitness for Purpose

I have already stated that the MUSKRAT advisor should compare the *requirements* of a selected problem solver with the *characteristics* of the available knowledge sources. The question of how this can be done is not only the subject of this section, but also the underlying concern of the rest of the dissertation. Informally, let us say that if there is a good match between a problem solver's input requirements for achieving a specific goal and the characteristics of knowledge bases that are to be used as inputs, then those knowledge bases are *fit for the purpose* of satisfying the goal with that problem solver. Using this new terminology, I can now state the requirement of the MUSKRAT advisor to check the fitness for purpose of available knowledge bases with respect to a chosen problem solver and goal.

Fitness for purpose is related to the notion of *competence* (e.g., Wielinga, Akkermans & Schreiber, 1998), but with some important differences. Firstly, whilst *competence* describes the general input–output relationship of a problem solver, the verification of fitness for purpose involves a test of a problem solver and a specific set of input instances against a given goal. Secondly, fitness for purpose is a *descriptive* notion, because it is applied to an existing system, whereas competence is a *prescriptive* term, applied to a system that is to be built. Lastly, unlike fitness for purpose, competence has not yet been realised computationally.

Recent work on the description of competence includes that of Benjamins et al. in the context of the **IBROW3** project (Benjamins et al., 1998; Benjamins et al., 1999), Fensel et al., as part of an ongoing investigation into the role of **assumptions** (Fensel & Schönegge, 1997; Fensel & Schönegge, 1998), and Wielinga et al., as a methodological approach to the operationalisation of knowledge-level specifications (Wielinga, Akkermans & Schreiber, 1998). Fensel's approach is probably the nearest to this work, because it investigates the *context dependency* of an existing problem solver/PSM through the discovery of assumptions. My work has also addressed the context dependency of problem solvers, but through the question of task suitability with the *available* knowledge. Thus, whilst Fensel investigates a problem solving method in isolation of its inputs in order to *derive* suitability conditions, I take a problem solver together with

its inputs and *test* their combined suitability for solving a specific task. Both lines of inquiry are intractable, and therefore demand some compromise(s) in any implementation. Fensel's compromise concerns the level of automation of his proof mechanism, which is an *interactive* verifier, rather than a fully automated proof mechanism. Since I would like to generate advice at *run-time* for the potential user of a problem solver, I compromise instead with the deductive power of my proof mechanism. I do not attempt to prove what the output of a problem solver will be, given its inputs; rather, I use constraint satisfaction techniques to prove what it is *not*. I believe that the strengths of the constraint satisfaction approach lie in its abilities to combine the results of multiple problem solvers (through propagation mechanisms), and run as a batch process. Figure 31 compares Fensel et al.'s approach to the discovery of a problem solver's properties with my approach to matching a problem solver to a toolkit user's goal.

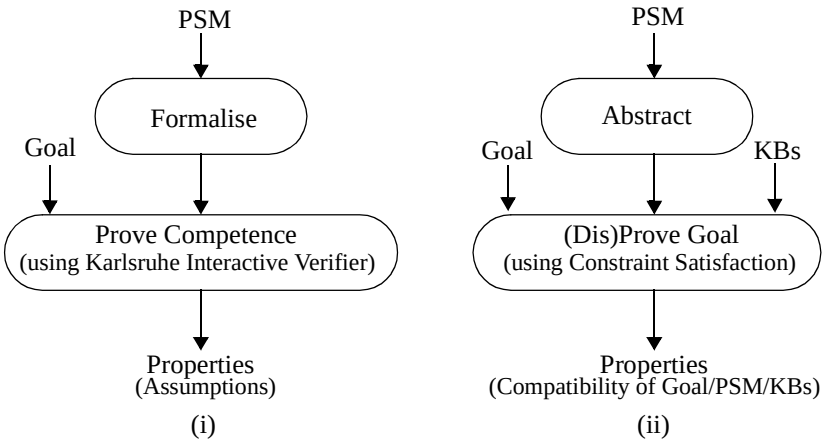


Figure 31: Comparison of the Fensel approach (i), and my approach (ii), to the discovery of problem solving properties

4.4.1 Three Actions Concerning Fitness for Purpose

My investigation of *fitness for purpose* was driven by the following three functional requirements, expressed in terms of the actions which MUSKRAT should be able to perform. The MUSKRAT Advisor should be able to:

★ **Determine** whether the knowledge required for solving a given problem is available, and in a form suitable for immediate use; [Action 1]

If the required is not available and applicable, then either

★ **Acquire** knowledge *ab initio* such that it meets the requirements of the problem solver; [Action 2]

or

★ **Modify** existing knowledge to meet the requirements of the problem solver. [Action 3]

The notion of *fitness-for-purpose* and the computational approximation to it (described later), were driven by the requirements of Action 1. I developed an approach to this problem using constraint technology, and subsequently recognised how the same technology could be used to build a separate knowledge acquisition tool that performs Action 2. In the following section, I explain the computational difficulties which arise in the determination of *fitness for purpose,* and sketch my solutions to the requirements of Actions 1 and 2.

4.4.2 Determining Fitness for Purpose

When the advisory subsystem of MUSKRAT addresses the problem of *fitness-for-purpose*, it is in effect posing the question “Is it possible to solve the given task instance with the available problem solver[5], knowledge bases, and KA tools?”. Clearly, this is a very difficult question to answer without actually running the problem solver, regardless of whether the answer is generated by a human or a machine. Indeed, the theory of computation has shown that it not possible, in general, to determine whether a pro-

5. For simplicity, I currently assume the application of a single, chosen problem solver.

gram (in this case, a problem solver) terminates. This is the **Halting Problem** (Turing, 1937). Therefore the most dependable way to affirm the suitability of a problem solver for solving a particular task is to run it and test the outcome against the goal. For the purposes of generating advice in a multistrategy toolbox, however, I cannot afford this luxury, particularly since running the problem solver could itself be computation-intensive. This presents a fundamental difficulty which can be addressed through the introduction of an appropriate *approximation* to fitness for purpose.

My approach exhibits an approximate, but more tractable, form of fitness for purpose, which I call **plausibility**. If the inputs to a problem solver can be demonstrated not to fulfil a specific goal, then the advisor tells the user not to run the problem solver. If, on the other hand, it cannot be demonstrated that the problem solver will not fulfil the goal with the given inputs, then the goal remains plausible. I am in effect posing the weaker question "Is it *plausible* that the given goal could be satisfied by the available problem solver, knowledge bases, and KA tools?". I argue that it is possible to demonstrate computationally that some configurations of a task, problem solver, existing knowledge bases and KA tools cannot, even *in principle*, generate an acceptable solution to the task. Such situations form a set of *recognisably implausible* configurations with respect to the problem at hand. Furthermore, the computational cost associated with recognising this implausibility is, in many cases, far less than that associated with running the problem solver.

Problem Inspection — Is the Problem 'Reasonable'?

For example, consider a question from simple arithmetic: is it true that $22 \times 31 + 11 \times 17 + 13 \times 19 = 1097$? Rather than evaluate the left-hand side of the equation straight away, let us first inspect the problem to see if it is reasonable. We know that when an even number is multiplied by an odd number, the result is always even; and that an odd number multiplied by an odd number is always odd. Therefore the left-hand side of the equation could be rewritten as $\langle even \rangle + \langle odd \rangle + \langle odd \rangle$. Likewise, an even number added to an odd number is always odd, and the sum of two odd numbers is always even. Then evaluating left to right, we have $\langle odd \rangle + \langle odd \rangle$, which is $\langle even \rangle$. Since 1097

is not even, it cannot be the result of the evaluation. We have thus answered the question without having to do the "hard" work of evaluating the numerical value of the left-hand side.

Consider also a common type of schoolboy's "joke" in which additional irrelevant information is intentionally provided as a distraction to the listener:

> "If Mr. McIver drives his train out of Aberdeen towards Edinburgh at 08.38, averaging 60mph for 30 minutes, then 100mph for 60 minutes, then 60mph for a further 30 minutes, stops in Edinburgh for 15 minutes, then proceeds to Glasgow for 30 minutes at an average speed of 30mph..."

Guided by the level of detail provided, the listener is by now expecting a question whose answer involves significant computation, such as "How many miles is it from Aberdeen to Glasgow?". Most listeners will try to remember all the details, and might even attempt to pre-empt the question with some calculations. However, the "joke" continues...

> "...what is the driver's name?"

The analogy with MUSKRAT is that the joke's listener represents a problem solver with eager evaluation, and the joke represents the knowledge base input. Presented with so much information, the "problem solver" was tempted to perform unnecessary computation, because it was not able to look forward to the goal. If the information and the goal had been presented simultaneously, an intelligent agent would only compute what was necessary to determine the goal.

As a further simple example, consider the truth of the statement 'If Pigs can fly, then I'm the Queen of Sheba', which can be written as $P \Rightarrow Q$. Given the background knowledge that the premise P is false, it follows from the truth table for logical implication[6] that the whole statement is true, since any implication with a false premise is true. Notice that I derived the result *without having to know* the truth of the consequent

6.

P	Q	$P \Rightarrow Q$
T	T	T
T	F	F
F	T	T
F	F	T

Q. In a similar way, it is possible to investigate the outputs of programs (in particular, problem solvers) without needing complete information about their inputs. This issue becomes important if running a problem solver has a high cost associated with it, such as the time it takes to perform an intensive search[7]. In such cases, a preliminary investigation of the plausibility of the task at hand could save much time if the intended problem solver configuration can be shown to be unsuitable for the task. I consider such an investigation to be a kind of "plausibility test" which should be carried out before running the actual problem solver. The idea was suggested as part of a general problem solving framework by Polya (Polya, 1957). In his book 'How to Solve It', he proposed some general heuristics for tackling problems of a mathematical nature, including an initial inspection of the problem to understand the condition that must be satisfied by its solution. For example, is the condition sufficient to determine the problem's "unknown"? And is there redundancy or contradiction in the condition? Polya summarised by asking the following:

> 'Is our problem "reasonable"? This question is useful at an early stage of our work *if* we can answer it easily. If the answer is difficult to obtain, the trouble we have in obtaining it may outweigh the gain in interest.' (Polya, 1957).[8]

For a more realistic example, consider the problem of the Königsberg Bridges (Johnson, 1997), in which one would like to find a path through the Prussian city of Königsberg, such that each of the seven bridges over the river Pregel is crossed only once (see Figure 32). Rather than immediately starting to look for a solution, Euler was able to show that it was not possible to solve the problem because a necessary condition is not upheld by the city's topology. This is best expressed by representing the topology as a graph of nodes (A-D) and arcs (a-g), as shown in Figure 32, and calling the number of arcs connected to a node the **degree** of that node. Observe that if the degree of a node is odd, then a path through the graph must begin or end at that node. Since the Königs-

7. Many AI programs perform searches of problem spaces which grow exponentially with the size of the problem.

8. One way to determine whether the plausibility test is useful is to compare the computational complexities of the problem solver and the plausibility test. If the order of complexity of the plausibility test is lower than that of the problem solver, one might assume it is reasonable to apply the plausibility test first. Unfortunately, this model takes no account of the *utility* of the information gained from the plausibility test.

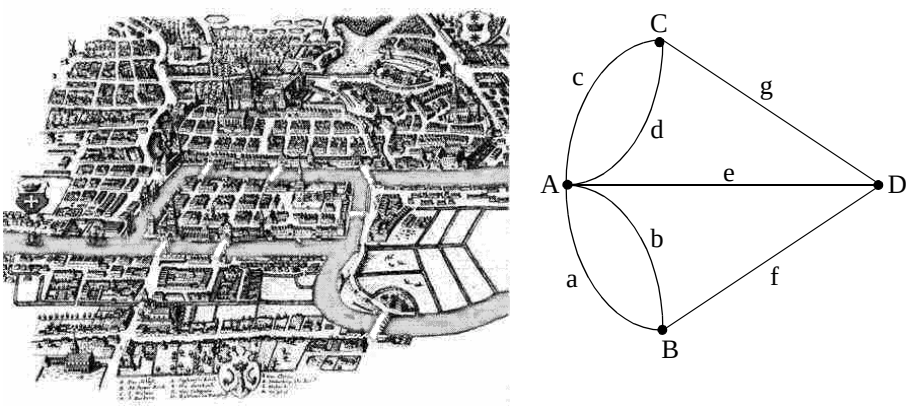


Figure 32: The Königsberg Bridges: Map and Topology

berg Bridges graph contains four nodes of odd degree (i.e., A, B, C & D), such a path does not exist. More generally, if there are more than two nodes of odd degree, then a journey visiting all nodes, and traversing each arc exactly once, is impossible[9].

Let us consider the nature of the four problems described in this section. The example of **logical implication** derived the output value by using knowledge of the operator's properties under the given input conditions. The output was bound to the only value which was *consistent* with the behaviour of the operator. For each of the three other problems much of the detail has been "ignored" or lost when assessing the solution; on the other hand, enough detail has been retained for a test to reflect the task at hand, and for the test to be useful. In this sense, the test is an *approximation.* The approach to these three problems are examples of *abstraction* (Giunchiglia & Walsh, 1992). The **arithmetic example** given above is a TI (truth increasing) abstraction, first mapping each number to a symbol representing its parity before applying a simple algebra. The problem of the **Königsberg Bridges** is also a TI abstraction, because it states a necessary condition on the parities of the degrees of the nodes in the graph. However, the precondition for solving the Königsberg Bridges problem does not vary with the *particular* problem being solved, unlike the precondition for the arithmetic problem. Moreover, there is a straightforward correspondence between a particular arithmetic

9. Euler discovered two other conditions which also apply. Firstly, if there are exactly two nodes of odd degree, then there is a path through the graph if the start point is either of these two nodes. Secondly, if there are no nodes of odd degree, then a path can be found starting from any node.

problem and the precondition to be applied. This relationship is discussed further in chapter 8 (see page 245). The **train joke example** can be solved by a TD (truth decreasing) abstraction that forgets details which are irrelevant to the goal.

In chapter 6, I explain how the ideas of abstraction can be applied to the problem solvers of MUSKRAT. In that chapter, I also demonstrate how constraint technology can be employed to realise such abstractions, and discover when problem solving goals are not achievable. In the following section, I discuss the complementary task of noticing implausible configurations at *acquisition time*, rather than at problem-solving time.

4.4.3 Acquiring Knowledge which is Fit for a Purpose

The second important action which a MUSKRAT toolbox should be able to perform is to *acquire* knowledge such that it is *already* fit for a particular purpose (see Actions 1, 2 and 3 on page 93). This is consistent with the goal-driven approach of MUSKRAT, and means that a flexible notation is needed to specify the knowledge to be acquired, as well as a mechanism for checking the acquired knowledge against the specification.

But what kind of notation could be used? At the symbol level, it is common for data format specifications to be given as context-free grammars. This has also been true of programming languages since the publication of the Algol 60 report in 1960, which contained the first *formal* definition of a computer language's syntax (Goldschlager & Lister, 1982). Using this approach, the syntax can be defined in a way that is both complete and precise.

My initial solution to the knowledge acquisition problem of MUSKRAT was to use a grammar as a machine-readable *target* of the knowledge elicitation process. The grammar can specify not only the concepts of interest, but also any allowable values for those concepts. For example, to acquire the name of a colour one might use the productions:

```
colour → red | <blue> | green
blue   → navy-blue | sky-blue
```

To automatically elicit a choice, a program would interpret these grammar statements, and ask the user to respond to the options given. For example ‘`Would you like red, blue, or green?`’. (For the user-interface to be most effective, the user should not see the grammar itself.) The production rules of the grammar also capture the refinement of concepts and choices. In the above-given example, if the user chooses `blue`, then the option is further refined to one of `navy-blue` or `sky-blue`.

Context-free grammars provide a flexible and declarative notation for specifying the knowledge to be acquired. However, they do have their limitations. Suppose a householder is choosing products from an electronic catalogue to redecorate a room. She wishes to specify that two major items in the room (say the curtains and the bed-linen) should be colour-coordinated. A grammar-based program acquiring the choice of colours should derive two *interpretations* of the production `colour`, but how would it avoid certain clashing colour combinations, such as `green` and `sky-blue`?[10]. The constraint could be implicitly written into the grammar by appropriately limiting the combinations of allowable colours. However, as the number of colours and associated constraints increase, the size and complexity of the grammar which avoids those colour combinations increases dramatically. The grammar is also difficult to maintain if the constraints are subject to change.

In chapter 7, I propose that constraint technology can be used as an additional “layer” on top of the basic grammar notation. In this sense, the constraints *augment* the grammar by restricting the values which non-terminal symbols may take. Furthermore, constraint technology can provide both a declarative notation for the required knowledge, as well as a checking mechanism against that specification.

10. Please do not be offended if this is the colour scheme in your house.

Chapter 5

Constraint Handling in Common LISP

‘Sometimes a scream is better than a thesis.’

Ralph Waldo Emerson

Chapter Summary

In this chapter, the reader is first reminded of the benefits of LISP as a language for artificial intelligence, then introduced to SCREAMER, a portable LISP-based constraint handling package. Shortcomings in the package’s expressiveness led to the development of SCREAMER+, an improved version which extends the original in three major directions. Firstly, it improves the capability to assert constraints on lists and their subparts; secondly, it provides constraint-based versions of higher-order functions such as `some`, `every` and `mapcar`; and, thirdly, it integrates SCREAMER constraints with CLOS, the object-oriented part of Common LISP. The functions of the extension are discussed, and SCREAMER+ is illustrated by application to some benchmark constraint problems. Detailed descriptions of SCREAMER+ functions are given in Appendix A.

5.1 Why LISP is Almost Wonderful[1]

LISP is the second oldest[2] programming language still in use (Russell & Norvig, 1995), and is still one of the dominant languages for Artificial Intelligence. It was defined by John McCarthy at MIT in 1958, and has evolved steadily ever since. It was

1. This was originally the title of a section on symbolic computation in (Jackson, 1990).
2. FORTRAN is one year older.

originally conceived as a list processing language with a very simple core functionality and syntax. In LISP, lists are parenthesised elements which can be used as symbolic representations of real-world objects or relationships, as in:

```
'(parents (philip elizabeth) (charles anne andrew edward))
```

Such an expression is, of course, subject to further processing and semantic interpretation when passed as an argument to a LISP function. The most basic functions for manipulating lists are `car` and `cdr` (also known as `first` and `rest`). The `car` of a list is its head, and the `cdr` is its tail. So, for example:

```
>(car '(parents (philip elizabeth) (charles anne andrew edward)))
PARENTS
>(cdr '(parents (philip elizabeth) (charles anne andrew edward)))
((PHILIP ELIZABETH) (CHARLES ANNE ANDREW EDWARD))
```

The basic syntax of a function call is very simple:

$$(function\ arg_1\ arg_2\ \ldots\ arg_n)$$

The brackets not only delimit the expression; they also signify that the function call is something to be *evaluated;* that is, a result is computed and returned. Notice that in the previous expression above, a quote (`'`) was used to protect the argument from immediate evaluation. The arguments provided to a function can themselves be the *result* of function calls, so LISP expressions often have multiple "layers" of bracketing; for example $\sqrt{x^2 + y^2}$ is written as:

```
(sqrt (+ (* x x) (* y y)))
```

A function *definition* also has this same basic form, so a LISP program consists of a set of such expressions. A major advantage of this approach is that a LISP function can be manipulated in the same way that data is manipulated. For example, functions can be passed as arguments to so-called *higher-order functions*, and functions can also generate *functions* as their output.

One of the most commonly used higher-order functions is called `mapcar`. It applies the given function to respective members of the arguments, collecting and returning the results as a list. For example, the function `integerp` determines whether or not its single argument is an integer, returning `T` for true, and `NIL` for false:

```
> (mapcar #'integerp '(b 3 (8) nil))
(NIL T NIL NIL)
```

Note that the hash sign (#) told LISP that `integerp` is a function, and can therefore be applied to other data. A function which takes multiple arguments, such as `+`, can also be supplied to `mapcar`, with powerful effect:

```
> (mapcar #'+ '(1 2 3) '(10 20 30) '(100 200 300))
(111 222 333)
```

On a more pragmatic level, LISP is (usually) both an *interpreted* and a *compiled* language. Interpreted languages are good for rapid prototyping because they generally provide good error diagnostics, and the *implement-test-refine* cycle does not require recompilation. LISP thus enables the programmer to first develop code quickly, then later compile the result into a more efficient form.

Before the mid-1980s, LISP existed in many different dialects, such as Franz LISP, InterLISP, MacLISP, SPICE LISP and ZetaLISP. In 1986, an ANSI standard for LISP emerged in the form of a language specification; now known as **Common LISP**. Common LISP[3] combines the best features of many of the earlier dialects, and is a powerful and comprehensive programming language. (In the rest of this thesis, 'LISP' refers only to Common LISP.) Unlike many other programming languages, such as C and Prolog, LISP defines many of the commonly used functions *as part of the language* rather than a library extension. Although this means that LISP manuals tend to be large, it also means that Common LISP programs are very portable across different Common LISP implementations because each implementation provides the same basic set of functions (even if the functions have been implemented in different ways).

Like many other modern programming languages, Common LISP is object-oriented. In fact, it was the first ANSI-standard object-oriented language, incorporating **CLOS** (the Common LISP Object System). CLOS provides a set of functions which enable the definition of **classes**, their **inheritance hierarchies** and their associated **methods**.

3. See `http://www.xanalys.com/software_tools/reference/hyperspec.html`

A class defines **slots** whose values carry information about object instances[4]. The class definition can also specify **default values** for slots, and additional non-trivial behavioural mechanisms (such as integrity checking) performed at object creation time.

```
(defclass shape ()
  (x y))

(defclass rectangle (shape)
  ((width :initarg :width)
   (height :initarg :height)))

(defclass circle (shape)
  ((radius :initarg :radius)))

(defmethod area ((obj circle))
  (let ((r (slot-value obj 'radius)))
    (* pi r r)))

(defmethod area ((obj rectangle))
  (* (slot-value obj 'width)
     (slot-value obj 'height)))
```

Figure 33: Simple CLOS Definitions of Rectangles and Circles

As an example of object-orientation in LISP, consider the definition of some `shapes`, which can be either `circles` or `rectangles`. A shape's position is given by `x` and `y` coordinates. Further, a circle has a `radius`, whereas a rectangle has a `width` and a `height`. It should be possible to compute the `area` of circles and rectangles. A working set of definitions is given in Figure 33. Using these definitions, a rectangle (instance) can be created with `make-instance`, for example with a width of 3 and a height of 4:

```
> (setq my-rect (make-instance 'rectangle :width 3 :height 4))
#<RECTANGLE @ #x8aee2f2>
```

and its area can be computed by calling the method `area`:

```
> (area my-rect)
12
```

Apart from the general advantages of LISP as an AI programming language, there were two main factors influencing the choice of implementation language for the work described in this dissertation. Firstly, I made use of some LISP legacy code from the in-

4. If required, a class can also define slots with *class allocation*. The value associated with such a slot is the same among all members of the class.

itial MUSKRAT work of Nicolas Graner. This code was an initial implementation of some problem solvers in the culinary domain (see “Meal Design and Scheduling” on page 183). Secondly, and more significantly, most of my programming experience was acquired using LISP, so that its choice did not imply a learning curve.

5.2 The SCREAMER Constraint Solving Package

SCREAMER (Siskind & McAllester, 1993; Siskind & McAllester, 1994) is one of the very few constraint handling packages available for LISP[5]. It is a freely available extension of Common LISP (Steele, 1990) that provides for nondeterministic[6] and constraint-based programming. Developed at the MIT AI Laboratory and the University of Pennsylvania, it is portable across most modern Common LISP implementations.

It provides a very useful basis for constraint programming in LISP, but I found it lacking in some important aspects. For example, although SCREAMER appears to provide ample facilities for efficient numeric constraint handling, it is surprising to see that it provides very little support for many usual LISP operations, particularly the manipulation of lists. I have addressed these and other shortcomings by providing an additional library of functions, called SCREAMER+, to extend the original functionality of SCREAMER. This library contains many important new functions, including those for expressing and manipulating constraints on LISP lists (including lists interpreted as sets), some additional higher order constraint functions (such as a constraint-based version of `mapcar`), and also some functions for dealing with constraints on CLOS objects.

5.2.1 Nondeterminism using SCREAMER

SCREAMER adds nondeterminism to LISP through an implementation of PROLOG-like *backtracking*. To do this, it defines a *choice point* operator and a *fail* operator. In a SCREAMER program, choice points are most easily introduced with the macro `ei-`

5. The only other LISP-based constraint-handling package I am aware of was called PECOS, implemented using LeLISP, and available from Ilog in the late 1980s. Unfortunately, this is no longer available, and I was unable to obtain any further literature describing PECOS.
6. A *nondeterministic* computation may have more than one result. One way to implement nondeterminism is through *backtracking*.

ther, and failures are generated with the function fail. Functions and other LISP expressions that define choice points but do not attempt to retrieve values from them are called **nondeterministic contexts**. A small number of new functions and macros are also supplied by SCREAMER for retrieving values from *nondeterministic contexts*, notably one-value and all-values. Let us now illustrate these ideas with the following simple example:

```
> (one-value (either 'black 'blue))
BLACK
```

The special form either has set up a choice point, providing a *nondeterministic context* for retrieving a single value with the macro one-value. If the same *non-deterministic context* is supplied to the macro all-values, then the list '(BLACK BLUE) is returned instead of the single symbol 'BLACK. Consider also the following examples, which explore the possible values at *two* choice points:

```
> (one-value (list (either 'tall 'short) (either 'fat 'thin)))
(TALL FAT)
> (all-values (list (either 'tall 'short) (either 'fat 'thin)))
((TALL FAT) (TALL THIN) (SHORT FAT) (SHORT THIN))
```

Either can be called with any number of expressions as arguments, and initially returns the value of the first supplied expression to the surrounding *nondeterministic context*. If a failure later causes a backtrack to this choice point, the next expression will be chosen and returned. If SCREAMER backtracks to a choice point which has no further values, then backtracking continues to the previous choice point. If no choice points remain, an error is generated.

Either also forms the basis of several other more specialised nondeterministic forms. For example, consider defining a function an-integer-between, which nondeterministically returns integers between two supplied integers *low* and *high*:

```
(defun an-integer-between (low high)
  (when (or (not (integerp low)) (> low high)) (fail))
  (either low (an-integer-between (1+ low) high))
  )
```

The first line of this definition traps unexpected parameter values and halts recursion; the second line nondeterministically returns either the integer low itself, or, should that fail, an integer between low + 1 and high. A similar function can be defined to non-

deterministically return the members of a list. SCREAMER already provides many of the commonly-used nondeterministic functions, as well as some macros for protecting LISP variables from any local side-effects caused by backtracking.

Nondeterministic programming using SCREAMER is exemplified by a solution[7] to the N-Queens problem, as given in Figure 34.

```
(defun attacks-p (qi qj distance)
  (or (= qi qj)
      (= (abs (- qi qj)) distance)))

(defun check-queens (queen queens &optional (distance 1))
  (unless (null queens)
    (if (attacks-p queen (first queens) distance) (fail))
    (check-queens queen (rest queens) (1+ distance))))

(defun n-queens (n &optional queens)
  (if (= (length queens) n)
      queens
    (let ((queen (an-integer-between 1 n)))
      (check-queens queen queens)
      (n-queens n (cons queen queens)))))

(defun queens (n)
  (dolist (sol (one-value (n-queens n)))
    (dotimes (c n)
      (format t " ~a" (if (= (1- sol) c) "Q" "+")))
    (terpri)
    )
  )
```

Figure 34: A Solution to the N-Queens Problem using nondeterministic programming in SCREAMER

5.2.2 Constraint Handling using SCREAMER

The SCREAMER constraints package provides LISP functions which enable a programmer to **create constraint variables,** (often referred to simply as *variables*), **assert constraints** on those variables, and **search for assignments of values to variables** according to the asserted constraints.

7. The code is adapted from (Siskind and McAllester, 1993).

5.2.2.1 Creating Constraint Variables

An unbound constraint variable, *x*, can be created with a call to the function `make-variable`; for example:

```
> (setq x (make-variable))
[40]
```

It returns a constraint variable structure, whose printed representation includes the name of the variable in brackets `[]`. By default, the name is the value of an integer counter which is incremented each time that a new variable is created. If desired, however, a symbolic name can be supplied as an optional argument:

```
> (make-variable 'fred)
[FRED]
```

`Make-variable` creates variables which are completely unconstrained. It is also possible to create variables which are already constrained. For example, a variable can be created such that it is constrained to be a number, an integer within a particular range, or one of a set of known values:

```
> (setq x (a-numberv))
[41 number]
> (setq y (an-integer-betweenv 1 10))
[42 integer 1:10 enumerated-domain:(1 2 3 4 5 6 7 8 9 10)]
> (setq z (a-member-ofv '(foo bar)))
[45 nonnumber enumerated-domain:(FOO BAR)]
>
```

Notice that if they are known, the type and the enumerated domain of a variable are also printed. For numbers, the upper and lower bounds of the range of values are also given. This feedback is not only reassurance that SCREAMER is working as expected, it also provides invaluable debugging information to the programmer when things are not working as expected.

The functions which create constrained variables are really just a shorthand notation for creating a constraint variable with `make-variable`, and then afterwards asserting some additional constraints. Constraints can be asserted with a range of other functions, as described below.

5.2.2.2 Asserting Constraints

Assertions are carried out using **constraint primitives**, which are generally constraint-based counterparts of some Common LISP function. So, for example, `integerpv` is the SCREAMER constraint primitive corresponding to the LISP function `integerp`, which determines whether its argument is an integer. It is a convention that constraint primitives end in the letter 'v', because each of them creates and returns a constraint variable to hold the result of evaluating the expression. To assert truths about constraint variables, one uses the primitive `assert!`, which takes an expression as its argument and binds the constraint variable associated with that expression to true. So, for example, the expression `(assert! (integerpv x))` constrains `x` to be an integer. Likewise, the expression `(assert! (andv (>=v x 1) (<=v x 10)))` sets the inclusive range of *x* to be between 1 and 10.

As soon as SCREAMER realises that an assertion over-constrains the problem, it generates a failure. This has been implemented as a LISP `throw`, which, without a corresponding `catch`, causes LISP to enter the debugger. The following extract demonstrates what happens when the Boolean variable *a* is constrained such that $a \wedge \neg a$ is true.

```
> (setq a (a-booleanv))
[48 Boolean]
> (assert! (andv a (notv a)))
Error: Attempt to throw to the non-existent tag FAIL
  [condition type: CONTROL-ERROR]

Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart)
[1] USER(42):
```

Sometimes, SCREAMER may not immediately realise that an assertion over-constrains the problem. For example, in the following, the variable *n* is first created as an integer between 1 and 2; i.e., $n \in \{1, 2\}$. Then, a constraint is asserted that *n* is neither 1 nor 2. This over-constrains the problem without generating an immediate failure.

```
> (setq n (an-integer-betweenv 1 2))
[49 integer 1:2 enumerated-domain:(1 2)]
> (assert! (andv (notv (=v n 1)) (notv (=v n 2))))
NIL
>
```

However, the search for a solution that satisfies the constraints *does* result in failure:

```
> (one-value (solution n (static-ordering #'linear-force)))
Error: Attempt to throw to the non-existent tag FAIL
  [condition type: CONTROL-ERROR]

Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart)
[1] USER(49):
```

The reason for this behaviour is that the function `=v`, which returns a Boolean constraint variable to indicate whether two numbers are equal, has weak propagation properties. More specifically, the truth of the equality relation is not checked until its arguments receive values as part of the search procedure.

5.2.2.3 Searching for Solutions

Once a problem has been set up by creating variables and asserting constraints on those variables, solutions can be sought with the nondeterministic function `solution`. This function explores the search space of domain values, returning each solution found to its surrounding nondeterministic context. Two arguments must be supplied to `solution`: firstly the variable(s) requiring assignments (or a `cons` structure containing those variables), and secondly, a function for ordering the variables at each choice point (i.e., a **variable ordering strategy**; see "Variable Ordering Strategies" on page 53). The simplest ordering function is a static-ordering which forces a constraint variable to assume each of its domain values in turn, until a solution is found. Such an ordering function is returned by `(static-ordering #'linear-force)`. Therefore, if `x` were a constraint variable with an enumerated domain, then I could find all of its possible values with:

```
(all-values (solution x (static-ordering #'linear-force))).
```

It is also possible to supply `solution` with a *dynamic variable ordering strategy* by using the flexible function `reorder`[8]. For example, the following code applies the *fail-first strategy* (see page 54):

8. `Reorder` takes four arguments: a cost function, a termination condition function (useful for stating when a numerical approximation is satisfactory), a function to determine whether the greatest or least cost variable should be forced next, and a forcing function (a forcing function nondeterministically returns the successive values to be assigned to a variable).

```
(all-values (solution x (reorder #'domain-size
                                 #'(lambda(x) nil)
                                 #'<
                                 #'linear-force)
```

It is also worth noting that SCREAMER can search for optimal solutions to a problem by calling `best-value` instead of `all-values` or `one-value`. `Best-value` must be supplied with two arguments: a *nondeterministic context*, such as the result of a call to `solution`; and an expression representing the numeric optimisation function. When called, it returns a pair – the solution found and the corresponding value of the optimisation function. For example, the following finds the greatest value of $10 - x^2$ for integral values of *x* in the range [-4, +4].

```
> (setq x (an-integer-betweenv -4 4))
[60 integer -4:4 enumerated-domain:(-4 -3 -2 -1 0 1 2 3 4)]
> (best-value (solution x (static-ordering #'linear-force)) (-
v 10 (*v x x)))
(0 10)
```

5.2.3 How the Constraints Package of SCREAMER Works

First of all, as with most constraint solvers, SCREAMER's problem solving process can be considered to be split into two main phases. I refer to the first phase, in which the constraint variables and their domains are established, the relationships between the variables are asserted, and, where possible, domain values eliminated, as the **problem set-up**. The second phase is called **searching** and involves the systematic assignment of remaining domain values to variables until consistent combinations are discovered.

5.2.3.1 Problem Set-up

At *problem set-up* time, a LISP data structure is created for each constraint variable of the problem (the data structure is created by the function `make-variable`). This data structure contains everything that is known about the variable, either explicitly or implicitly. The structure contains slots for the variable's name, type information, domain values, upper and lower bounds (for numbers), and, once assigned, its value. A programmer can inspect the constraint variable's internal data structure by using the Common LISP function `describe`. For example, the following statements create and display the structure of a variable which is an integer between 0 and 4, but not 2.

```
> (setq x (an-integer-betweenv 0 4))
[84 integer 0:4 enumerated-domain:(0 1 2 3 4)]
> (assert! (notv (=v x 2)))
NIL
> (describe x)
[84 integer 0:4 enumerated-domain:(0 1 2 3 4)] is a structure
of type SCREAMER+::VARIABLE+.  It has these slots:
 NAME               84
 NOTICERS
   (#<Closure (:INTERNAL SCREAMER::ASSERT!-/=V2 0) @ #x861c4d2>
    #<Closure (:INTERNAL SCREAMER::ASSERT!-<=V2 0) @ #x861bdf2>
     #<Closure (:INTERNAL SCREAMER::ASSERT!-<=V2 1) @
#x861b9aa>)
 ENUMERATED-DOMAIN  (0 1 2 3 4)
 ENUMERATED-ANTIDOMAIN  NIL
 VALUE            [84 integer 0:4 enumerated-domain:(0 1 2 3 4)]
 POSSIBLY-INTEGER?  T
 POSSIBLY-NONINTEGER-REAL?  NIL
 POSSIBLY-NONREAL-NUMBER?  NIL
 POSSIBLY-BOOLEAN?  NIL
 POSSIBLY-NONBOOLEAN-NONNUMBER?  NIL
 LOWER-BOUND        0
 UPPER-BOUND        4
 NONNUMBER-TYPE     NIL
>
```

Note that although x has been constrained to have a value other than 2, the enumerated domain still contains that value. The reason for this behaviour is that the SCREAMER constraint function =v has weak propagation properties (as mentioned on page 109). The constraint has not been forgotten, but instead has been retained as a **noticer**, and will be checked whenever x receives a value as part of a search. A *noticer* is a body of executable code that incorporates and assures the conditions associated with a constraint. *Noticers* are used when the truth of a constraint condition cannot be immediately upheld by other means, such as domain reduction[9], or they can be used to propagate the properties of a constraint variable to other variables. In the example above, three noticers are associated with x. The first of these guarantees x's disequality with 2; the other two appear as a result of the implicit constraints that `(>=v x 0)` and `(<=v x 4)`, and propagate truth values to the boolean outputs of these relational expressions.

As another example of "delayed constraint checking" consider what happens if I create a variable without an enumerated domain and then constrain its value to be something other than the symbol `'foo`. This information is retained as a noticer:

9. The function =v is an exception – the designers of SCREAMER decided to use only bounds propagation for numbers because it is faster than domain reduction – see "Searching" on page 112.

```
> (setq f (make-variable))
[7]
> (assert! (notv (equalv f 'foo)))
NIL
> (describe f)
[7] is a structure of type SCREAMER+::VARIABLE+.  It has these
slots:
 NAME              7
 NOTICERS
    (#<Closure (:INTERNAL SCREAMER::ASSERT!-NOTV-EQUALV 0) @
       #x20685b52>)
 ENUMERATED-DOMAIN  T
 ENUMERATED-ANTIDOMAIN  NIL
 VALUE             [7]
 POSSIBLY-INTEGER?  T
 POSSIBLY-NONINTEGER-REAL?  T
 POSSIBLY-NONREAL-NUMBER?  T
 POSSIBLY-BOOLEAN?  T
 POSSIBLY-NONBOOLEAN-NONNUMBER?  T
 LOWER-BOUND       NIL
 UPPER-BOUND       NIL
 NONNUMBER-TYPE    NIL
>
```

If I now attempted to assert that `f` had the value `'foo`, the assertion would fail because of the knowledge contained in the noticer:

```
> (assert! (equalv f 'foo))
Error: Attempt to throw to the non-existent tag FAIL
  [condition type: CONTROL-ERROR]
Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart)
[1] >
```

Noticers are invoked whenever a constraint variable is in any way "touched", either directly by means of constraint assertion or value assignment (e.g., as part of the search process); or indirectly by means of propagation. Note that the actions of one noticer can cause other noticers to be invoked, with the possible effect of propagating knowledge through the network.

5.2.3.2 Searching

At problem set-up time and at search time, SCREAMER employs several different propagation techniques. At problem set-up time, as we have seen, propagation may result in a variable having no allowable domain values. This causes an immediate failure. At search time, the assignment of values to variables can also provoke propagation, and

perhaps cause failure. Any failures generated in this way cause the search to backtrack and try new value assignments to variables. SCREAMER's efficiency is achieved by this interleaving of search with propagation.

The propagation techniques employed by SCREAMER are *unification*, *binding propagation*, *Boolean constraint propagation*, *generalised forward checking*, and *propagation of bounds*.

Unification – *Unification* operates on equality constraints. When a constraint variable *x* is asserted to be `equalv` to a constraint variable *y*, then any constraints on *x* are also imposed on *y*, and vice versa. Constraints asserted on *x* some time later are also passed on to *y*. Thus, values are propagated between any variables which are constrained to be equal. Similarly, after asserting `(notv (equalv x y))`, any attempt to bind *x* and *y* to be equal will immediately fail.

Binding Propagation – *Binding propagation* (also called *value propagation*) attempts to assign a value to a constraint variable if it is the only unbound variable in a constraint. Consider a constraint variable *z* = `(f` x_1 x_2 x_3 ... x_n`)`, which is constrained to be the result of applying some function `f` to the (constraint variable) arguments $x_1 \ldots x_n$. Binding propagation will always bind the output variable of the constraint primitive whenever all of the input arguments are bound. For example, if *z* is constrained to be the sum of *x* and *y*, *z* = `(+v` *x y*`)`, and at some point *x* becomes bound to 2, and *y* becomes bound to 3, then *z* automatically becomes bound to 5. Under some circumstances, binding propagation can also derive an input binding from an output binding. Using the same arithmetic example, if *x* became bound to 2, and *z* became bound to 5, then *y* would automatically become bound to 3.

Note that the extent to which propagation can occur from an output binding to an input binding may be prohibited by the nature of the function involved in the constraint. For example, suppose we defined a constraint-based version of the Common LISP function `zerop`, which returns true (`T`) if its numeric argument is zero, and false (`NIL`) otherwise. Then we could set up a constraint such that *z* `=(zeropv` *x*`)`. In this situation, if the value of *x* is known, then *z* can always be derived. When propagating in the other

direction, the nature of the propagation depends on the value of *z*. If we knew *z* were true, then we could derive $x = 0$. If we knew *z* were false, however, then we could only derive the much weaker statement that $x \neq 0$.

Boolean Constraint Propagation – *Boolean constraint propagation* implements arc consistency for the boolean constraint primitives `andv`, `orv` and `notv`. That is, values (either `T` or `NIL`) are immediately removed from the domains of boolean constraint variables if they are inconsistent with the constraints of that variable. For example, consider what happens if *z* is constrained to be the logical conjunction of *x* and *y*; i.e., *z* `=(andv` *x y*`)`. Then, clearly, as soon as *x* and *y* become bound, *z* can be derived. When propagating in the other direction, suppose firstly that *z* is known to be true. Then the values of *x* and *y* must both be true, since a logical AND only produces a true output when all of its inputs are true. On the other hand, if *z* is known to be false, then nothing can be deduced immediately about the truth or falsehood of any of the inputs. But if we know additionally that *x* is true, then we can derive that *y* is false, because if *y* were true then *z* would be true, and that would contradict what we already know about *z*.

Note that Boolean constraint propagation is a stronger form of propagation than binding propagation because it can sometimes determine a value when there is more than one unknown involved in a constraint.

Generalised Forward Checking – *Generalised Forward Checking* (see page 56) is applied to constraint variables which have an enumerated domain, and have been constrained using the SCREAMER constraint primitive `funcallv`. `Funcallv` accepts a conventional LISP function as its first argument, together with the constraint variables to which the function should be applied. For example, `(setq z (funcallv #'first x))` would constrain *z* to be the first element of the sequence *x*. Now, if *x* is known to be one of the lists `'(a b c)`, `'(d e f)` or `'(g h i)`, and *z* becomes bound to the symbol `a`, then `'(d e f)` and `'(g h i)` can be removed from the domain of *x* because they are inconsistent with the constraint. Since this leaves an enumerated domain of size one, *x* immediately becomes bound to the value `'(a b c)`.

Propagation of Bounds – *Propagation of bounds* applies only to numeric variables, for which SCREAMER maintains a maximum and a minimum value. When numeric variables are combined using an arithmetic constraint operation, then knowledge of their ranges can be propagated. For example, if *z* is constrained to be the sum of *x* and *y*, *x* is known to be an integer between 0 and 3, and *y* is known to be an integer between 3 and 6; then the range of *z* is derived to be between 3 and 9:

```
> (setq x (an-integer-betweenv 0 3))
[25 integer 0:3 enumerated-domain:(0 1 2 3)]
> (setq y (an-integer-betweenv 3 6))
[28 integer 3:6 enumerated-domain:(3 4 5 6)]
> (setq z (+v x y))
[31 integer 3:9 enumerated-domain:(3 4 5 6 7 8 9)]
>
```

Propagation of bounds can also derive Boolean values for numeric relations such as `=` and `>`. It does this by inspecting whether the ranges of the variables involved in a constraint overlap. For example, if, as above, $0 \le x \le 3$ and $3 \le y \le 6$, then `(>v x y)` will immediately return false (`NIL`).

5.3 The Extension to SCREAMER

Type Restrictions:	`numberpv realpv integerpv booleanpv memberv`
Boolean:	`andv orv notv`
Numeric:	`<v <=v >v >=v =v /=v +v -v *v /v minv maxv`
Expression:	`equalv`
Function:	`funcallv applyv`

Table 5: The Constraint Primitives of SCREAMER

I was impressed with the way that SCREAMER integrates nondeterminism and constraint-handling into Common LISP. It appears to be very good for solving sets of logical and/or numeric constraints; indeed, it is superior to many other constraint solvers in its ability to deal with sets of non-linear constraints. I was surprised, however, at the lack of facilities for symbolic constraint handling, such as the expression of constraints on lists. Two of the major features of Common LISP are its list handling and its provi-

sion of higher-order functions (functions which take functions as arguments), such as `mapcar`. I therefore felt that in order to maximise the utility of constraints in LISP, SCREAMER should be extended to embrace those qualities. In addition, I felt the ability to impose constraints on Common LISP objects would be useful for more sophisticated data/knowledge modelling problems. The functions of my extension to SCREAMER have been designed to provide the required functionality for these three main directions.

Type Restrictions[a]:	`listpv conspv symbolpv stringpv typepv`
Boolean:	`impliesv`
Expression:	`ifv make-equal setq-domains`
Lists[b]:	`carv cdrv consv firstv restv secondv thirdv fourthv nthv subseqv lengthv appendv make-listv all-differentv`
Sets and Bags:	`set-equalv subsetpv intersectionv unionv bag-equalv`
Arrays:	`make-arrayv arefv`
Objects:	`make-instancev classpv slot-valuev class-ofv class-namev slot-exists-pv reconcile`
Higher Order Fns:	`funcallinv mapcarv maplistv everyv somev noteveryv notanyv at-leastv at-mostv exactlyv constraint-fn`
Stream Output:	`formatv`

a.The following constraint variable generators are also defined: `a-listv`, `a-consv`, `a-symbolv`, `a-stringv`, and `a-typed-varv`.
b. Surprisingly, a function `listv` is unnecessary because the LISP function `list` already works with constraint variables as arguments.

Table 6: The Additional Constraint Primitives of SCREAMER+

To assess the scope of SCREAMER and its extension, SCREAMER+, refer to tables 5 and 6. Table 5 summarises the constraint primitives provided by SCREAMER, and Table 6 summarises the additional primitives provided by SCREAMER+. The primitives of SCREAMER are documented in (Siskind, 1991); those of SCREAMER+ are outlined in this chapter, and documented more fully in Appendix A.

I believe that SCREAMER is difficult for seasoned LISP programmers to grasp initially, because its programs tend to be a mixture of nondeterministic and/or constraint-based programming, together with conventional LISP functions. At first, I found it easier to write nondeterministic programs than constraint-based ones. However, I believe that the efficient search procedure (see section 5.2.3.2 on page 112) of the constraints package combined with its superior ability to deal with infinite domains makes it a more attractive approach for most applications. For this reason, I have only extended the constraints package of SCREAMER; the underlying nondeterminism of SCREAMER remains unchanged.

Another advantage of the constraints package is that it lends itself to an interactive session, because it is easy to inspect intermediate results. For example, the following demonstrates the immediately visible effects of constraint assertion:

```
;;; Create a constraint variable
> (setq x (make-variable))
[72]

;;; Assert x to be an integer
> (assert! (integerpv x))
NIL                              ; assert! always returns nil
;;; Inspect the value of x
> x
[72 integer]

;;; Assert x to be between 0 and 10
> (assert! (andv (>=v x 0) (<=v x 10)))
NIL
;;; Inspect the value of x
> x
[72 integer 0:10 enumerated-domain:(0 1 2 3 4 5 6 7 8 9 10)]

;;; Assert x to be even
> (assert! (funcallv #'evenp x))
NIL
;;; Inspect the value of x
> x
[72 integer 0:10 enumerated-domain:(0 2 4 6 8 10)]
>
```

Unfortunately, not all constraints have an immediate effect on the domain values of variables. When the condition associated with the constraint is wrapped up into a *noticer*, the domain values remain unchanged. Moreover, the condition associated with a noticer cannot (easily) be inspected. I feel that since the inspection of intermediate values is such an advantage, the propagation from constraints to domain values should occur as early as possible. Early propagation also enables inconsistencies to be detected early. I recognise that too much propagation can impinge on efficiency when searching for solutions, but my main objective (described chapter 6) was to provide a *rich notation* for expressing constraints, and to *detect inconsistencies* as early as possible. My approach fulfils these requirements by providing SCREAMER with additional constraint-based functions which echo the semantics of LISP, together with a mechanism for maintaining the consistency of variables used by those functions[10]. (See “Consistency Maintenance” on page 55.) I am more interested in *quickly developing a constraint program which solves the problem* than *developing a program which quickly solves the problem*.

Unfortunately, SCREAMER lacks the ability to *retract* constraints, so that some previous problem solving state can be resumed. This is important because the assertion of a constraint which ultimately fails can irrevocably alter the values of constraint variables[11]. I found that this problem can be circumvented either by taking copies[12] of constraint variable structures *before* assertions are made, or checking first that the assertion does not fail by using the SCREAMER macro `possibly?`.

I found that in some cases, SCREAMER had not carried some of the finer detail of LISP through to its constraint-based counterparts. Most importantly, the SCREAMER function `equalv` had not been defined in terms of the LISP function `equal`, but in terms of the function `eq` instead! This led to some unexpected results; for example, `(equalv “foo” “foo”)` returned `nil`. This probably reflects the fact that SCREAMER was originally designed for solving constraint problems with *atomic* (e.g. numbers,

10. I have written noticers for most of the new functions of SCREAMER+
11. This does not apply to nondeterministic failure within the scope of a call to `solution`.
12. Constraint variables are stored in LISP structures, so a temporary copy of its state can be made by the LISP function `copy-structure`

Boolean values), rather than more general *symbolic* solutions (such as lists). In the copy of SCREAMER used for my work, I changed the definition of `equalv` so that it does mirror the LISP definition of `equal`.

Similarly, I noticed that `memberv` does not accept a keyword *test* argument, like the Common LISP function `member`. (Recall that `member` tests for the equality of some value against each of the members of some list using a test function, which can be supplied as an argument, but is `eq` by default). Thus, it is not possible in SCREAMER to constrain a variable to be one of, say, three candidate lists. In SCREAMER+, I have changed the definition of `memberv` so that it always uses the test `equal` instead of `eq`. This is sufficient to allow us to work with compound structures such as lists.

As a final observation, I found that the SCREAMER function `(funcallv` $f\ x_1 \ldots x_n$`)` makes an idempotency[13] assumption of its function argument. That is, rather than call the supplied function just once as soon as the other arguments became bound, the function was called many times! Often, this is not too detrimental, impinging on efficiency but not on correctness. On the occasions when the function *f* has side-effects, however, the difference can be crucial. The problem is illustrated with the following example:

```
> (setq a (make-variable))
[1]

> (funcallv #'print a)        ; Print a when it becomes bound
[2]

> (assert! (equalv a 'hello))
HELLO
HELLO
HELLO
NIL
```

As soon as the constraint variable became bound, the message “HELLO” was output not once, but three times! Note that this particular problem has been addressed by the function `formatv` in SCREAMER+, but the idempotency assumption of `funcallv` remains.

13. A function is idempotent if repeated applications have the same effect as a single application.

In the following subsections, I summarise the extensions to SCREAMER in the three areas of asserting constraints on lists, higher-order functions and CLOS objects. A complete set of descriptions of the functions in the SCREAMER+ extension is contained in Appendix A.

5.3.1 Constraints on Lists

SCREAMER does not provide any functions specifically for asserting constraints on lists and their associated values. The SCREAMER programmer must instead use the general-purpose functions `funcallv` and `applyv`. These functions are invoked with another function as an argument, together with the associated constraint variables to which the function will eventually be applied. For example, to construct a list whose tail is not yet determined, one can use:

```
> (setq a (funcallv #'cons 'head (a-member-ofv '((one) (two)))))
[100]
```

The variable `a` is now constrained to be the result of `cons`ing the atom `'head` with a list, namely either `'(one)` or `'(two)`. Unfortunately, the knowledge of the inputs to `cons` has not been propagated immediately to the outputs. Instead, it has been retained in a noticer and will be used at solution *search* time. The SCREAMER+ function `consv`, on the other hand, propagates such knowledge as early as possible so that it can be used *prior* to searching the solution space. In a similar way, `carv` can be used to find the head of a list before the list becomes bound. Continuing the example above, compare the SCREAMER and SCREAMER+ code fragments given below in which a list with an unbound (but finite domain) tail is first constructed, before its length and 'head' are computed.

```
;;; *** SCREAMER Version ***
;;; Construct a list in which the tail is not yet determined
> (setq a (funcallv #'cons 'head (a-member-ofv '((one) (two)))))
[100]
;;; the length of the list is not directly accessible
> (funcallv #'length a)
[101]
;;; the head of the list is also not directly accessible
> (funcallv #'car a)
[102]

;;; *** SCREAMER+ Version ***
;;; Construct a list using the same arguments as above
```

```
> (setq a (consv 'head (a-member-ofv '((one) (two)))))
[103 nonnumber enumerated-domain:((HEAD ONE) (HEAD TWO))]
;;; Retrieve its length
> (lengthv a)
2
;;; Retrieve the head of the list
> (carv a)
HEAD
```

The SCREAMER+ version was able to propagate its knowledge earlier because the function definitions are more specialised. Knowledge of the behaviour of each basic Common LISP function (e.g. `cons` and `car`) has been used to optimise their constraint-based equivalents. Other list manipulation functions, such as `appendv`, `nthv`, and `subseqv` have been optimised in a similar way. See Appendix A for further details.

5.3.2 Constraints on Higher-Order Functions

The higher order functions defined by SCREAMER+ enable the programmer to express complex constraints over lists. For example, `somev` is a constraint-based version of the higher-order function `some`. `Some` returns true if at least one of the supplied list arguments satisfies the given predicate:

```
> (some #'integerp '(e bar "gum" 5))
T
> (some #'integerp '(e bar "gum"))
NIL
```

The constraint-based version, `somev`, returns a Boolean variable constrained to indicate whether the given (constraint-based) predicate is true of at least one of the members of the supplied list. As you would expect, if the arguments to `somev` are bound at the time of function invocation, it returns the same (bound) result as the equivalent call to `some`:

```
> (somev #'integerpv '(e bar "gum" 5))
T
> (somev #'integerpv '(e bar "gum"))
NIL
```

In the following example, however, the list supplied to somev has been constructed to contain a single unbound value. Since the other values in the list are non-integers, constraining `some` member of the list to be an integer propagates that property through to the unbound constraint variable:

```
> (setq a '(e bar "gum" ,(make-variable)))
(E BAR "gum" [5603])

> (assert! (somev #'integerpv a))
NIL

> a
(E BAR "gum" [5603 integer]) ; now known to be an integer
>
```

The functional argument supplied to the higher-order function can either be a combination of SCREAMER+ functions, or implemented as a conventional LISP function. If it is a conventional LISP function, then it can be converted ("lifted") to a constraint-based one by using the SCREAMER+ function `constraint-fn`. So, for example, suppose we defined a conventional LISP function which XOR'ed two truth values:

```
(defun xor (x y) (not (eq x y)))
```

then we could generate a constraint-based version of the same function (technically a *closure*[14]) thus:

```
>(setq xorv (constraint-fn #'xor))
#<Interpreted Closure (:INTERNAL CONSTRAINT-FN) @ #x8aa1842>
```

This can now be applied to (or `funcall`ed with) constraint variables such that the outputs become bound when the inputs become bound:

```
;;; Introduce an unbound boolean constraint variable
>(setq a (a-booleanv))
[160 Boolean]

;;; Constrain z to be the result of XOR'ing a with NIL
>(setq z (funcall xorv a nil))
[161]

;;; Make the variable a true
> (assert! (equalv a t))
NIL

;;; Inspect z
> z
T
```

In terms of generality, the behaviour of the function returned by `(constraint-fn f)` is similar to that of `funcallv` applied to the same function: in both cases the result becomes bound as soon as all the arguments become bound. The difference is that con-

14. A *closure* is a specialised version of a function which accesses private variables (i.e., they are closed off to the outside world).

straint-fn returns a *functional entity* (technically, a 'closure'), instead of a constraint variable. This is a requirement for the development of higher-order constraint-based functions, to which a constraint-based *function*, and not a variable, must be supplied as an argument.

For details of the other higher-order functions defined by SCREAMER+, such as `everyv`, `at-leastv`, `exactlyv`, and `mapcarv`, see Appendix A.

5.3.3 Constraints on Objects

I used three basic principles relating to the combination of SCREAMER constraints and Common LISP objects. Firstly, SCREAMER constraint variables can represent either (predefined) *classes*[15], object *instances*, or *slot values* within an object instance. They should not contain slot definitions. Secondly, for simplicity, the constraint functions listed here do not define any new classes. The classes used for constraint-based reasoning should be defined in advance with the `defclass` construct. (You cannot `funcall #'defclass` anyway, because it is defined as a Common LISP macro.) Thirdly, the only constraint function which creates any new object instances is `make-instancev`. Note, however, that very often, programs do not need to use `make-instancev`, because it is possible to use the conventional form `make-instance`, instead. In such cases, constraint variables may fill the slot values of objects, but may not represent the objects themselves.

An anomaly can arise when dealing with CLOS objects and constraint variables together. Firstly, recall that CLOS objects may have slots which are unbound; that is, the slot contains no value at all, and an attempt to retrieve the value will normally result in an error. Likewise, constraint variables themselves may be unbound. In the case of constraint variables, this means that a structure has been created for maintaining the domain properties of the variable, but the variable has not yet been assigned a definite value. By combining constraint variables and CLOS objects, it is therefore possible to create a bound slot, containing an unbound constraint variable. That is, the Common LISP predicate (`slot-boundp` *obj slot-name*) would return `t`, whereas the SCREAM-

15. as would be returned by the Common LISP function `class-of`

ER expression `(bound? (slot-value` *obj slot-name*`))` would return `nil`. One possible approach to this problem is to ensure that the slot of any new class receives by default an `:initform` argument at creation time which fills the slot with an unbound constraint variable. The definition of a class would then resemble the following:

```
(defclass my-class ()
  ((slot-1 :accessor slot-1 :initform (make-variable)
    ...
   (slot-n :accessor slot-n :initform (make-variable))))
```

Then, when I create an instance of the class, the slots already contain constraint variables:

```
> (setq x (make-instance 'my-class))
#<MY-CLASS @ #x82d01a>
> (describe x)
#<MY-CLASS @ #x82d01a> is an instance of #<STANDARD-CLASS MY-
CLASS>:
 The following slots have :INSTANCE allocation:
  SLOT-N   [234]
  SLOT-1   [235]
> (slot-boundp x 'slot-1)
T
> (type-of (slot-value x 'slot-1))
SCREAMER::VARIABLE
> (bound? (slot-value x 'slot-1))
NIL
```

The integration of constraints and objects has real advantages in terms of the readability of code. Reconsider the method given for computing the area of a rectangle in Figure 33. A constraint-based version of this method is straightforward to write because it is very similar to the procedural version:

```
(defmethod areav ((obj rectangle))
  (*v (slot-valuev obj 'width)
      (slot-valuev obj 'height)))
```

Now suppose I said that a rectangle is *almost-square* if the difference between its width and height is exactly 1. A method to determine this property could be defined as follows:

```
(defmethod almost-squarev ((obj rectangle))
  (let (
        (diff (-v (slot-valuev obj 'height)
                  (slot-valuev obj 'width)))
        )
    (orv (=v diff -1) (=v diff 1))
    )
  )
```

If we now created a rectangle whose width and height were constrained to be integer values between 1 and 10, we could easily search for all *almost-squares* of those dimensions with an area of less than 50. The following function performs such a search:

```
(defun solve ()
  (let (
        (r (make-instance 'rectangle :x 0 :y 0
                          :width (an-integer-betweenv 1 10)
                          :height (an-integer-betweenv 1 10)))
       )

    (assert! (almost-squarev r))
    (assert! (<v (areav r) 50))

    (all-values (solution r (static-ordering #'linear-force)))
    )
  )
```

The example is intended to demonstrate that the object framework allows the programmer to abstract from the low-level detail of constraint variables to higher level notions, in which the individual constraint variables may be hidden. The function `solve` reasons about irregular shapes ("almost-squares") and their areas without having to explicitly construct a condition on the constraint variables (i.e., the width and height) of the rectangle object. The hiding of constraint variable details represents significant progress towards the expressiveness required for determining fitness for purpose. It allows the constraint programmer to concentrate on modelling properties of the outside world, instead of focussing on the conditions that must be upheld by the constraint variables.

5.3.4 Other SCREAMER+ Constraint Functions

Apart from the functions for manipulating arrays, `make-arrayv` and `arefv` (See Appendix A), there are two other new functions in SCREAMER+ which are of particular interest. Firstly, `formatv` provides a constraint-based version of formatted output. That is, it sends formatted output to a stream as soon as all of its arguments become bound. Since it usually spends most of its time waiting for its arguments to become bound, I think of the function as a kind of output **daemon**. Because constraint variables become temporarily bound during search for solutions, `formatv` can be usefully employed to trace the progress of a search. For example, suppose $x, y, z \in 1...5$ and we

would like a set of values of these variables for which $x^2 + y^2 = z^2$. To do this, the domains of the variable are first stated using `an-integer-betweenv`, the condition is asserted using SCREAMER's arithmetic constraint functions, and then the "formatting daemon" is introduced. Finally, the search for a solution using `one-value` activates the tracing mechanism for each visited leaf node of the search tree.

```
> (setq x (an-integer-betweenv 1 5))
[599 integer 1:5 enumerated-domain:(1 2 3 4 5)]
> (setq y (an-integer-betweenv 1 5))
[602 integer 1:5 enumerated-domain:(1 2 3 4 5)]
> (setq z (an-integer-betweenv 1 5))
[605 integer 1:5 enumerated-domain:(1 2 3 4 5)]
> (assert! (=v (*v z z) (+v (*v x x) (*v y y))))
NIL
> (formatv t "Trying x = ~d, y = ~d, z = ~d~%" x y z)
[612]
> (one-value (solution (list x y z) (static-ordering #'linear-
force)))
Trying x = 1, y = 1, z = 1
Trying x = 1, y = 1, z = 2
Trying x = 1, y = 2, z = 1
Trying x = 1, y = 2, z = 2
Trying x = 1, y = 2, z = 3
Trying x = 1, y = 2, z = 4
Trying x = 1, y = 2, z = 5
Trying x = 1, y = 3, z = 2
Trying x = 1, y = 3, z = 3
Trying x = 1, y = 3, z = 4
Trying x = 1, y = 3, z = 5
Trying x = 1, y = 4, z = 4
Trying x = 2, y = 1, z = 1
Trying x = 2, y = 1, z = 2
Trying x = 2, y = 1, z = 3
Trying x = 2, y = 1, z = 4
Trying x = 2, y = 1, z = 5
Trying x = 2, y = 2, z = 2
Trying x = 2, y = 2, z = 3
Trying x = 2, y = 2, z = 4
Trying x = 2, y = 3, z = 4
Trying x = 2, y = 4, z = 4
Trying x = 2, y = 4, z = 5
Trying x = 3, y = 1, z = 2
Trying x = 3, y = 1, z = 3
Trying x = 3, y = 1, z = 4
Trying x = 3, y = 1, z = 5
Trying x = 3, y = 2, z = 4
Trying x = 3, y = 3, z = 4
Trying x = 3, y = 4, z = 5
(3 4 5)
>
```

Another interesting feature of SCREAMER+ is the addition of **conditional constraints**, which are introduced with the macro `ifv`. The form of an `ifv` statement is as follows:

(ifv *boolean-constraint-variable value1* &optional (*value2* nil))

with the following semantics. If the constraint-variable becomes bound to true (T), then ifv returns *value1*; if it becomes bound to false (NIL), then *value2* (default value NIL) is returned instead. The critical point is that neither *value1* nor *value2* are evaluated until the boolean condition becomes bound. Using this mechanism, constraints can adopt a quiescent state and only be activated when they are actually needed. This can improve the efficiency of a solution, since an activated constraint implies a computational overhead (due to the condition checking and propagation carried out by noticers, or other similar daemons).

```
;;; p ⇒ q; p; THEREFORE q.
> (setq p (a-booleanv))
[11 Boolean]
> (setq q (a-booleanv))
[12 Boolean]
> (assert! (impliesv p q))
NIL
> (assert! p)
NIL
> q
T
```

```
;;; IFV p q; p; THEREFORE q.
> (setq p (a-booleanv))
[4 Boolean]
> (setq q (a-booleanv))
[5 Boolean]
> (assert! (ifv p q))
NIL
> (assert! p)
NIL
> q
T
```

Figure 35: Compares Forward Propagation of impliesv (Modus Ponens) with that of ifv

It is also interesting to compare and contrast the propagation properties of ifv with its logical counterpart, impliesv. The comparisons in figures 35 and 36 show that although both functions propagate “forwards” in the same way using *modus ponens*, only impliesv propagates “backwards” using *modus tollens*.

5.3.5 How SCREAMER+ Works

Some of the SCREAMER+ functions are written as combinations of functions which SCREAMER already provides. Consider impliesv, which is simply defined as:

```
(defun impliesv (p q)
  (orv (notv p) q))
```

```
;;; p ⇒ q; ¬ q; THEREFORE ¬ p
> (setq p (a-booleanv))
[7 Boolean]
> (setq q (a-booleanv))
[8 Boolean]
> (assert! (impliesv p q))
NIL
> (assert! (notv q))
NIL
> p
NIL
```

```
> (setq p (a-booleanv))
[10 Boolean]
> (setq q (a-booleanv))
[11 Boolean]
> (assert! (ifv p q))
NIL
> (assert! (notv q))
NIL
> p
[10 Boolean]  ; p still unknown
```

Figure 36: Contrasts Backward Propagation of `impliesv` (Modus Tollens) with that of `ifv`

Most SCREAMER+ definitions are not so simple, and involve the use of *noticers* to achieve the required results. By way of illustration I give below pseudo-code definitions of some SCREAMER+ functions in a LISP style.

carv

```
(defun carv (x)
  (if x_is_bound
    (car x) ; return value immediately
    (let ((z (make-variable))) ; create result variable
      (attach-noticer-to x such-that
        (if x_has_an_enumerated_domain
          (assert! z to_be_a_member_of
                   (mapcar #'car (enumerated-domain x)))
        (if x_becomes_bound
          (assert! z to_be (car x)))
        )
      (attach-noticer-to z such-that
         (if z_becomes_bound_before_x
           (assert! x to_be (consv z (make-variable)))
        )
      z) ; return value
     )
  )
```

`Carv` returns a value which is constrained to be the `car` of a cons *x*. It works by first checking whether *x* is already bound. If it is bound, then `(car x)` can be returned immediately as the value of the expression; otherwise, a constraint variable z is created, and noticers are attached to *x* and *z* to ensure consistency and perform bidirectional propagation. The constraint variable *z* is returned as the value of the function call, even though it might still be unbound when the function returns. It does not matter that *z* might be unbound, as it can still receive its value some time later by virtue of the noticers.

cdrv

```
(defun cdrv (x)
  (if x_is_bound
     (cdr x)  ; return value
    (let ((z (make-variable))) ; create variable to hold result
       (attach-noticer-to x such-that
         (if x_has_an_enumerated-domain
            (assert! z to_be_a_member_of
                    (mapcar #'cdr (enumerated-domain x)))
            )
         (if x_becomes_bound
            (assert! z to_be (cdr x))
            )
          )
       (attach-noticer-to z such-that
         (if z_becomes_bound_before_x
            (assert! x to_be (cons (make-variable) z))
            )
          )
       z)    ; return value
    )
  )
```

The pattern of the definition of `cdrv` is similar to that of `carv` — if the arguments to the function are bound, a conventional LISP function is used to immediately return the result. Otherwise, a constraint variable is created to hold the value of the expression, and noticers are attached to the variables of the constraint. For example, suppose *z* is constrained to be `(cdrv` *x*`)`, then it is important that *z* becomes bound to `(cdr` *x*`)` as soon as *x* becomes bound.

When a SCREAMER+ function is a constraint-based version of a Common LISP function, knowledge of the LISP function's behaviour can be used to increase the propagation power of the constraint-based counterpart. In the above example, the *noticers* on `cdrv` used this kind of knowledge to propagate from a bound value of *z* to a "mostly bound" value of *x*, and from the enumerated domain of *x* to the enumerated domain of *z*.

consv

```
(defun consv (x y)
  (if (and x_is_bound y_is_bound)
    (cons x y) ; return value
    (let ((z (make-variable)))
      ;; Noticer on z
      (attach-noticer-to z such-that
        (if z_has_an_enumerated_domain
          (assert! x to_be_a_member_of
                  (mapcar #'car (enumerated-domain z)))
          (assert! y to_be_a_member_of
```

```
                    (mapcar #'cdr (enumerated-domain z)))
          )
        (if z_becomes_bound
          (assert! x to_be (car z))
          (assert! y to_be (cdr z))
          )
        )
      ;; Noticer on x
      (attach-noticer-to x such-that
        (if (and x_is_bound y_has_an_enumerated_domain)
          (assert! z to_be_a_member_of
                   list_constructed_by_consing_x_to
                   each_of_the_enumerated_domain_of_y)
           )
        (if (and x_becomes_bound y_becomes_bound)
           (assert! z to_be (cons x y))
           )
         )
      ;; Noticer on y
      (attach-noticer-to y such-that
        (if y_has_an_enumerated_domain
          (if x_is_bound
            (assert! z to_be_a_member_of
                   list_constructed_by_consing_x_to
                   each_of_the_enumerated_domain_of_y)
          (if (and x_has_an_enumerated_domain
                   (< (* (domain-size x) (domain_size y))
                      *enumeration-limit*)
            (assert! z to_be_a_member_of
                     the-possible-conses ; could be several!
                     )
            )
          )
        (if (and y_is_bound x_has_an_enumerated_domain)
           (assert! z to_be_a_member_of
                    list_constructed_by_consing
                    each_of_the_enumerated_domain_of_x_to_y)
          )
        (if (and x_is_bound y_is_bound)
           (assert! z to_be (cons x y))
          )
        )
    z) ; return z
  )
)
```

`Consv` is a little more complex than either of `carv` or `cdrv` because it is a function of two arguments, instead of just one. If the arguments *x* and *y* are both bound at the time of function invocation, the result is returned immediately without creating any variables or attaching any noticers. Otherwise, a constraint variable is created and noticers are attached to each of the variables. The noticers ensure that if the value of any one of the variables can be computed, then it is computed and assigned to the appropriate variable. In addition, it makes a number of observations about the behaviour of variables

with enumerated domains. For example, if *z* = `(consv 'head y)`, and *y* is known to be either `'(tail)` or `'(rest)`, then the enumerated domain of *z* is inferred to be the pair of lists {`(head tail)`, `(head rest)`}:

```
> (setq y (make-variable))
[3]
> (setq z (consv 'head y))
[4]
> (assert! (memberv y '((tail) (rest))))
NIL
> z
[4 nonnumber enumerated-domain:((HEAD TAIL) (HEAD REST))]
```

slot-valuev

```
(defun slot-valuev (objvar slotname)
  (if (and objvar_is_bound (objvar_has_slot slotname))
    (slot-value objvar slotname) ; return req. value
    (let ((z (make-variable)))
      (attach-noticer-to objvar such-that
        (if (and objvar_becomes_bound
                 (objvar_has_slot slotname))
            (assert! z to_be (slot-value objvar slotname))
          )
        (if objvar_has_an_enumerated_domain
           (assert! z to_be_a_member_of
                    the_corresponding_slot_values_of_the_enume
rated_domain_of_objvar)
           )
        )
      ;; If the domain of z is reduced, the domain of
      ;; objvar is made consistent with the domain of z.
      (attach-noticer-to z such-that
        (if (and objvar_becomes_bound
                 (objvar_has_slot slotname))
            (assert! z to_be (slot-value objvar slotname))
          )
        )
     z) ; return z
```

`Slot-valuev` returns a value, *z*, constrained to be the value of the named slot of the given object. It demonstrates the use of the same pattern of noticer applied to CLOS objects and their slots.

ifv

```
(defmacro ifv (condition exp1 &optional (exp2 nil))
  `(let ((c ,condition))
    (assert! c to_be boolean)
    (if c_is_bound
       (if c_is_true
           ,exp1 ; evaluate exp1
           ,exp2 ; evaluate exp2
        )
```

```
       )
       (let ((z (make-variable)))
          (attach-noticer-to c such-that
            (if c_becomes_bound
               (if c_is_true
                  (assert! z to_be ,exp1)
                  (assert! z to_be ,exp2)
                   )
                )
            )
          z) ; return z
        )
     ) ; let
   )
```

`Ifv` is somewhat different to functions such as `carv` and `consv`, since it is necessarily defined as a macro. It should not evaluate the “then” or “else” parts of the statement until the condition becomes bound. The following test confirms that this is the case:

```
> (setq x (a-booleanv))
[5 Boolean]
> (ifv x (print 'yes) (print 'no))
[6]
> (assert! x)
YES
NIL
>
```

constraint-fn

```
(defun constraint-fn (f)
  (let* (
         (fn-name name-of-f)
         (constraint-fn-name fn-name-plus-letter-v)
         )
     (if constraint-fn-name is_a_known_function
        return the_known_definition_of_the_function
        return the_lambda_function
          #'(lambda(&rest args) (applyv f args))
        )
    )
  )
```

The current definition of `constraint-fn` assumes that if both the functions `foo` and `foov` are defined, then `foov` must be the existing constraint-based counterpart of the function `foo`. It makes this assumption because a predefined version of the constraint-based version is likely to have superior propagation properties than any version that can be automatically generated. For example, the SCREAMER+ function `carv` makes better use of enumerated domains than the semantically equivalent lambda function that uses `applyv` with the argument `#'car`. Thus, `(constraint-fn #'car)` re-

turns the SCREAMER+ function `carv`, instead of an inferior version using `applyv`. If there is no constraint-based counterpart already defined, then a version based on `applyv` is constructed and returned.

SCREAMER+ is backwards-compatible with SCREAMER, so a working SCREAMER program will also work under SCREAMER+. However, *SCREAMER+ is superior to SCREAMER because:*

- **SCREAMER+ has more general unification.** SCREAMER+ unification is based on the LISP predicate `equal` (which can be applied to sequences such as lists and strings), rather than on the more limited test `eq` (which can only be applied to LISP atoms).
- **SCREAMER+ has more powerful propagation mechanisms.** Most SCREAMER+ functions employ propagation mechanisms which make use of the semantics of their Common LISP counterpart.
- **SCREAMER+ introduces *conditional* constraints.** *Conditional constraints* provide the constraint programmer with *more control* over propagation, enabling the specification of constraints which are only triggered under certain circumstances. When used wisely, this mechanism can improve the efficiency of search.
- **SCREAMER+ has superior representation capabilities.** SCREAMER+ embraces more complex representations than LISP atoms, such as lists, strings, arrays, CLOS objects, and indeed *all* LISP types, *including those which are user-defined*.

With regard to the latter point, it is edifying to consider the limited type hierarchy[16] used by SCREAMER (see Figure 37). The type hierarchy demonstrates SCREAMER's emphasis on atomic values such as Booleans, integers and reals. Compound types such as lists, strings, and CLOS objects are simply labelled as being "non-numbers".

16. There are also three exceptional cases which are not shown in the figure: a *non-integer* which may or may not be a number, a *non-integer number* which may or may not be a real, and a *non-real* which may, or may not, be a number. I have not included them in the diagram because they are only "intermediate" types. That is, they reflect current knowledge of the variable, but a bound variable cannot be of that type. For example, a bound value cannot be a non-integer number without knowing whether it is real.

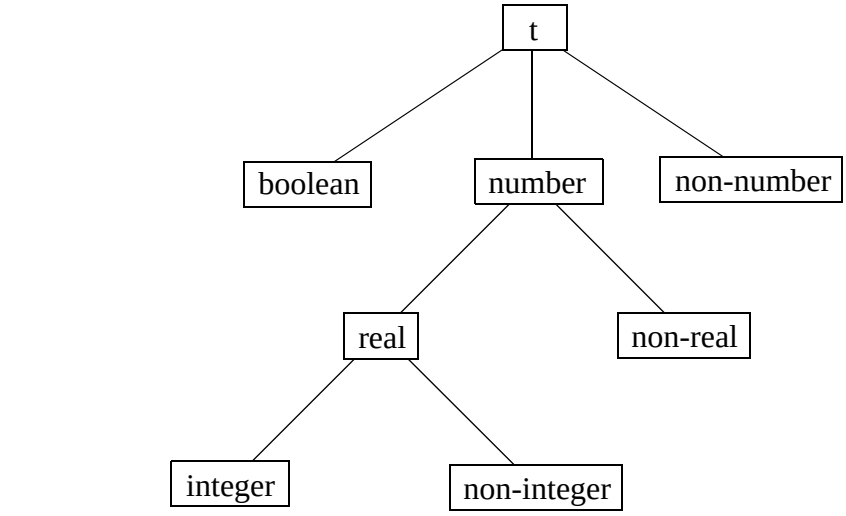


Figure 37: The Type Hierarchy of SCREAMER

In SCREAMER+, a variable which is a non-number can be constrained to be of *any type known to LISP*, such as a symbol, string, list, etc. For instance, suppose I create a variable, constrain it to contain a string, and then subsequently assert that it is one of the values `'e`, `'(b a r)`, or `"gum"`:

```
;;; *** SCREAMER+ Version ***
;;; Create a variable
> (setq x (make-variable))
[22]

;;; Assert that it is a string
> (assert! (stringpv x))
NIL

;;; Inspect x to see if it is known to be a string
> x
[22 string]

;;; Assert that x is either 'e, '(b a r), or "gum"
> (assert! (memberv x '(e (b a r) "gum")))
NIL

;;; x should be known to be "gum" since it is the only string
> x
"gum"
```

Contrast this with the behaviour of SCREAMER, which fails to (immediately) infer that the variable must be `"gum"` because it is the only string of the given domain:

```
;;; *** SCREAMER Version ***
;;; Create a variable
```

```
> (setq x (make-variable))
[130]

;;; Assert that it is a string. SCREAMER must use funcallv to do
;;; this since stringpv is part of SCREAMER+
> (assert! (funcallv #'stringp x))
NIL

;;; Inspect x
> x
[130]

;;; Assert x to be one of 'e, '(b a r), or "gum"
> (assert! (memberv x '(e (b a r) "gum")))
NIL

;;; Inspect x again
> x
[130 nonnumber enumerated-domain:(E (B A R) "gum")]
>
```

Furthermore, *user-defined* types are also acceptable. Suppose I wished to define a **palindrome** as a LISP type. A sequence *x* is a palindrome if *x* is the same as its reverse. So, for example, the string “noon” is a palindrome because reversing it also produces the string “noon”. In LISP, the type `palindrome` can be defined as follows:

```
(deftype palindrome ()
  '(and sequence (satisfies palindromep)))

(defun palindromep (x)
  (equal x (reverse x)))
```

Now, using SCREAMER+, I can constrain a variable to be a `palindrome`:

```
> (setq a (a-typed-varv 'palindrome))
[76 palindrome]
```

If I assert the variable to be one of the values `"hello"`, `'(p e e p)`, `"noon"`, or `'(1 2 3)`, then SCREAMER+ filters out those which are not `palindromes`:

```
> (assert! (memberv a '("hello" (p e e p) "noon" (1 2 3))))
NIL
> a
[76 palindrome enumerated-domain:("noon" (P E E P))]
>
```

5.4 Evaluation

I have evaluated SCREAMER+ by applying it to five benchmark constraint problems: the "Einstein" logic problem, the game Mastermind, a cross*number* puzzle, self-referential quiz, and car sequencing. The ability to solve these problems supports the belief that SCREAMER+ can compete with other constraint programming systems such as CHIP, ILOG Solver and Oz. Moreover, the ability to solve them in ways that are *concise*, *natural* (for a LISP programmer), and (reasonably) *efficient*, supports the belief that SCREAMER+ has something to offer. I believe that SCREAMER+ will be of interest to the constraints community and other AI researchers who program using LISP.

5.4.1 A Logic Problem

The following is a well-known logic problem, which I received as a problem-solving "challenge" from a friend. The problem concerns five houses, each of a different colour. Each house has one resident owner, whose nationality is unique among those owners. Each of the five owners drinks a certain type of beverage, smokes a certain brand of cigar and keeps a certain pet. However, no two of the five owners have the same pet, smoke the same brand of cigar or drink the same beverage.

The problem also provides the following additional information:

- the Briton lives in the red house;
- the Swede keeps dogs as pets;
- the Dane drinks tea;
- the green house is on the left of the white house;
- the owner of the green house drinks coffee;
- the person who smokes Pall Mall rears birds;
- the owner of the yellow house smokes Dunhill;
- the person living in the centre house drinks milk;
- the Norwegian lives in the first house;
- the person who smokes blends lives next to the one who keeps cats;
- the person who keeps horses lives next to the person who smokes Dunhill;
- the owner who smokes BlueMaster drinks beer;

- the German smokes Prince;
- the Norwegian lives next to the blue house;
- the person who smokes blends has a neighbour who drinks water.

The question is: *Who owns the fish?*

The task is to determine, for each of the five houses, the *position* (relative to the other houses), *colour*, *owner's nationality*, *owner's favourite beverage*, *owner's smoking habits*, and the *owner's pet*. Once these have been determined, it will be clear which of the owners keeps fish. A brief inspection of the additional information (and the question) reveals the domains for all of the variables in the problem. For example, the domain for each of the owner's *pet* variables would be {*dogs*, *birds*, *cats*, *horses*, *fish*}.

```
(use-package :screamer+)

(defclass house ()
  (
   (colour :accessor colour :initarg :colour)
   (nationality :accessor nationality :initarg :nationality)
   (beverage :accessor beverage :initarg :beverage)
   (cigar :accessor cigar :initarg :cigar)
   (pet :accessor pet :initarg :pet))
  (:default-initargs :colour (a-member-ofv '(blue red green white yellow))
    :nationality (a-member-ofv '(brit swede dane norwegian german))
    :beverage (a-member-ofv '(tea coffee milk beer water))
    :cigar (a-member-ofv '(dunhill blends bluemaster prince pall-mall))
    :pet (a-member-ofv '(dogs birds cats horses fish)))
  )

(defun solve ()
  (let (h1 h2 h3 h4 h5 houses blendy catty norwegiany bluey watery)
    ;; Create five different houses with the general constraints set
    (setq-domains (h1 h2 h3 h4 h5) (make-instance 'house))
    (setq houses (list h1 h2 h3 h4 h5))
    ;; Assert the uniqueness constraints
    (assert! (apply #'all-differentv (mapcar #'colour houses)))
    (assert! (apply #'all-differentv (mapcar #'nationality houses)))
    (assert! (apply #'all-differentv (mapcar #'beverage houses)))
    (assert! (apply #'all-differentv (mapcar #'cigar houses)))
    (assert! (apply #'all-differentv (mapcar #'pet houses)))
    ;; *     the man living in the centre house drinks milk
    (assert! (equalv (slot-valuev h3 'beverage) 'milk))
    ;; *     the Norwegian lives in the first house
    (assert! (equalv (slot-valuev h1 'nationality) 'norwegian))
    (setq-domains (blendy catty horsey dunhilly norwegiany bluey watery)
                  (an-integerv))
    (assert! (orv (=v blendy (-v catty 1)) (=v blendy (+v catty 1))))
    (assert! (orv (=v horsey (-v dunhilly 1)) (=v horsey (+v dunhilly 1))))
    (assert! (orv (=v norwegiany (-v bluey 1)) (=v norwegiany (+v bluey 1))))
    (assert! (orv (=v blendy (-v watery 1)) (=v blendy (+v watery 1))))
    (do* (
          (hs houses (cdr hs))
          (h (car hs) (car hs))
          (c 0 (+ c 1))
          )
         ((endp hs) t)
    ;; *     the Briton lives in the red house
      (assert! (equalv (eqv (slot-valuev h 'colour) 'red)
                       (eqv (slot-valuev h 'nationality) 'brit)))
    ;; *     the Swede keeps dogs as pets
      (assert! (equalv (eqv (slot-valuev h 'nationality) 'swede)
                       (eqv (slot-valuev h 'pet) 'dogs)))
    ;; *     the Dane drinks tea
      (assert! (equalv (eqv (slot-valuev h 'nationality) 'dane)
                       (eqv (slot-valuev h 'beverage) 'tea)))
    ;; *     the green house is on the left of the white house
      (when (> c 0)
        (assert! (impliesv (eqv (slot-valuev h 'colour) 'white)
                           (eqv (slot-valuev (nthv (- c 1) houses) 'colour) 'green)))
       )
    ;; *     the owner of the green house drinks coffee
      (assert! (equalv (eqv (slot-valuev h 'colour) 'green)
                       (eqv (slot-valuev h 'beverage) 'coffee)))
    ;; *     the person who smokes Pall Mall rears birds
      (assert! (equalv (eqv (slot-valuev h 'cigar) 'pall-mall)
                       (eqv (slot-valuev h 'pet) 'birds)))
    ;; *     the owner of the yellow house smokes Dunhill
      (assert! (equalv (eqv (slot-valuev h 'colour) 'yellow)
                       (eqv (slot-valuev h 'cigar) 'dunhill)))
```

Figure 38: A SCREAMER+ Program to Solve a Logic Problem

```
   ;; * the person who smokes blends lives next to the one who keeps cats
      (assert! (impliesv (eqv (slot-valuev h 'cigar) 'blends)
                         (=v blendy c)))
      (assert! (impliesv (eqv (slot-valuev h 'pet) 'cats)
                         (=v catty c)))
   ;; * the person who keeps horses lives next to the one who smokes Dunhill
      (assert! (impliesv (eqv (slot-valuev h 'cigar) 'dunhill)
                         (=v dunhilly c)))
      (assert! (impliesv (eqv (slot-valuev h 'pet) 'horses)
                         (=v horsey c)))
   ;; *     the owner who smokes BlueMaster drinks beer
      (assert! (equalv (eqv (slot-valuev h 'cigar) 'bluemaster)
                       (eqv (slot-valuev h 'beverage) 'beer)))
   ;; *     the German smokes Prince
      (assert! (equalv (eqv (slot-valuev h 'nationality) 'german)
                       (eqv (slot-valuev h 'cigar) 'prince)))

   ;; * the Norwegian lives next to the blue house
      (assert! (impliesv (eqv (slot-valuev h 'nationality) 'norwegian)
                         (=v norwegiany c)))
      (assert! (impliesv (eqv (slot-valuev h 'colour) 'blue)
                         (=v bluey c)))
   ;; * the person who smokes blend has a neighbour who drinks water
      (assert! (impliesv (eqv (slot-value h 'beverage) 'water)
                         (=v watery c)))
      (assert! (ifv (equalv (slot-value h 'beverage) 'water)
                    (=v watery c)
                    t))
      )

    (mapcar #'describe
            (one-value (solution houses (static-ordering #'linear-force))))
    )
  )
```

Figure 38 (cont/d): A SCREAMER+ Program to Solve a Logic Problem

To solve the problem using SCREAMER+, I decided to implement a solution in which each house is represented by a CLOS object, and the row of houses is represented by a list of five `house` objects. The `house` class contains a slot for each of the features of interest, other than position. This is a natural approach because the positions of the houses in the row are easily represented by the positions of the objects in the list, and the objects themselves reflect the structure of the problem by each containing the information known about a particular house. I believe it is easier to gain an overview of a solution that takes this object-oriented approach than one with a "flat" structure of 25 constraint variables. To set up the problem, constraints are asserted both across the slots of the five house objects ("inter-object constraints") and across the slots within the objects ("intra-object constraints").

My SCREAMER+ solution is shown in figure 38. When the program is run, the solution is output as follows:

```
#<HOUSE @ #x2067cb7a> is an instance of #<STANDARD-CLASS HOUSE>:
 The following slots have :INSTANCE allocation:
  COLOUR        YELLOW
  NATIONALITY   NORWEGIAN
  BEVERAGE      WATER
  CIGAR         DUNHILL
  PET           CATS
#<HOUSE @ #x2067ec0a> is an instance of #<STANDARD-CLASS HOUSE>:
 The following slots have :INSTANCE allocation:
  COLOUR        BLUE
  NATIONALITY   DANE
  BEVERAGE      TEA
  CIGAR         BLENDS
  PET           HORSES
#<HOUSE @ #x20680c9a> is an instance of #<STANDARD-CLASS HOUSE>:
 The following slots have :INSTANCE allocation:
  COLOUR        RED
  NATIONALITY   BRIT
  BEVERAGE      MILK
  CIGAR         PALL-MALL
  PET           BIRDS
#<HOUSE @ #x20682d2a> is an instance of #<STANDARD-CLASS HOUSE>:
 The following slots have :INSTANCE allocation:
  COLOUR        GREEN
  NATIONALITY   GERMAN
  BEVERAGE      COFFEE
  CIGAR         PRINCE
  PET           FISH
#<HOUSE @ #x20684dba> is an instance of #<STANDARD-CLASS HOUSE>:
 The following slots have :INSTANCE allocation:
  COLOUR        WHITE
  NATIONALITY   SWEDE
  BEVERAGE      BEER
  CIGAR         BLUEMASTER
  PET           DOGS
(NIL NIL NIL NIL NIL)
>
```

When running under Allegro Common LISP 5.0 on a 450MHz Sun SPARCstation, the program solves the problem in an average CPU time of 0.72 seconds[17].

5.4.2 The Mastermind Game

This example was introduced as a non-trivial programming problem by Sterling and Shapiro (Sterling & Shapiro, 1986), and has also been the topic of other articles (e.g., Van Hentenryck, 1989; Merelo, 1996). The problem concerns the game of Mastermind, in which the objective is for the code-breaker to crack the opponent's secret code

17. Or 1.95 seconds on a 133 MHz PC running the same version of Allegro Common LISP.

in as few ‘moves’ as possible. In this version, the code consists of an ordering of 4 differently coloured pegs, in which the pegs are known to be coloured either red, green, blue, yellow, white, or black. Each ‘move’ consists of the code-breaker guessing the code and the opponent truthfully providing two pieces of information: firstly, how many of the guessed pegs are correctly coloured, and secondly, how many of the guessed pegs are both correctly coloured *and* positioned. (In practice these two pieces of information are provided in the reverse order, since if all the pegs are both correctly coloured and positioned, then the code has been cracked.) The problem described here is to write a program which cracks the code by each time making a guess which is consistent with its current knowledge of the code. That is, it should always try to guess the right answer, rather than, say, ‘sacrificing’ a guess which it knows *not* to be correct in an attempt to find out about the colours and/or positioning of the pegs more quickly (an optimal strategy is of this type).

```
(defun mastermind ()
  (if (catch 'fail
        (let* (
               (colours '(red green blue yellow white black))
               (peg1 (a-member-ofv colours))
               (peg2 (a-member-ofv colours))
               (peg3 (a-member-ofv colours))
               (peg4 (a-member-ofv colours))
               (sol (list peg1 peg2 peg3 peg4))
               guess total-correct colour-correct
               )
          (assert! (all-differentv peg1 peg2 peg3 peg4))

          (loop
            (setq guess (one-value (solution sol (static-ordering #'linear-force))))
            (if (ith-value 1 (solution sol (static-ordering #'linear-force)) nil)
                (format t "My guess is ~s~%" guess)
              (progn
                (format t "The code is ~s~%" guess)
                (return t)
                )
              )
            (format t "How many are the right colour and correctly positioned ? ~%")
            (setq total-correct (read))
            (if (= total-correct (length sol))
                (return t)
              (assert! (notv (equalv sol guess)))
              )
            (assert! (exactlyv total-correct #'equalv guess sol))
            (format t "How many are the right colour ? ~%")
            (setq colour-correct (read))
            (assert! (=v (lengthv (intersectionv guess sol)) colour-correct))
                )
          )
        )
      t
    (format t "Your replies must have been inconsistent.~%")
     )
```

Figure 39: A SCREAMER+ Program to Play the Mastermind Game

A SCREAMER+ program to play mastermind in this way is given in Figure 39. The program uses the functions `all-differentv`, `exactlyv`, `lengthv`, and `intersectionv` from the SCREAMER+ library. Note that if the problem becomes insoluble because the information provided by the human operator is inconsistent, then the program aborts. Notice also that the program tests at each iteration to see if there is only one candidate solution left. If this is the case, the program reports the solution, and then halts.

An example session with the program is given below (the user's inputs have been underlined):

```
My guess is (RED GREEN BLUE YELLOW)
How many are the right colour and correctly positioned ?
1
How many are the right colour ?
2
My guess is (RED BLUE WHITE BLACK)
How many are the right colour and correctly positioned ?
1
How many are the right colour ?
3
My guess is (RED YELLOW BLACK WHITE)
How many are the right colour and correctly positioned ?
1
How many are the right colour ?
3
My guess is (WHITE BLUE BLACK YELLOW)
How many are the right colour and correctly positioned ?
1
How many are the right colour ?
4
The code is (WHITE YELLOW BLUE BLACK)
```

In an experiment in which the computer guessed ten different codes, it took the program an average of 4.2 guesses to crack the code. The average response time for each guess was 105 milliseconds when running under Franz Allegro Common LISP on a 133MHz PC.

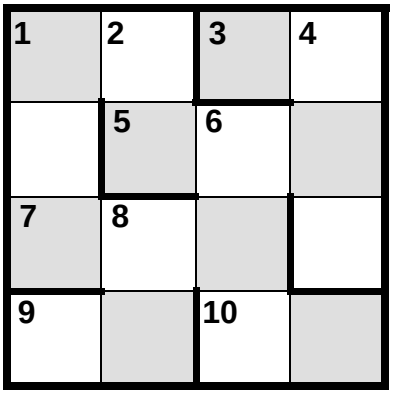
5.4.3 Crossnumber Puzzle

A cross*number* puzzle is similar to a crossword puzzle, but the solution is an array of interlinking numbers rather than words. Crossnumber puzzles are listed as **`prob021`** in the CSP benchmark library (Gent & Walsh, 1999). Figure 40 contains an example crossnumber puzzle which can be solved by SCREAMER+. The crossnumber clues

across and down provide constraints which must be satisfied by the solution. The structure of the solution grid also provides constraints, since the interlinking answers must be consistent.

Across

1. see 10 across
3. a prime
5. multiple of 3 across
7. multiple of 8 down
9. multiple of 2 down
10. a square - 1 across



Down

1. a square
2. see 9 across
4. 1 down + 6 down
6. a square
8. see 7 across

This crossnumber puzzle has digits of one parity (i.e., odd or even) on the shaded squares and digits of the opposite parity on the plain squares. No answer starts with a zero and multiples of a number exclude the number itself.

Figure 40: A Crossnumber Puzzle (from Mathematics Today, 34:2; April 1998); Solution is on page 155

From the figure, we know that the sum of 1-across and 10-across is a square number, whilst 1-down is itself a square. The second digit of 1-across is the first digit of 2-down, which is a divisor of 9-across, and so on. Note also the additional constraint that digits on the shaded squares are of one parity (i.e., odd or even), and digits on the plain squares are of the opposite parity, though the puzzle does not specify which is which.

It seems natural to represent the solution as an array in which each cell is constrained to be a digit. This is exactly the approach taken by the “core” program (given in Figure 41), and its supplementary functions (given in Figure 42). In the program, the function `get-numberv` returns a variable constrained to be the decimal number obtained by moving the given number of cells (down or across) from the given starting cell. This is important because it represents, in a general way, the relationship between the digits of individual cells and the numbers produced when the cells are combined.

```
(defun get-numberv (array direction p q length &optional (first t))
  (when first
    (assert! (notv (eqv (arefv array p q) 0)))
    )
  (if (= length 1)
      (arefv array p q)
    (if (equal direction 'across)
        (+v (*v (expt 10 (1- length)) (arefv array p q))
            (get-numberv array 'across p (1+ q) (1- length) nil))
      (+v (*v (expt 10 (1- length)) (arefv array p q))
          (get-numberv array 'down (1+ p) q (1- length) nil)))))

(defun crossnumber ()
  (let* ((x (make-arrayv '(4 4)))
         (dark-squares nil)
         (light-squares nil)
         (1across (get-numberv x 'across 0 0 2))
         (3across (get-numberv x 'across 0 2 2))
         (5across (get-numberv x 'across 1 1 3))
         (7across (get-numberv x 'across 2 0 3))
         (9across (get-numberv x 'across 3 0 2))
         (10across (get-numberv x 'across 3 2 2))
         (1down (get-numberv x 'down 0 0 3))
         (2down (get-numberv x 'down 0 1 2))
         (4down (get-numberv x 'down 0 3 3))
         (6down (get-numberv x 'down 1 2 3))
         (8down (get-numberv x 'down 2 1 2))
         some-square
        )
    ;; Constrain the array values to be digits & extract dark/light coords
    (dotimes (m 4)
      (dotimes (n 4)
        (assert! (integerpv (arefv x m n)))
        (assert! (>=v (arefv x m n) 0))
        (assert! (<=v (arefv x m n) 9))
        (if (oddp (+ m n))
            (push (list x m n) dark-squares)
          (push (list x m n) light-squares)
          )
        )
      )
    ;; Set up the chessboard parity
    (assert! (orv (apply #'andv
                         (append (mapcar #'even-cellpv dark-squares)
                                 (mapcar #'odd-cellpv light-squares)))
                  (apply #'andv
                         (append (mapcar #'odd-cellpv dark-squares)
                         (mapcar #'even-cellpv light-squares)))))
    ;; Now give the crossnumber clues
    (assert! (memberv 3across (primes 100)))
    (assert! (=v (funcallv #'mod 5across 3across) 0))
    (assert! (=v (funcallv #'mod 7across 8down) 0))
    (assert! (=v (funcallv #'mod 9across 2down) 0))
    (setq some-square (a-member-ofv (squares 200)))
    (assert! (=v 10across (-v some-square 1across)))
    (assert! (memberv 1down (squares 1000)))
    (assert! (memberv 6down (squares 1000)))
    (assert! (=v 4down (+v 1down 6down)))
    (one-value (solution x (static-ordering #'linear-force)))))
```

Figure 41: SCREAMER+ Crossnumber Puzzle Solver

The problem has a large solution space. Acknowledging the fact that none of the answers begins with a zero, there are sixteen variables, and $9^{10} \times 10^6 = 3486784401000000$ potential solutions to explore. The solution space is far too large for a simple backtracking approach to solve in reasonable time. However, my SCREAMER+ program solves the problem in an average CPU time of 230 milliseconds when running on a 450 MHz Sun SPARCStation running Franz Allegro LISP 5.0[18]

```
;;; Returns a list of square numbers between 1 and n

(defun squares (n)
  (do ((i 1 (1+ i))
       (squares nil))
      ((> (* i i) n) squares)
    (setq squares (nconc squares (list (* i i))))
    ) ; do
  ) ; defun

;;; Returns a list of primes between 1 and n using Sieve of Eratosthenes

(defun primes (n)
  (let ((candidates nil) (factor-limit (sqrt n)))
    (dotimes (c (1- n))
      (setq candidates (nconc candidates (list (+ 2 c))))
      )
    (do ((i 2 (+ i 1)))
        ((> i factor-limit) candidates)

      (when (member i candidates)
        (do ((x (* 2 i) (+ x i)))
            ((> x n) t)

          (setq candidates (delete x candidates)))
        ) ; when
      ) ; do
    ) ; let
  ) ; defun

;;; Constrain a number, or array cell, to be even/odd

(defun evenpv (n) (funcallv #'evenp n))
(defun even-cellpv (cell)
  (evenpv (arefv (first cell) (second cell) (third cell))))
(defun oddpv (n)(funcallv #'oddp n))
(defun odd-cellpv (cell)
  (oddpv (arefv (first cell) (second cell) (third cell))))
```

Figure 42: Supplementary Functions Used by the Crossnumber Puzzle Solver

5.4.4 Self Referential Quiz

A form of problem which can be particularly baffling is the **self-referential quiz** or SRQ (Fernández & Hill, 1997; see also `http://www.lcc.uma.cs/personal/fernandez_l/srq/index.html`). The term describes a set of multiple-choice questions in which the given options for each question may refer to answers elsewhere in the quiz. A simple two question SRQ is:

1. The answers to both of these questions are:
   (A) The Same (B) Different
2. The answers to both of these questions are:
   (A) Different (B) The Same

18. Or 693 milliseconds on a 133MHz PC running Allegro Common LISP 5.0.

If we assign the answer A to question 1, then A must also be the answer to question 2, but this contradicts the statement of question 2 that the two answers should be different. If we assign the answer B to question 1, however, then A must be the answer to question 2, and this is also consistent with the statement of question 2. The solution 1 (B), 2 (A) is unique within the search space of four possible assignments.

To each of the following questions, exactly one of the alternatives is true. Which one?

1. The first question whose answer is B is question
   (A) 2 (B) 3 (C) 4 (D) 5 (E) 6

2. The only two consecutive questions with identical answers are questions
   (A) 2 and 3 (B) 3 and 4 (C) 4 and 5 (D) 5 and 6 (E) 6 and 7

3. The last question with the same answer as this one is question
   (A) 10 (B) 9 (C) 8 (D) 7 (E) 6

4. The number of questions with the answer A is
   (A) 0 (B) 1 (C) 2 (D) 3 (E) 4

5. The answer to this question is the same as the answer to question
   (A) 10 (B) 9 (C) 8 (D) 7 (E) 6

6. The number of questions with answer A equals the number of questions with answer
   (A) B (B) C (C) D (D) E (E) none of the above

7. Alphabetically, the answer to this question and the answer to the following question are
   (A) 4 apart (B) 3 apart (C) 2 apart (D) 1 apart (E) the same

8. The number of questions whose answers are vowels is
   (A) 2 (B) 3 (C) 4 (D) 5 (E) 6

9. The number of questions whose answer is a consonant is
   (A) a prime (B) a factorial (C) a square (D) a cube (E) divisible by 5

10. The answer to this question is
   (A) A (B) B (C) C (D) D (E) E

Figure 43: The Small Self Referential Aptitude Test (Small SRAT)

A much more difficult self-referential quiz is shown in Figure 43, called the "Small Self-Referential Aptitude Test", or "Small-SRAT". It was devised by Martin Henz, now at the National University of Singapore[19], consists of ten questions each with five possible answers, and also has a unique solution. There are 5^{10} = 9765625 combinations of values to be explored. This is a much smaller search space than that of the crossnumber problem, but the self referential nature of the problem often means that it is very difficult to eliminate combinations of values without actually trying them.

Using SCREAMER+, it took a 450 MHz Sun SPARCStation[20] 1.03 seconds to find the solution to the small SRAT when running under Franz Allegro Common LISP 5.0. This compares with timings for the same machine of 0.05 seconds for ECLIPSE, and 29.7 seconds for a simple backtracking approach. Fernández and Hill reported timings on a 40MHz SPARCStation of 0.37 seconds for Oz, 0.85 seconds for ECLIPSE, and 18.1 seconds for Constraint Handling Rules (Fernández & Hill, 1997).

5.4.5 The Car Sequencing Problem

The **car sequencing problem** (Dincbas, Simonis & van Hentenryck, 1988; see also Tsang, 1993) is well-known within the constraints community. It is listed as **`prob001`** in the CSP benchmark library (Gent & Walsh, 1999). An implementation of a program to solve the car sequencing problem demonstrates some of the facilities of SCREAMER+.

Not all cars on a car manufacturing assembly line require the same set of options. For example, while some cars may require the installation of air conditioning, others may not. The same may be true of sunroofs, stereo equipment, and numerous other options. This can present difficulties for the production line, since the team of technicians responsible for the installation of a particular option cannot cope with a glut of cars on the assembly line which all require that option. To counter this difficulty, the assembly line is designed with a predetermined capacity ratio for each option. This ratio compares the maximum allowable number of cars *with some option* with a corresponding *total throughput* of cars in the same time. So, for example, a capacity ratio of 1 out of

19. See `http://www.comp.nus.edu.sg/~henz/`
20. It took a 133 MHz PC 3.01 seconds to run the same program.

```
(defun reifyv (test x)
  (if (ground? x)
      (if (funcall test x) 1 0)
    (let ((z (an-integer-betweenv 0 1)))
      (assert! (equalv z (funcallv #'(lambda(y) (if (funcall test y) 1 0)) x)))
      z)))
(defun how-manyv (test x)
  (apply #'+v (mapcar #'(lambda(y) (reifyv test y)) x)))
(defun srq ()
  (let* (q1 q2 q3 q4 q5 q6 q7 q8 q9 q10 sol alpha-diff
         count-a count-b count-c count-d count-e count-vowels count-consonants
         (pairs nil))
    (setq-domains (q1 q2 q3 q4 q5 q6 q7 q8 q9 q10) (a-member-ofv '(a b c d e)))
    (setq sol (list q1 q2 q3 q4 q5 q6 q7 q8 q9 q10))
    (assert! (memberv q1 '(a c d e)))
    (assert! (memberv q4 '(b c d e)))
    ;; Question 1
    (assert! (ifv (eqv q1 'a)
                  (eqv q2 'b)
                  (ifv (eqv q1 'c)
                       (andv (eqv q4 'b)
                             (notv (eqv q2 'b))
                             (notv (eqv q3 'b)))
                       (ifv (eqv q1 'd)
                            (andv (eqv q5 'b)
                                  (notv (eqv q2 'b))
                                  (notv (eqv q3 'b))
                                  (notv (eqv q4 'b)))
                            (andv (eqv q6 'b)
                                  (notv (eqv q2 'b))
                                  (notv (eqv q3 'b))
                                  (notv (eqv q4 'b))
                                  (notv (eqv q5 'b)))))))
    ;; Question 2
    (assert! (andv (notv (eqv q1 q2)) (notv (eqv q8 q9)) (notv (eqv q9 q10))))
    (dotimes (n (1- (length sol)))
      (setq pairs (cons (list (nth n sol) (nth (1+ n) sol)) pairs)))
    (assert! (=v 1 (apply #'+v
                          (mapcar #'(lambda(a) (ifv (eqv a t) 1 0))
                                  (mapcar #'(lambda(y) (eqv (first y) (second y)))
pairs)))))
    (assert! (ifv (eqv q2 'a)
                  (eqv q2 q3)
                  (ifv (eqv q2 'b)
                       (eqv q3 q4)
                       (ifv (eqv q2 'c)
                            (eqv q4 q5)
                            (ifv (eqv q2 'd)
                                 (eqv q5 q6)
                                 (eqv q6 q7))))))

    ;; Question 3
    (assert! (ifv (eqv q3 'a)
                  (eqv q10 'a)
                  (ifv (eqv q3 'b)
                       (andv (eqv 'b q9) (notv (eqv 'b q10)))
                       (ifv (eqv q3 'c)
                            (andv (eqv 'c q8)
                                  (notv (eqv 'c q10))
                                  (notv (eqv 'c q9)))
                            (ifv (eqv q3 'd)
                                 (andv (eqv 'd q7)
                                       (notv (eqv 'd q10))
                                       (notv (eqv 'd q9))
                                       (notv (eqv 'd q8)))
                                 (andv (eqv 'e q6)
                                       (notv (eqv 'e q10))
                                       (notv (eqv 'e q9))
                                       (notv (eqv 'e q8))
                                       (notv (eqv 'e q7)))))))))
    ;; Question 4
    (setq count-a (how-manyv #'(lambda(y) (eq y 'a)) sol))
    (assert! (memberv count-a '(1 2 3 4)))
    (assert! (ifv (eqv q4 'b)
                  (=v count-a 1)
                  (ifv (eqv q4 'c)
                       (=v count-a 2)
                       (ifv (eqv q4 'd)
                            (=v count-a 3)
                            (=v count-a 4))))))
```

Figure 44: Solver for the Small Self Referential Aptitude Test

```
;; Question 5
(assert! (ifv (eqv q5 'a)
              (eqv q10 'a)
              (ifv (eqv q5 'b)
                   (eqv q9 'b)
                   (ifv (eqv q5 'c)
                        (eqv q8 'c)
                        (ifv (eqv q5 'd)
                             (eqv q7 'd)
                             (eqv q6 'e))))))

;; Question 6
(setq count-b (how-manyv #'(lambda(y) (eq y 'b)) sol))
(setq count-c (how-manyv #'(lambda(y) (eq y 'c)) sol))
(setq count-d (how-manyv #'(lambda(y) (eq y 'd)) sol))
(setq count-e (how-manyv #'(lambda(y) (eq y 'e)) sol))
(assert! (ifv (eqv q6 'a)
              (=v count-a count-b)
              (ifv (eqv q6 'b)
                   (=v count-a count-c)
                   (ifv (eqv q6 'c)
                        (=v count-a count-d)
                        (ifv (eqv q6 'd)
                             (=v count-a count-e)
                             (andv (notv (=v count-a count-b))
                                   (notv (=v count-a count-c))
                                   (notv (=v count-a count-d))
                                   (notv (=v count-a count-e))))))))
;; Question 7
(setq alpha-diff (funcallv
                  #'(lambda(p q)
                      (abs (- (char-code (character (value-of p)))
                              (char-code (character (value-of q))))))
                  q7 q8))
(assert!
 (ifv (=v alpha-diff 0)
      (eqv q7 'e)
      (ifv (=v alpha-diff 1)
              (eqv q7 'd)
              (ifv (=v alpha-diff 2)
                   (eqv q7 'c)
                   (ifv (=v alpha-diff 3)
                        (eqv q7 'b)
                        (andv (=v alpha-diff 4) (eqv q7 'a)))))))
;; Question 8
(setq count-vowels (how-manyv #'(lambda(y) (member y '(a e))) sol))
(assert! (ifv (=v count-vowels 4)
              (eqv q8 'c)
              (ifv (=v count-vowels 3)
                   (eqv q8 'b)
                   (ifv (=v count-vowels 5)
                        (eqv q8 'd)
                        (ifv (=v count-vowels 2)
                             (eqv q8 'a)
                             (andv (=v count-vowels 6) (eqv q8 'e)))))))

;; Question 9
(setq count-consonants (how-manyv #'(lambda(y) (member y '(b c d))) sol))
(assert! (ifv (eqv q9 'a)
              (memberv count-consonants '(2 3 5 7))
              (ifv (eqv q9 'b)
                   (memberv count-consonants '(1 2 6))
                   (ifv (eqv q9 'c)
                        (memberv count-consonants '(1 4 9))
                        (ifv (eqv q9 'd)
                             (memberv count-consonants '(1 8))
                             (memberv count-consonants '(0 5 10)))))))
;; A surrogate constraint
(assert! (=v (+v count-vowels count-consonants) 10))
(one-value (solution sol
                     (reorder #'domain-size
                              #'(lambda(x) (declare (ignore x)) nil)
                              #'<
                              #'linear-force)))))
```

Figure 44 (cont/d): Solver for the Small Self Referential Aptitude Test

2 for sunroofs would indicate that no two consecutive cars should require a sunroof. Note that a ratio of 1 out of 2 (also written 1 / 2) is different from 2 out of 4, since the latter allows more flexibility in the ordering. The car sequencing problem, then, is to find an ordering for some given set of cars, with their different choices of options, such that the capacity constraints are satisfied.

An instance of the car sequencing problem is given in Table 7. In this problem there are ten cars and five different options. Since some of the cars required the same sets of options, they were subdivided into *types* of car[21]. This step reduces the size of the search space considerably, since we now assign a car type instead of a car to each of the 10 positions in the final ordering. Thus, the ten car problem has a search space size of 10^6 instead of 10^{10}, which would have arisen if the cars had not been clustered into types. The table also lists the capacity ratios for each of the options.

	type-1	**type-2**	**type-3**	**type-4**	**type-5**	**type-6**	**Capacity Ratio**
option-1	yes	no	no	no	yes	yes	1 / 2
option-2	no	no	yes	yes	no	yes	2 / 3
option-3	yes	no	no	no	yes	no	1 / 3
option-4	yes	yes	no	yes	no	no	2 / 5
option-5	no	no	yes	no	no	no	1 / 5
number of cars	1	1	2	2	2	2	-

Table 7: An Instance of the Car Sequencing Problem with 10 Cars

To solve the problem, I follow (Dincbas, Simonis, and van Hentenryck, 1988) by using four different sorts of constraints. (The sections of code corresponding to the assertions of each of these four types of constraint are labelled in the implementation in Figure 45.) Firstly, I constrain each of the ten elements in the solution to be one of the given car types (1). Secondly, I constrain the numbers of each car type appearing in the solution to match those given in the bottom row of Table 7 (2). Thirdly, I constrain every

21. These were called *classes* in (Dincbas, Simonis & Hentenryck, 1988), but the term *car type* has been introduced so that “car classes” can be more easily distinguished from the CLOS classes of the implementation.

possible sublist of the appropriate length to remain within the limits of the capacity ratio requirement (3). Finally, I use some additional constraints, called **surrogate constraints**, which are semantically redundant but nevertheless help to discover fruitless branches of the search tree more quickly (4). Let me illustrate the idea behind this last set of constraints with an example. There are three cars requiring option 3 (one of type 1, and two of type 5), which must all fit into the final ordering of ten cars. Since the capacity constraint for option 3 is 1 / 3, there cannot be more than 1 car with option 3 in the last 3 cars of the ordering. Therefore, one would also expect at least two cars with option 3 in the first seven of the ordering. I use the same reasoning to derive the constraint that the first four cars must contain at least one car with option 3. Similar constraints are generated for each of the options.

Notice that I have taken care to avoid any duplication of the problem instance information in the implementation. For example, the program itself computes the number of cars with a particular option, given the option choices for each of the car types and the number of cars of each type.

The implemented solution makes use of the SCREAMER+ functions `make-listv`, `constraint-fn`, `exactlyv`, `at-mostv`, and `at-leastv` resulting in an efficient, concise program solution. It finds all six solutions to the ten car sequencing problem in a CPU time of approximately 1.23 seconds on a 450MHz Sun SPARCstation running Franz Allegro LISP 5.0. This compares with approximately 0.25 seconds using CHIP 5 to reproduce the program described by Dincbas, Simonis and van Hentenryck on the same machine. The reason for this time discrepancy is threefold. Firstly, and most importantly, CHIP's constraint predicates are directly implemented in C, rather than the host language, PROLOG. This results in a considerable speed advantage, but also has the disadvantage that, unlike SCREAMER+, the constraint library is no longer portable across different host language implementations.

Secondly, my implementation is truly symbolic, in the sense that the requirements for options are left as boolean values (`t` or `nil`) in the program. Dincbas, Simonis and Hentenryck converted all their Boolean values to 1 or 0, so that when checking their capacity constraints, they could sum up these numbers and use a linear inequality on the sum. Checking a truly symbolic constraint is in general more time consuming than

```
(defclass car-type ()
  ((option-1 :initarg :o1)
   (option-2 :initarg :o2)
   (option-3 :initarg :o3)
   (option-4 :initarg :o4)
   (option-5 :initarg :o5)))

(defun sublists (n x)
  (do ((acc nil)
       (going x (cdr going)))
      ((< (length going) n) acc)
    (push (subseq going 0 n) acc)))

(defun surrogate (sequence cars-with-option which-option)
  (declare (special *capacities*))
  (do* ((capacity (cadr (assoc which-option *capacities*)))
        (capacity-size (caddr (assoc which-option *capacities*)))
        (countdown (length sequence) (- countdown capacity-size))
        (options-left cars-with-option (- options-left capacity))
        (fn (constraint-fn #'(lambda(x) (slot-value x which-option)))))
      ((< countdown 0) t)
    (assert! (at-leastv options-left fn (subseq sequence 0 countdown)))))

(defun assert-capacities (seq which-option)
  (declare (special *capacities*))
  (do* ((option-capacity (cdr (assoc which-option *capacities*)))
        (maxnum-in-sub (first option-capacity))
        (from-sequence (second option-capacity))
        (sub (sublists from-sequence seq) (cdr sub))
        (s (car sub) (car sub)))
      ((endp sub) t)
    (assert!
     (at-mostv maxnum-in-sub
               (constraint-fn #'(lambda(x) (slot-value x which-option))) s))))

(defun count-numbers (car-dist which-option)
  (do* ((count 0)
        (diminish car-dist (cdr diminish))
        (current (car diminish) (car diminish)))
      ((endp diminish) count)
    (when (slot-value (car current) which-option)
      (setq count (+ count (cdr current))))))

(defun solve ()
  (let (*capacities* car-types car-dist seq type1 type2 type3 type4 type5 type6)
    (declare (special *capacities*))
    (setq *capacities* '((option-1 1 2)
                         (option-2 2 3)
                         (option-3 1 3)
                         (option-4 2 5)
                         (option-5 1 5)))
    (setq
     type1 (make-instance 'car-type :o1 t   :o2 nil :o3 t   :o4 t   :o5 nil)
     type2 (make-instance 'car-type :o1 nil :o2 nil :o3 nil :o4 t   :o5 nil)
     type3 (make-instance 'car-type :o1 nil :o2 t   :o3 nil :o4 nil :o5 t)
     type4 (make-instance 'car-type :o1 nil :o2 t   :o3 nil :o4 t   :o5 nil)
     type5 (make-instance 'car-type :o1 t   :o2 nil :o3 t   :o4 nil :o5 nil)
     type6 (make-instance 'car-type :o1 t   :o2 t   :o3 nil :o4 nil :o5 nil))

    (setq car-types (list type1 type2 type3 type4 type5 type6))
    (setq car-dist (pairlis car-types '(1 1 2 2 2 2)))

    (setq seq (make-listv 10))            ; Sequence 10 cars
1   (dolist (s seq) (assert! (memberv s car-types)))
    (dolist (ty car-dist)              ; Assert distribution of car types
      (assert!
2      (exactlyv (cdr ty) #'eqv seq (make-list 10 :initial-element (car ty))))
      )
    (dolist (o *capacities*)
3     (assert-capacities seq (car o))
4     (surrogate seq (count-numbers car-dist (car o)) (car o)))
    (all-values (solution seq (static-ordering #'linear-force)))))
```

Figure 45: A SCREAMER+ Program for Solving an Instance of the Car Sequencing Problem with 10 Cars

checking a linear inequality, since we cannot make assumptions about the types of variables involved in the constraint. (For example, my constraint primitives use `equal` rather than, say, `=` or `eq`.) On the other hand, my approach is more flexible since I can easily change the program to cope with option variables which can take one of *many* values (instead of just the two possible values of a boolean variable), a modification which cannot be carried over easily to the use of linear inequalities.

Thirdly, my constraint predicates `at-leastv` and `at-mostv` are more flexible than those of CHIP. The CHIP predicate `atmost(N, LIST, VAL)` is an *assertion* that at most `N` elements of the *list of natural numbers* `LIST` have the value `VAL`. Recall that the SCREAMER+ predicate, `(at-mostv` n f x_1 x_2 ... x_n`)` takes a *predicate function* as an argument, together with the *multiple, symbolic list* arguments to which it should be applied, and returns a *boolean variable* indicating whether the predicate function is true of its arguments at most n times. Thus, my implementation is a higher-order function, not confined to constraining only a single list of numbers, which enables the programmer to assert the falsehood as well as the truth of the constraint by an appropriate binding of the output variable (e.g. `(assert! (notv (at-mostv ...)))`). The flexibility of my predicates also pays a price in terms of speed for this particular problem, but will gain in other situations where less flexible predicates prove to be insufficient.

Note also that my implementation has used object-orientation to model more closely the structure of the problem. The option information for each car type is held as part of the corresponding object. This can become a great advantage when the number of car types modelled increases significantly. In my implementation, I could extend the CLOS class structure of car-types to encompass an inheritance hierarchy with appropriate default values. This would provide a more natural summary of the information than extensive sets of lists containing car options, particularly if the list elements are to be referenced positionally.

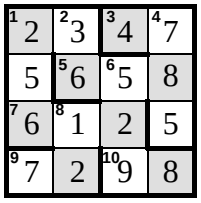
5.5 Summary

In this chapter, I first recalled the advantages of Common LISP as a programming language, and then described SCREAMER, a package that embeds nondeterminism and explicit constraint handling into Common LISP. I commented on my experiences of using SCREAMER, and reported some of the problems and surprises encountered. I also summarised the deficiencies of SCREAMER in handling symbolic constraints, and demonstrated how these deficiencies can be addressed by an extension to the SCREAMER library. I call the extended version of SCREAMER "SCREAMER+".

SCREAMER+ was evaluated by means of five well-known constraint problems. My solution to a well-known *logic problem* was expressed in a natural way using the object-oriented features of SCREAMER+. The *mastermind problem* demonstrated the use of constraints on lists interpreted as sets/bags. The *crossnumber puzzle* used the ability to assert constraints on arrays to arrive at its solution. The solution to the *self-referential quiz* took advantage of the facility for conditional constraints, and the solution to the car sequencing problem employed higher-order functions and used constrained CLOS objects for its representation. The first two problems are not so difficult —it should be possible to implement solutions to the "Einstein" logic problem and the Mastermind game in any useful constraint language. However, the other three problems are somewhat more difficult: the crossnumber puzzle is difficult because of the size of the search space and the nature of the constraints (particularly the constraint on the parity of the squares); the self-referential quiz is difficult because of its self-referential nature – this poses difficulties both in expressing the constraints and also solving the problem quickly. Car sequencing is a difficult problem to solve because of the nature of the constraints and the number of variables required. My object-oriented approach led to a more structured solution, and a program that could easily be reused for different instances of the problem. The biggest test for SCREAMER+, however, will be the ability to describe the fitness for purpose of a problem solver/knowledge base combination.

Solutions

The crossnumber problem given on page 143 has the following solution:



The Small Self-Referential Aptitude Test (see page 146) has the following solution:

1. C 2. D 3. E 4. B 5. E 6. E 7. D 8. C 9. B 10. A

Chapter 6

Determining Fitness for Purpose

'You cannot prove a theory, but you can disprove it.'

Karl Popper

Chapter Summary

This chapter discusses the problem of automatically recognising the fitness for purpose of a set of knowledge bases and a problem solver with respect to a particular problem solving goal. I propose the notion of *plausibility* as a tractable approach to this problem, and present a model of plausibility that takes the roles which knowledge bases play in problem solving into account, as well as the goal. I claim that a constraint-based approach is very suitable for the implementation of the model, and present problem-solving examples using the SCREAMER+ library discussed in the previous chapter.

6.1 Introduction

The automatic determination of fitness for purpose promises to be an important time-saving device for the users of a multistrategy toolbox such as MUSKRAT. Ideally, the toolbox supports the user by endorsing the choice of tool(s) which are suitable for solving the task at hand, and rejecting those which are not. Some established approaches to assessing whether a software component will solve the task at hand are by *design*, *testing*, *verification & validation*, *proving properties*, and *syntactic/structural matching with requirements*. I now briefly discuss each of these approaches.

By Design — A prevalent view in software engineering is that if the software development process is good, then the software developed by following that process will (automatically) be fit for its intended purpose. Formal specification methods such as VDM (Jones, 1986) and *Z* (Spivey, 1992) aim to assist program development by providing mathematically rigorous notations for the specification of system functionalities. It is argued that a formal specification is nearer to a working program than an informal specification, and the formalisation process raises issues which might not otherwise be discovered until quite late during development. Similarly-motivated specification formalisms exist within the knowledge acquisition community (e.g., CML, $(ML)^2$, KARL; see from page 37), and methods for operationalising such specifications have also been made (Wielinga, Akkermans & Schreiber, 1998). However, since my interest is in assessing the fitness for purpose of *existing* software, those technologies which assist in the *development* of the artefact cannot be applied.

By Testing — This method is used routinely and universally as part of the software development process. In this approach, a set of sample inputs are prepared as *test cases*. If no faults are found when the software is applied to these test cases, then the software is assumed to be fit for its intended purpose. Of course there is no guarantee of finding existing faults, as noted by Dijkstra:

> 'Program testing can be used to show the presence of bugs, but never to show their absence', Dijkstra[1], in (Backhouse, 1986).

Although it is usually impractical to test software exhaustively, the chance of finding faults can be improved by careful selection of test cases.

By Verification & Validation — *Verification and Validation* (V&V) seeks to discover whether a software system meets its users' requirements. Although the principles of V&V can be applied to any software artefact, practitioners are usually concerned with knowledge-based systems. Preece distinguishes the terms as follows:

> **'...verification** is the process of checking whether the software system meets the *specified* requirements of the users, while **validation** is the proc-

1. Compare this quote with the quote from Popper given on page 156.

> ess of checking whether the system meets the *actual* requirements of the users' (Preece, 1999).

Two approaches to V&V are **inspection** and **static verification**. *Inspection* is the human proof-reading of the system's source texts. This is a commonly employed technique, but often fails to discover faults in the system. *Static verification* seeks logical anomalies in the system's knowledge base(s), such as redundant or conflicting rules.

By Proving Properties — A potentially more informative approach is to *prove* the properties of a software component. When done by hand, this is a very difficult and involved process and is only really feasible for small programs. When automated, it can also be problematic. For example, on the subject of program verification, MacKenzie notes that

> 'wholly automatic theorem provers have so far been considered inadequate to these tasks' (MacKenzie, 1995).

In contrast, *interactive* theorem provers have the considerable asset of human guidance in their search for a proof. Unfortunately, to guide the search effectively, the user must often understand not only the proof being sought but also the mechanisms of the theorem prover. This requirement has meant that for the Boyer-Moore Interactive Theorem Prover (also called NQTHM):

> 'nearly all the successful users ... have in fact also taken a course from [Boyer and Moore] at the University of Texas at Austin on proving theorems in [their] logic' (MacKenzie, 1995).

By Syntactic/Structural Matching with Requirements — This approach views component selection as a problem of information retrieval. The methods search for structural or lexical similarities between the required component and the available components, identifying those which match best[2]. Unfortunately, this approach suffers from a typical problem of information retrieval – if the keyword vocabulary is not controlled, the most suitable component can easily be missed by the search. This is critical

2. A somewhat crude example of this approach is the Unix command `apropos` (equivalent to `man -k`), which returns one-line summaries of Unix commands whose descriptions contain the keyword input.

for the retrieval of software components, because users expect to retrieve only the best candidate components for reuse.

A structural matching approach has been adopted by the ROSA (Reuse Of Software Artefacts) project at the University of Geneva (Girardi & Ibrahim, 1995). This system uses heuristics to generate structural descriptions of software components' functionalities. When a user requests a component by submitting a query, the query is first translated to a structural functionality description, and this is compared to the stored component descriptions by using a similarity metric. The developers claim that the most appropriate components will produce the closest similarity match.

A different approach to the assessment of fitness for purpose supports the choice *among* software *libraries* by highlighting the similarities between their corresponding components (Michail & Notkin, 1999). Similarities are derived through syntactic matching of component names. This approach could not be applied to MUSKRAT, however, because MUSKRAT users must select a single tool(s) based on knowledge of the task at hand.

Other Approaches — As we have seen in chapter 4, the Consultant of the Machine Learning Toolbox made recommendations for machine learning algorithms by asking questions of the user, and applying a knowledge base to infer the most appropriate tool. The knowledge base contained heuristic knowledge which made the associations between the user's responses and the suitability of the tools.

Some state-of-the-art approaches to fitness for purpose *combine* aspects of other approaches. For example, Armengol et al. discuss combining the syntactic matching of components' inputs and outputs with a proof mechanism (Armengol et al., 1998). Following this approach, those components which do not have the same input–output characteristics as the required component are immediately rejected as being unsuitable, whilst the remaining components are subjected to further tests, supported by an automated (or interactive) theorem prover. This two stage process can be seen as a *filter* in which the most unsuitable candidates are easily removed.

6.2 The Plausibility Approximation

Chapter 4 pointed out the computational difficulties associated with recognising fitness for purpose, as well as MUSKRAT's requirement for the timely generation of advice. In order to satisfy both of these aspects, I introduce the notion of plausibility as a tractable approximation to fitness for purpose. This is later developed into a model which can be applied in a problem solving context.

6.2.1 Approximating Fitness for Purpose

The approximation of fitness for purpose focusses on the idea of a rejection filter which eliminates "bad" combinations of problem solving goal, problem solver, and available knowledge. A "bad" combination is one in which the problem solver, when using the available knowledge as its inputs, *cannot* output a value that satisfies the goal.

When considering such combinations, the (potential) outputs of a problem solver are classified as **plausible values**, **possible values**, or **actual values**. Informally, a *plausible value* is any value which cannot easily be disproved to be the output of the problem solver[3]. A *possible value* can be a solution to the problem in one or more cases. An *actual value* is the solution in a particular case. It is worth noting that all *possible* values are also *plausible*, and any *actual* value is always *possible* (see Figure 46).

As an example of this classification, consider the formula for the area of a circle, πr^2. Suppose that r, the radius, is known to be larger than 1, but no greater than 10, i.e., $r \in [1, 10]$ and $r^2 \in [1, 100]$. Now ranges of possible and plausible values can be defined for the circle's area. The range of possible values is $[\pi, 100\pi]$, but since π is rather a difficult number to deal with, the calculation of plausibility might approximate it to 3 for estimating the lower bound of the range, and 3.5 for estimating the upper bound of the range. The lower bound of the range is therefore $lower\_bound(r^2) \times 3 = 3$; and the upper bound is $upper\_bound(r^2) \times 3.5 = 350$. This gives the range of plausibility as [3, 350]. Note that we were careful to *underestimate* the lower bound of the plausibility range, and *overestimate* the upper bound, so that the range of possible values lies com-

3. Recall the example from page 94, in which 1097 was shown to be an *implausible* value of an arithmetic expression without doing the 'hard' work of calculating the true value.

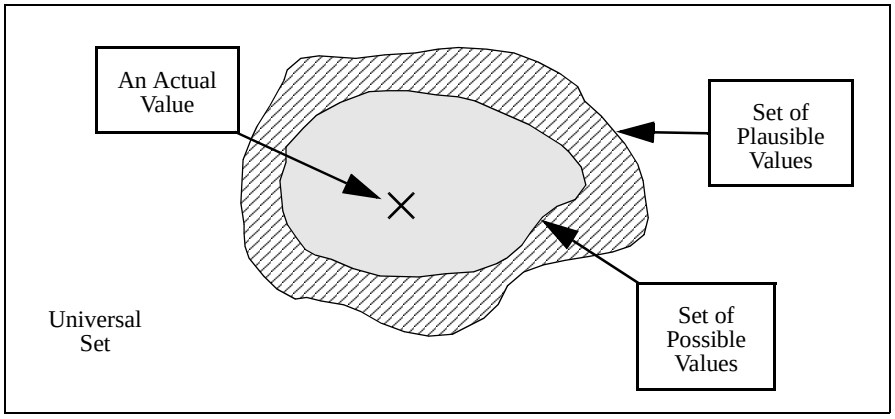


Figure 46: Venn Diagram showing the relationships between plausible, possible, and actual values

pletely within the range of plausible values (cf. Theory of Errors in the physical sciences). When evaluating the expression, an *actual value* might be 78.54.

The set of *plausible values* is an approximation to the set of *possible values*, but I define it such that

the set of possible values is always contained within the set of plausible values.

This guarantees that an implausible value is not possible, so the test for a possible value can be approximated by a test for plausibility. Note, however, that a *plausible* value may not necessarily be *possible.*

Using the terminology of machine learning (see “Inductive Learning” on page 23), we are interested in the *concept* of possibility. A value which is known to be possible is said to be a **positive** instance of that concept, and a value which is not possible is a **negative** instance. The plausibility approximation helps to classify values as either *positive* (i.e., possible) or *negative* (i.e., impossible). When the classification is correct, it is a **true** classification; otherwise it is **false**. Therefore in the context of my plausibility approximation, a **true positive** is a value correctly classified as possible, and a **true negative** is a value correctly classified as impossible. Similarly, a **false positive** is a value *incorrectly* classified as possible, and a **false negative** is a value *incorrectly* classified as impossible. The nature of the plausibility approximation allows *false positives* but excludes *false negatives.*

Note also that for any gain in computational effort, testing an element for membership of the set of plausible values must be less expensive than testing it for membership of the set of possible values.

6.2.2 Approximation in the Context of Problem Solving

A **problem solver** is a program which solves problems. Every problem solver is a program, but not every program is a problem solver. For example, a text editor, however useful it might be, is not normally considered to be a problem solver. Problem solvers, like other programs, have inputs and outputs. Problem solvers apply *knowledge* to solve problems, so I have called any input to a problem solver a **knowledge base**. My approach assumes that the problem solver runs as a batch, rather than an interactive, process. Note, however, that I have made no assumptions about the representation of the knowledge; when I say *knowledge base* I do not imply a set of rules. Nor have I made any assumptions about the specificity of knowledge bases with respect to the given task instance. That is, a knowledge base might remain constant over all task instances, it might change occasionally, or it might change with every task instance.

When problem solving, different inputs are used for different purposes. To avoid confusion, each input is usually assigned a **problem-solving role** (or simply **role**).

Definition 11: (Problem-solving) role[4]

A (problem-solving) *role* represents the way in which an input, such as a knowledge base or other knowledge source, is used for problem solving.

For instance, when an employer allocates a *job title* to an employee, the job title is a denotation of the employee's *role*. A job title is an example of a *role* because it specifies a person's duties within the organisation, often including the nature of any required interaction with other employees. Similarly, when a knowledge base (or other program input) is allocated a *role*, the role specifies how that input will be used by the problem

4. In theory, a role could also be used to denote a problem solver's output. However, I currently assume multiple inputs, and a single output value, so there is no need to distinguish between different outputs.

solver. Note that my notion of a *problem-solving role* corresponds very closely to the notion of a *knowledge role* in CommonKADS. CommonKADS defines *knowledge roles* as

> '...abstract names of data objects that indicate their role in the reasoning process' (Schreiber et al., 1999; p. 105).

The idea of approximating fitness for purpose can be applied to problem solving scenarios in which a number of knowledge bases (KB_1, KB_2, ..., KB_n) are assigned input roles (R_1, R_2, ..., R_n) to a problem solver, and a desired goal is stated. I call this a **problem solver configuration**.

Definition 12: Problem Solver Configuration

A (problem solver) *configuration* is the set of assignments of knowledge bases to the input roles of a given problem solver to solve a stated goal.

Since it is sometimes useful to refer to the assignment of knowledge bases to roles without considering the goal, I also define a **role configuration**.

Definition 13: Role Configuration

A *role configuration* is a set of assignments of knowledge bases to the input roles of a given problem solver.

A typical problem solver configuration is shown in Figure 47. It shows a single problem solver with multiple knowledge base inputs, and a single knowledge base output. The question of interest is whether the problem solver's output will satisfy the user's goal. The aim is to comment on this question without recourse to running the problem solver.

We can use the plausibility approximation to help make predictions about the success, or otherwise, of using a problem solver to solve a **goal**, which is defined as follows.

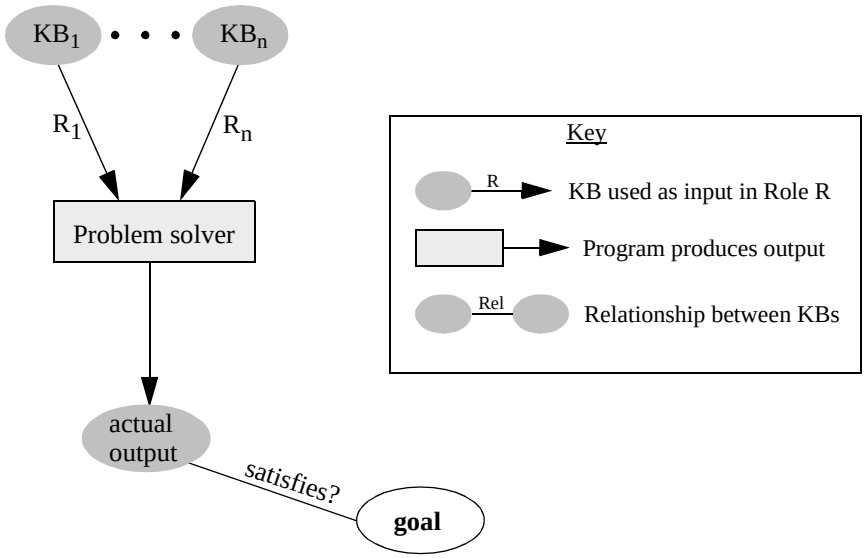


Figure 47: A Problem-solver Configuration

Definition 14: Problem-solving Goal

A *problem-solving goal* (or simply *goal*) is the set of acceptable output values from a problem solver. Successful problem solving is synonymous with satisfaction of the goal.

The reasoning used to support problem solving is based on the notion that a problem solving goal can only be satisfied if it contains at least one of the possible output values of the problem solver. Let us denote the set of acceptable output values by **goal**. If the problem solver *PS* uses the inputs $I = \{I_1, I_2, ... I_n\}$ in roles $R = \{R_1, R_2, ..., R_n\}$, then we denote the set of possible problem solver outputs by $\mathbf{possible}_{(PS, I, R)}$.

We then note the following three lemmas.

– The goal is only satisfiable if $(\exists p)\ \ (p \in \mathbf{possible}_{(PS,I,R)}) \wedge (p \in \mathbf{goal})$, or equivalently, $\mathbf{possible}_{(PS,I,R)} \cap \mathbf{goal} \neq \varnothing$. [1]

– If $\mathbf{possible}_{(PS,I,R)} \cap \mathbf{goal} = \varnothing$, then the problem solver *cannot* satisfy the goal. [2]

– If every possible output value satisfies the goal, i.e., $\mathbf{possible}_{(PS, I, R)} \subseteq \mathbf{goal}$, then the goal *must* be satisfied. [3]

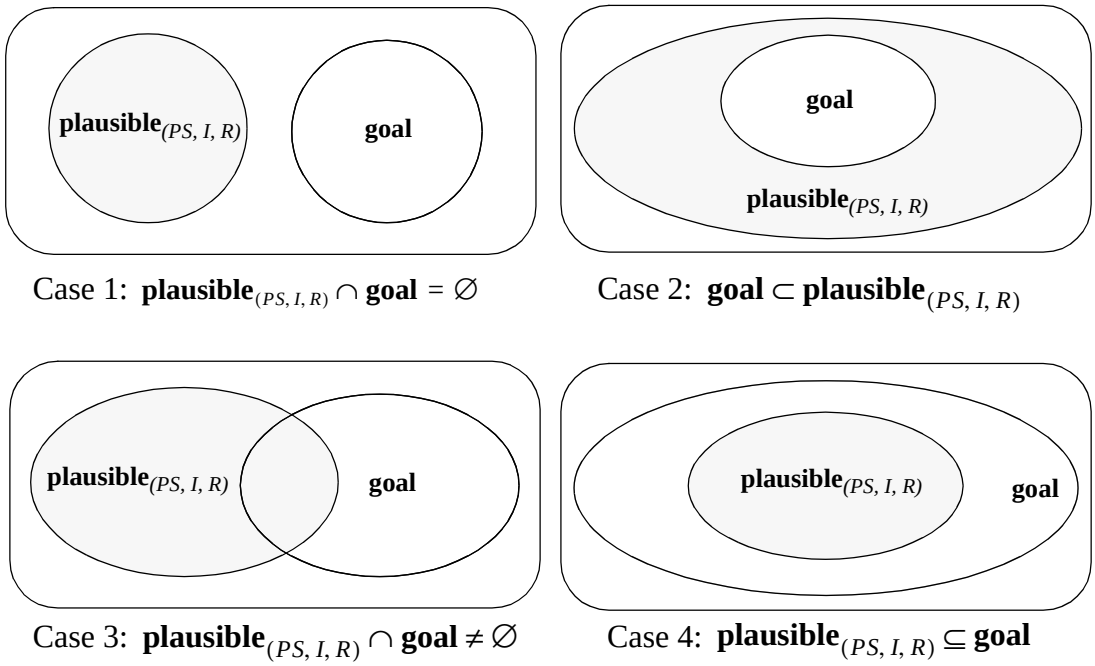


Figure 48: Inspecting the relationship between the problem solving goal and the plausibility approximation

The plausibility approximation to **possible**$_{(PS, I, R)}$ is the set of values which are plausibly output by the problem solver *PS*, given the inputs $I = \{I_1, I_2, ... I_n\}$ in roles $R = \{R_1, R_2, ..., R_n\}$. We denote this approximation by **plausible**$_{(PS, I, R)}$, where **plausible**$_{(PS,I,R)} \supseteq$ **possible**$_{(PS,I,R)}$. Let us now inspect the intersection **plausible**$_{(PS, I, R)} \cap$ **goal** , without additional knowledge of **possible**$_{(PS, I, R)}$.

There are four cases to consider (see figure 48 on page 165).

Case 1: Goal is <u>*unsatisfiable*</u> if **plausible**$_{(PS, I, R)} \cap$ **goal** $= \varnothing$. This follows because the plausible set of values contains the possible set of values (by definition), so

plausible$_{(PS, I, R)} \cap$ **goal** $= \varnothing \Rightarrow$ **possible**$_{(PS, I, R)} \cap$ **goal** $= \varnothing$.

It follows from [2] on page 164 that the goal is unsatisfiable.

Case 2: The goal is <u>*plausible*</u> if **goal** $\subset$ **plausible**$_{(PS, I, R)}$. Although **goal** $\subset$ **plausible**$_{(PS, I, R)}$, it does not necessarily follow that **goal** $\subset$ **possible**$_{(PS, I, R)}$. There are two cases to consider – either

$\mathbf{possible}_{(PS, I, R)} \cap \mathbf{goal} = \varnothing$, which would imply that the goal is unsatisfiable (from [2]), or $\mathbf{possible}_{(PS, I, R)} \cap \mathbf{goal} \neq \varnothing$, which leaves the goal plausible (from [1]). Without knowing the contents of $\mathbf{possible}_{(PS, I, R)}$, we do not know which of the two cases applies, so we must draw the weaker of the two conclusions, i.e., that the goal is plausible.

Case 3: Goal is *plausible* if $\mathbf{plausible}_{(PS, I, R)} \cap \mathbf{goal} \neq \varnothing$, with the same reasoning as for case 2.

Case 4: Goal is *satisfied* if $\mathbf{plausible}_{(PS, I, R)} \subseteq \mathbf{goal}$. If the plausibility approximation is completely contained within the goal, then all the possible problem solver outputs must also satisfy the goal. That is,

$\mathbf{plausible}_{(PS, I, R)} \subseteq \mathbf{goal} \Rightarrow \mathbf{possible}_{(PS, I, R)} \subseteq \mathbf{goal}$.

Therefore, from [3] on page 164, the goal must be satisfied.

6.2.3 Verifying Fitness for Purpose

The objective of verifying fitness for purpose is to assess whether a problem solver will plausibly satisfy a problem-solving goal. The activities associated with verifying fitness for purpose can be split into two main phases: a **goal-independent phase** and a **goal-dependent phase**.

The *goal-independent phase* captures a description of the problem solver in terms of the properties of its inputs and outputs. Acquiring such a description is a complex and time-consuming task, but it need only be performed once. Note that in my approach, the problem solver 'description' is itself a *program*. It implicitly describes the problem solver by generating its plausible output. I call such a description a **meta–problem-solver** (right hand side of Figure 49). The problem solver described by the meta–problem-solver is said to be at the **ground-level**.

The *goal-dependent phase* captures a description of the goal and tests its plausibility against the known properties of the problem solver's output. The idea is to determine whether the goal is consistent with the set of plausible outputs generated by the meta–problem-solver, because if the goal is *not* consistent with this set, then it will also not

be satisfied by the output of the problem solver itself. In such cases, the problem solver need not be run to test its outcome.

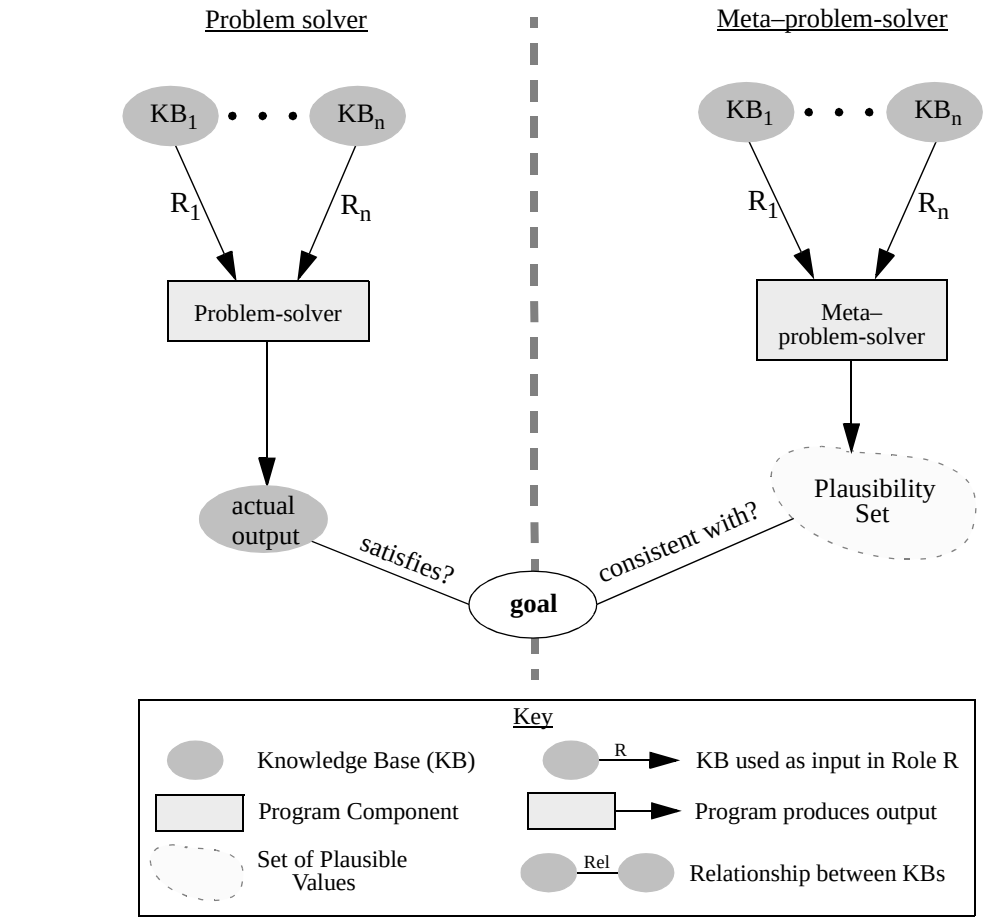


Figure 49: Problem solver and Meta–problem-solver Configurations

But how can this functionality be implemented? In a naive *generate-and-test* approach, one might answer the plausibility question by testing every point in the plausibility set against the goal. Unfortunately, this is both computationally inefficient (O'Hara & Shadbolt, 1996), and enables the modelling of sets of only finite size. To achieve more flexibility and efficiency, I prefer a *constraint-based* approach[5]. The idea is that a plausibility approximation is represented by a set of constrained entities. The set is described *intensionally* by its constraints, rather than *extensionally* by enumerating its

5. Consider the whimsical analogy of a man wishing to buy a pair of shoes: he does not walk into a bookshop, pick up every book, and inspect it for its shoe-like qualities (the generate-and-test approach). Instead, he recognises from the outset that bookshops sell books (i.e. the plausibility set is a set of books), and that this is not consistent with his goal of buying shoes.

members. In this way, we can deal with large (even infinite) sets of plausible values. A goal can also be represented by a constraint. To test the plausibility of the goal, the corresponding constraint is applied to the set of plausible values generated by a meta–problem-solver. If the additional constraint reduces the set of plausible values to the empty set, then the goal was not plausible; if the resulting set is non-empty (or cannot easily be demonstrated to be empty), then the goal remains plausible. Furthermore, unlike the meta-level reasoning required for the Halting Problem, the question of plausibility, if defined appropriately, can be guaranteed to terminate within a finite number of program steps, because the meta-level need only be an *approximation* to the problem solving level[6].

More general scenarios are also possible, because intensional descriptions of plausibility sets can also be used as *inputs* to further meta–problem-solvers. Such an input set could describe the plausible output of another problem solver, or even a knowledge acquisition tool. By cascading descriptions in this way, descriptions of plausibility sets are combined. Although less is known of the plausible output values when descriptions of inputs are used instead of concrete values, I believe that the output description thus generated may still contain enough knowledge to answer some interesting questions.

I mentioned earlier that a constraint-based approach to answering the question “Is the goal contained in the plausibility set?” is preferable. In fact, it is crucial to the implementation that constraints derived from the goal can be used to reduce the number of plausible values *without first enumerating them*. This technique makes the plausibility test less expensive than running the problem solver because the whole search space does not have to be explored to determine whether a goal is implausible. It also leads one naturally to consider an implementation using constraint programming. Indeed, one can rephrase the plausibility question in the terminology of constraint programming: “Is the goal condition *consistent with* the constraint network that represents the problem solver?”. As the tools I wished to model and the rest of the MUSKRAT framework were already implemented in Common LISP, I sought to implement the plausi-

6. Consider the case of the Halting Problem, which seeks to determine whether a program runs forever or halts. At the meta-level, one might choose to run the program for some given length of time to see what happens. If the program halts within the allotted time, then we know that it halts. Otherwise it remains plausible that it runs forever.

bility approximation to fitness for purpose using SCREAMER+, the constraint programming library described in chapter 5.

6.2.3.1 Building a Meta–Problem-Solver

To capture a description of a problem solver in terms of its inputs and outputs, I propose that the MUSKRAT developer[7] perform the following activities.

- Elicit the necessary properties of each of the problem solver's input roles, and express those properties as constraints.
- Elicit the properties necessary for inter-role consistency, and express those properties as constraints across the input roles.
- Elicit guaranteed properties of the problem solver's output(s), and express those properties as constraints. Often, properties of the output(s) are expressed in terms of the input roles.

These three sets of constraints form the basis of the meta–problem-solver. Note that the activities to derive these constraint are all human-oriented, because they involve interaction with a person who has detailed knowledge of the problem solver. The aim of the process is to collect constraints that capture detailed ("glass-box") knowledge of the problem solver. Once captured, this knowledge can subsequently be made available as "black box knowledge" to a user of the MUSKRAT toolbox. The acquisition of this set of constraints is a **once-only activity** because the problem solver description is also **goal-independent**. In other words, the same meta–problem-solver can be reused for the fitness-for-purpose verification of *different* tasks.

Necessary Properties of Input Roles

Before a knowledge base can be allocated to a role, the knowledge base must be checked against the *necessary properties* of that role. Failure to satisfy any one of those properties is sufficient grounds for rejection of the knowledge base for that role. On the other hand, if the knowledge base satisfies all of the properties, it is no *guarantee* that it will really help to solve the problem.

7. Also known as a MUSKRATEER.

Two ways of implementing the check for role satisfaction using SCREAMER+ are

- by using *typed constraint variables*, in which the necessary properties are part of the definition of the type;
- by defining CLOS classes whose instances are *automatically constrained* to possess the necessary properties (see the "class constraints" of ILOG Solver on page 65).

Typed Constraint Variables — Recall from chapter 5 that LISP allows the definition of types whose values must satisfy a given (arbitrarily complex) LISP predicate. SCREAMER+ adopted the same approach, so that types can also be attributed to a constraint variable by the satisfaction of an *arbitrarily complex* predicate (or constraint). This mechanism is well-suited to checking satisfaction of the necessary properties of a role. For example, suppose a problem solver requires as input an alphabetically sorted sequence of strings. A *type* for such a sequence can be defined as follows:

```
(deftype ordered-list ()
  '(and list (satisfies orderedp)))

;;; Generate successive pairs in a list
(defun pairs (x)
  (mapcon #'(lambda(y)
              (when (> (length y) 1)
                (list (subseq y 0 2))))
          x))

;;; Check that every successive pair is ordered

(defun orderedp (x)
  (every #'identity
         (mapcar #'(lambda(y)
                     (string-lessp (first y) (second y)))
                 (pairs x))))
```

To check for the satisfaction of role properties, a constraint variable is created for that role and a value is "assigned" (constrained to be equal) to the constraint variable. If the assignment fails, that value did not satisfy the properties of the role.

```
;;; Create a constraint variable (role) constrained to be an
;;; alphabetically ordered list of strings
> (setq my-ordered (a-typed-varv 'ordered-list))
[151 ordered-list]

;;; Assert that the variable has the value '("c" "a" "b")
> (assert! (equalv my-ordered '("c" "a" "b")))
```

```
Error: Attempt to throw to the non-existent tag FAIL
[condition type: CONTROL-ERROR]
Restart actions (select using :continue):
0: Return to Top Level (an "abort" restart)
[1] >
```

When there is more than one candidate for a role, you can assert that the value assigned to that role be one of those candidates. The consistency mechanism of SCREAMER+ will ensure that any of the candidates which do not satisfy the role will be pruned automatically.

```
;;; Create a constraint variable (role) constrained to be an
;;; alphabetically ordered list of strings
> (setq my-ordered (a-typed-varv 'ordered-list))
[153 ordered-list]

;;; Assert that the variable is one of the given lists
> (assert! (memberv my-ordered '(("a" "b" "c") ("fee" "fi" "fo"
"fum") ("c" "b" "a"))))
NIL

;;; Inspect the variable
> my-ordered
[153 ordered-list enumerated-domain:(("fee" "fi" "fo" "fum")
                                     ("a" "b" "c"))]
>
```

Automatically Constraining Object Instances — It is possible to automatically perform an arbitrary computation when creating an instance of a CLOS class. This mechanism can be used to assert constraints on the slots of a newly created object, such as the necessary properties for fulfilling a role. So, when the programmer creates an instance of a class with `make-instance`, a "pre-constrained" object is returned. The approach is implemented by using one of the standard LISP mechanisms for controlling object initialisation. The idea is to provide an appropriate definition for the standard CLOS method[8] `initialize-instance`. Then, whenever an instance of a class has been created, and before the object is returned to the programmer, the `initialize-instance` method is called, which asserts constraints as required. Constraints can also be inherited by subclasses.

As an example, consider a problem solving role that needs to be filled by a (CLOS) description of a boy. An implementation might first define a class to describe a `per-`

8. Strictly speaking, this is an *after* method, but I don't want to complicate the discussion by mentioning detailed ideas of object-orientation, such as *before*, *after*, and *around* methods. Consult (Steele, 1990) for a more detailed discussion on CLOS object initialisation.

son, then define a class called boy as a specialisation of the person class. A boy is a person, constrained to be male, and with an age of less than 18.

```
;;; This defines a class called person, with 3 slots - name
;;; sex and age. When creating an instance of the class,
;;; the integrity of the slots is constrained.
;;; The name is constrained to be a string, the sex is
;;; constrained to be either male or female, and the age
;;; is constrained to be an integer between 0 and 120.

(defclass person ()
  ((name :initarg :name :initform (a-stringv))
   (sex :initarg :sex :initform (a-member-ofv '(male female)))
   (age :initarg :age :initform (an-integer-betweenv 0 120))))

;;; The class boy is a specialisation of the class person
;;; The description has no more slots than a person, but
;;; the slot values are more restricted

(defclass boy (person)
  ())

;;; Assert constraints on the age and sex slots of a boy
;;; This method is called whenever an instance of a boy
;;; is created.

(defmethod initialize-instance :after ((b boy) &rest args)
  (assert! (equalv (slot-value b 'sex) 'male))
  (assert! (<v (slot-value b 'age) 18)))
```

Once a class has been suitably defined for the role, any given instance can be tested for conformance with that role. For example, I could create an instance of a 30 year-old person, and then attempt to reconcile that instance with the role of a boy (see description of reconcile on page 293). When this fails, it is confirmation that the instance of the person cannot fulfil the role of the boy.

```
> (setq role (make-instance 'boy))
#<BOY @ #x205e702a>
> (setq p (make-instance 'person :age 30))
#<PERSON @ #x205e42fa>
> (reconcile p role)
Error: Attempt to throw to the non-existent tag FAIL
  [condition type: CONTROL-ERROR]
Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart)
[1] >
```

Ensuring Inter-role Consistency

Inter-role consistency is concerned with the *mutual compatibility* of input roles. For example, suppose a problem solver requires two inputs: an ordered list of peoples'

names, and a correspondingly ordered list of their ages. Suppose further that the problem solver assumes the two lists to be of the same length. Then a constraint must be asserted to correspond to that assumption. The constraint is asserted across the constraint variables representing the input roles of the problem solver. For example, if the inputs to the meta–problem-solver were `x` and `y`, then we could `(assert! (=v (lengthv x) (lengthv y)))`.

In general, the problem of inter-role consistency is rather more complex than the previous example suggests. Consider what happens if a knowledge base is constrained to commit to a given ontology. How can this be checked? Whilst a *match* of terminology does not guarantee that the knowledge base is using the terms in the same sense, a *difference* in terminology *does* guarantee that the knowledge base *does not* commit to that ontology. Therefore a plausible approximation to ontological commitment is given by constraining the terminology of the knowledge base to be a subset of the terminology of the ontology. Checking the ontological commitments of a knowledge base is a current research topic (see, for example, Visser et al., 1997), and will not be discussed further here.

There are two main algorithmic alternatives for checking inter-role consistency: either it occurs after role assignment has been completed, or it is interleaved with role assignment. If it occurs after role assignment, then each assigned role must be checked (in the worst case) against every other role for consistency. Inconsistencies could be detected earlier if role consistency is interleaved with role assignment. In this case, a role would be provisionally assigned, then checked against all the other provisional assignments already made. Using this approach, failures would usually occur at an early stage, thus saving computation compared to checking the inter-role consistency of a full role configuration. I also suggest that a *fail-first strategy* might be adopted when performing role assignment; that is, the most constrained roles should be assigned values first in an attempt to minimise backtracking.

Guaranteed Properties of the Problem Solver's Output(s)

The guaranteed properties of the problem solver's output(s) are expressed as (declarative) constraints in the meta–problem-solver. These constraints are of two main types:

either they stipulate the *type* of the output; that is, that the output must satisfy some given, *arbitrary* predicate; or[9] they can express the relationship between the problem solver's inputs and outputs. They do not attempt to describe *how* the problem solver works.

Recall the problem of the Königsberg Bridges from chapter 4 (see from page 94). A naive ground-level problem solver might approach this problem by accepting a graph as its input, then using a brute-force search to find a path that traverses the graph (if one exists). A meta–problem-solver that incorporates Euler's conditions would be stating a relationship between the inputs and the outputs ("A path through the graph can only be found under the following conditions..."). Moreover, the meta–problem-solver would *add value* to such a ground-level problem solver, because it would indicate whether the problem is *solvable*. The meta–problem-solver would not *solve* the problem because evaluating Euler's conditions alone does not give the path that traverses the graph.

6.2.3.2 Testing a Goal

Before testing a goal, it is assumed that a problem solver has already been selected and that its corresponding meta–problem-solver is also available. Provided these conditions are met, the MUSKRAT toolbox should perform the following activities.

- Elicit the problem solving goal and express it as a constraint. The goal usually takes the form of a constraint on the output of the problem solver.
- First combine the constraint associated with the goal with the constraints of the meta–problem-solver, then check that the two components are not inconsistent. (For example, one way to combine them is to constrain the problem solver's output to be `equal` to a goal value. If the problem solver's output is demonstrably not the goal value — for example because it is of a different type — then the values are inconsistent and the goal is implausible.)
- If the input KB–role assignments (role configuration) are already known, then check that the role properties and inter-role properties are satisfied.

9. This is an inclusive 'or'; i.e., they can also do both.

- Propagate properties of the inputs through to the output, and check their consistency with the goal.
- Search for plausible KB–role assignments by attempting to solve the CSP[10]. This step assumes a closed world in which the required KB, if it exists, is also available.

Note that although these activities must be **repeated** for each new goal, they are well-defined and can be fully automated. As we shall see, each step lends itself to an implementation using constraint technology. Note also that role assignment and inter-role consistency checking also have the same *inclusive* properties as the plausibility approximation itself. That is, knowledge bases are only rejected from roles or (partial) role configurations when the goal can (demonstrably) not be satisfied. Notice that in general, there are many different input configurations, so the quest for a plausible one is a *search* process.

I now explain the goal-testing activities in more detail.

Eliciting the Goal

Eliciting the goal consists of two parts: firstly, the articulation of a condition on the problem solver's output; and secondly, the representation of that condition as a SCREAMER+ constraint. I believe that both of these steps can be automated provided that the composition of the goal can be anticipated (for example, with a grammar). However, this capability has not yet been demonstrated and I have therefore included it as a suggestion for further work in chapter 8 (see page 253).

In this chapter, I work directly with constraint expressions as goals, and assume that a tool can later be built to assist with the elicitation of goals and their formulation as SCREAMER+ constraints.

Combining the Goal with the Meta–problem-solver

As both the meta–problem-solver and the goal are represented as constraints, they both give rise to constraint networks. Furthermore, since the goal usually takes the form of

10. This is only possible if each constraint variable of the CSP has a finite domain. Typically, this means that there must be a known set of candidate KBs.

conditions on the output of the problem solver, when combining the goal with the meta–problem-solver, the constraint network associated with the goal is usually "appended" to the constraint network of the problem solver.

This can give rise to a constraint "failure" if the goal is inconsistent with the output of the problem solver. If that should occur, the problem solver will not be able to satisfy the goal *regardless of its inputs*.

6.3 Examples

6.3.1 Set Pruning

As an example of my approach, consider the *set pruning* method (Benjamins *et al.*, 1999; Groot, ten Teije & van Harmelen, 1999). It iterates over the members of a set, removing those members which do not exhibit a given property. It can therefore be regarded as a simple filtering method.

6.3.1.1 Problem Solver

A program with this functionality can be written in LISP as follows:

```
(defun set-prune (input-set test)
    (remove-if-not test input-set))
```

The set-pruning function can be demonstrated as follows:

```
;;; Create a set for testing (represented by a list)
> (setq my-set '(2 to 22 two "too"))
(2 TO 22 TWO "too")

;;; Prune away all the non-integers
> (set-prune my-set #'integerp)
(2 22)
```

6.3.1.2 Meta–problem-solver

The important issue here is not writing the problem solver itself, but the ability to describe it with a meta-program. I shall now develop a meta–problem-solver according to the guidelines set out in Section 6.2.3 (see from page 166). Later (in Section 6.3.1.3),

I demonstrate how the meta–problem-solver can be used to prove the implausibility of a problem solving goal.

Necessary Properties of the Input Roles

The set pruner accepts two inputs: a *set* to be pruned, and a *test* which each member of the output must satisfy. In my implementation, the set is represented by a list, and the test is represented by a function. A plausible LISP type for the role of a set can be defined as a synonym for the LISP type `list`:

```
(deftype plausible-set ()
  'list)
```

A plausible test can be any function or CLOS method of one argument[11]. This can be checked in Allegro Common LISP as follows:

```
;;; Predicate to check whether given function accepts a
;;; single argument
(defun one-arg-functionp (f)
  (let ((args (arglist f)))
    (cond
     ((= (length args) 1) t)
     ((member (car args) '(&rest &optional)) t)
     (t nil))))

;;; Defines a specialisation of the function type for
;;; functions of one argument
(deftype plausible-test ()
  '(and function (satisfies one-arg-functionp)))
```

Given these definitions, it is now possible to constrain an input to be a `plausible-set` or a `plausible-test`.

Properties Necessary for Inter-role Consistency

For inter-role consistency, we could check the compatibility between the *test* and the types of the members of the *set*. This is a condition of inter-role consistency because it could detect some error-causing problems, such as when the *test* is for a number being prime and the set contains some strings as well as numbers:

```
(defmethod primep ((n integer))
  (setq n (abs n))
  (when (= n 2) (return-from primep t))
```

11. Note that technically, CLOS methods are also classified as functions.

```
  (unless (evenp n)
    (do ((divisor 3 (+ divisor 2))) ; check odd numbers from 3
        ((> divisor (sqrt n)) t)   ; up to root of n
      (when (zerop (mod n divisor))
        (return-from primep nil)))))
;;; Assign test1 to be our implemented test for primeness
> (setq test1 #'primep)
#<STANDARD-GENERIC-FUNCTION PRIMEP>

;;; Assign set1 to be a plausible set which includes a string
> (setq set1 (list 0 5 "m"))
(0 5 "m")

;;; Demonstrate that errors can occur if the test cannot be
;;; applied to every member of the set
> (set-prune set1 test1)
Error: No methods applicable for generic function
    #<STANDARD-GENERIC-FUNCTION PRIMEP> with args ("m") of
classes (STRING)
  [condition type: PROGRAM-ERROR]
Restart actions (select using :continue):
 0: Try calling it again
 1: Return to Top Level (an "abort" restart)
[1c] >
```

However, a meta–problem-solver can recognise the condition of inter-role inconsistency that causes such an error. For the example of primeness, the idea can be realised by introducing a new function, called `testable`, which returns true (`T`) if the given test can be applied to a supplied argument, and false (`NIL`) otherwise.

```
(defun testable (input-set test)
  (when (not (equal (type-of test) 'standard-generic-function))
    (return-from testable t))
  (dolist (element input-set)
    (when (null (compute-applicable-methods test (list ele-
ment)))
      (return-from testable nil)))
  t ; returns true by default
  )
```

The function works by considering two cases. Firstly, if the test is a “true function” (i.e., *not* a CLOS method) then it is not checked further for its compatibility with the given arguments. If, however, the test is implemented as a generic function with one or more CLOS methods, then the function determines whether the generic function is applicable to each of the values in the given set[12]. That is, the function checks whether a method exists that accepts arguments of the same type as the elements of the supplied set. Note that in LISP, the generic function approach (using `defmethod`) allows for

stronger typing of input arguments than the weakly typed `defun` form. The generic function approach therefore provides good support for checking inter-role consistency.

Notice that `testable` has been implemented in a very general way, and does not encode any prior knowledge of the test. Furthermore, it will also be a cheap test to execute – certainly faster than executing the test itself. (We can be sure of this because we know that the LISP system must compute the applicable methods of a generic function with any given arguments before a method can be applied. Also, Since this is such an important part of LISP's run-time evaluation system, we can be confident that it has been optimised.)

Given the definition of `testable`, we can now check whether `test1` can safely be applied to each of the members of `set1`:

```
> (testable set1 test1)
NIL
```

We can also *assert* that `test1` be safely applied to all members of `set1`, as follows. Note that with the value of `set1` given above, the assertion fails, as expected:

```
> (assert! (funcallv #'testable set1 test1))
Error: Attempt to throw to the non-existent tag FAIL
  [condition type: CONTROL-ERROR]
Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart)
[1] >
```

Guaranteed Properties of the Problem Solver's Output(s)

When pruning a set, it is guaranteed that

- the output is of type `plausible-set`;
- the `output-set` is a subset of the `input-set`; and,
- every member of the `output-set` satisfies the given `test`.

These three statements are captured by the following three assertions[13]:

12. I also tried an alternative implementation which only computed the applicability of the test for each *distinct element type* in the given set. This was a promising approach because an input set may contain many elements of the same type, and the applicable methods would be computed once for each type instead of once per member of the set. However, my investigations led me to the conclusion that this approach was no faster than the given implementation – probably because determining the type of a LISP value is just as computationally expensive as computing a generic function's applicable methods.

```
(assert! (typepv output-set 'plausible-set))
(assert! (subsetpv output-set input-set))
(assert! (everyv (constraint-fn test) output-set))
```

By combining these constraints with the constraints given earlier for ensuring satisfaction of *role properties* and *inter-role consistency*, the following definition of the meta-pruner is derived:

```
(defun meta-prune (input-set test)
  (let ((output-set (make-variable)))
    ;;; Assert properties of input roles
    (assert! (typepv input-set 'plausible-set))
    (assert! (typepv test 'plausible-test))
    ;;; Assert inter-role consistency
    (assert! (funcallv #'testable input-set test))
    ;;; Assert known properties of the output
    (assert! (typepv output-set 'plausible-set))
    (assert! (subsetpv output-set input-set))
    (assert! (everyv (constraint-fn test) output-set))
    output-set) ; output-set is the return value
  )
```

The meta–problem-solver first generates a constraint variable to be used as its output, then asserts the necessary properties of its input roles. Next, it guarantees the inter-role consistency of its inputs by asserting that the supplied test must be applicable to each member of the input-set. Finally, the properties of the output are asserted; namely, that each member of the output-set is also a member of the input-set, and that the supplied test is satisfied by each member of the output-set. Note also that a meta–problem-solver must return the constraint variable which represents the plausible output value of the problem solver.

6.3.1.3 Goal

The meta–problem-solver is interesting because it enables reasoning about the output of the problem solver *without executing it*. In particular, many cases in which the problem solver would not satisfy the goal are detected early; for example even before the inputs to the problem solver are considered. I now describe several examples in which failures are detected by the meta-pruner:

13. Recall from chapter 5 that `constraint-fn` accepts a conventional LISP function as an argument, and returns a constraint-based function as its result.

Goal Incompatible with the Type of the Output-Set

In this example, the goal value, `5`, is not compatible with the type of the meta-pruner's output-set[14] (i.e., `5` is an atomic value, not a `plausible-set`):

```
> (setq plausible-output (meta-prune set1 #'integerp))
[63 plausible-set]

> (assert! (equalv plausible-output 5))
Error: Attempt to throw to the non-existent tag FAIL
  [condition type: CONTROL-ERROR]
Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart)
[1] >
```

Goal Incompatible with the Subset Constraint

In this example, the goal data structure `'(3)` is rejected because it is not a subset of the given input:

```
> (setq plausible-output (meta-prune set1 #'integerp))
[20 plausible-set]
> (assert! (equalv plausible-output '(3)))
Error: Attempt to throw to the non-existent tag FAIL
   [condition type: CONTROL-ERROR]
Restart actions (select using :continue):
0: Return to Top Level (an "abort" restart)
[1] >
```

Goal Incompatible with the Given Test

Here, the goal data structure `'("m")` is rejected because it is not a set of integers, as specified by the test.

```
;;; Generate the plausible output set of the PSM
> (setq plausible-output (meta-prune set1 #'integerp))
[96]

;;; Compare a "goal" data structure with the plausible output
> (assert! (equalv plausible-output '("m") ))
>> Error: Attempt to throw to the non-existent tag FAIL
   [condition type: CONTROL-ERROR]
Restart actions (select using :continue):
0: Return to Top Level (an "abort" restart)
[1] >
```

14. The goal value is also incompatible with the subset constraint, but the assertion fails even without that constraint.

Input-Set Incompatible with the Required Type

In this example, the argument `5`, representing the `input-set`, is incompatible with the expected type `plausible-set`:

```
> (meta-prune 5 #'integerp)
Error: Attempt to throw to the non-existent tag FAIL
  [condition type: CONTROL-ERROR]
Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart)
[1] >
```

Supplied Test Incompatible with the Required Type

Here the argument `'not-a-function`, representing the `test`, is incompatible with the expected type `plausible-test`:

```
> (meta-prune '(0 5 "m") 'not-a-function)
Error: Attempt to throw to the non-existent tag FAIL
  [condition type: CONTROL-ERROR]
Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart)
[1] >
```

Similarly, the argument #'< is also not a plausible-test, because it is not a function of a single argument:

```
> (meta-prune '(0 5 "m") #'<)
Error: Attempt to throw to the non-existent tag FAIL
  [condition type: CONTROL-ERROR]
Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart)
[1] >
```

Inter-role *Inconsistency*

Here, the supplied test (for whether an integer is prime) cannot meaningfully be applied to all the members of the input-set:

```
> (meta-prune '(0 5 "m") #'primep)
Error: Attempt to throw to the non-existent tag FAIL
  [condition type: CONTROL-ERROR]
Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart)
[1] >
```

Searching for Plausible Configurations

Suppose that several input-sets were available, as well as a single goal. Using SCREAMER+, together with the meta-pruner, it is possible to discover which of the input-sets plausibly satisfy the goal. For example, suppose that the input-sets were {}, {0, 1, 2}, {A, B, C}, {0 A}; and that the test prunes away items that are not integers. Suppose further that the goal is to obtain the set {0}.

Then the problem is first set up as follows:

```
> (setq kbs (a-member-ofv '(() (0 1 2) (a b c) (0 a))))
[69 nonnumber enumerated-domain:(NIL (0 1 2) (A B C) (0 A))]
> (setq goal '(0))
(0)
> (assert! (equalv (meta-prune kbs #'integerp) goal))
NIL
```

Notice that no failures have been generated so far, indicating that the task instance is still plausible. The following SCREAMER search idiom returns all input-sets that *plausibly* satisfy the goal:

```
> (all-values (solution kbs (static-ordering #'linear-force)))
((0 1 2) (0 A))
```

In effect, the MUSKRAT method is recommending to try the input-sets {0 1 2} and {0 A} to satisfy the goal. Notice that {0 1 2} does not satisfy the goal, but {0 A} does:

```
> (equal (set-prune '(0 1 2) #'integerp) goal)
NIL
> (equal (set-prune '(0 A) #'integerp) goal)
T
```

6.3.2 Meal Design and Scheduling

This section investigates a problem solving scenario in the domain of meal preparation. The domain was chosen not only because it is rich and challenging, but also because examples can be introduced easily without having to explain the domain terminology. This choice of domain is not unique. For example, Hammond describes a case-based planner called CHEF which outputs a recipe which satisfies some given criteria (Hammond, 1993), and Hinrichs describes a system called JULIA which designs dinner parties to satisfy the food preferences of a group of guests (Hinrichs, 1992).

6.3.2.1 Problem Solvers

The scenario applies two distinct problem solvers: a *designer* and a *scheduler*. The designer constructs solutions (in this case, dishes) from the available components (ingredients) which are consistent with some given design constraints. The scheduler assigns time slots to tasks and subtasks carried out in the production of a given dish by using knowledge of their durations and resource dependencies, as well as knowledge of the resources which are available in the kitchen. Although the problem solvers can be used independently, they can also be cascaded such that the output from the designer is used as input to the scheduler. The combined effect of the two problem solvers is to transform the design requirements into a specific design, together with the preparation tasks for making the artefact. The scenario is thus analogous to automated design and production scheduling.

CKRL Entity maps to →	**LISP Entity**
Concept	CLOS class
Property	CLOS slot
Instance	CLOS object / class instance
Sort	LISP type
Relation	LISP list
Fact	LISP list
Rule	LISP function

Table 8: Mapping CKRL entities to Common LISP

The knowledge used as inputs to both of the problem solvers is represented using CKRL (see chapter 4). Recall that CKRL is a high-level *interchange format*, not a programming language. Its constructs are therefore not directly executable, but must be first translated into executable code in the native language of the problem solver. As the native language for this scenario is Common LISP (the implementation language of the designer and the scheduler), I have written a translator which automatically maps CKRL constructs to Common LISP expressions. The nature of this mapping is sum-

marised by Table 8, and detailed in Appendix B. Note that a translator from LISP to CKRL is not needed, as CKRL outputs can be produced directly by problem solvers.

Designer

The design process is performed in two stages: the first phase satisfies hard constraints on the design, the second phase applies preferences to a set of proposed solutions in order to select the 'best' design.

The first phase applies a simple *generate-and-test* strategy to:

- select from the known components;
- assemble those components together to form a composite object; and,
- test that object against the constraints to see if it constitutes a valid design.

The second phase selects one of the proposed designs by:

- applying a linear evaluation function to each proposed design;
- ordering the proposed designs according to this evaluation; and,
- selecting the design at the head of the ordering (i.e., the design for which the evaluation function is greatest).

In the culinary domain, the *components* are the ingredients of dishes, and a design is a complete dish. Table 9 contains the component description knowledge used in this scenario. The terminology of dish design is provided by a dish ontology which defines different components of a dish; in this example, a `dish` consists of a `main-component` (e.g., gammon steak), together with a `carbohydrate-component` (e.g., french fries) and a `vegetable-component` (e.g., peas). The designer also calculates the `cost` of a dish as the sum of the costs of its component parts[15], and the number of `dietary-points` associated with a dish as the sum of the dietary points of its components.

15. The costs associated with the actual *construction* of the design from its constituent parts can be carried by the components. However, this is unlikely to be a good policy when different *designs* cost very different amounts. Such discrepancies can arise, for example, when one design requires significantly more work for its construction than other designs using the same components.

Ingredient Name	Ingredient Type	Dietary points[a]	Cost (£)
Lamb Chop	Main (meat)	7	4
Gammon Steak	Main (meat)	10	4
Fillet of Plaice	Main (fish)	8	4
Sausages	Main (meat)	8	2
Chicken Kiev	Main (meat)	8	3
Quorn Taco	Main (vegetarian)	6	3
Jacket Potato	Carbohydrate	5	1
French Fries	Carbohydrate	6	1
Rice	Carbohydrate	6	1
Pasta	Carbohydrate	2	1
Runner Beans	Vegetable	1	1
Carrots	Vegetable	1	1
Cauliflower	Vegetable	1	1
Peas	Vegetable	2	1
Sweetcorn	Vegetable	2	1

a. Points schemes like these are commonly used by slimmers as a simplification of calorific value. When using such schemes, slimmers allow themselves to consume no more than, say, 40 points worth of food in a single day (the actual limit usually depends on the slimmer's sex, height, and weight).

Table 9: Some Meal Preparation Knowledge used by the Designer

Scheduler

Given the 'design' of a particular dish, as well as recipe knowledge for the preparation of its components and knowledge of the available resources, the scheduler assigns start times to each of the tasks and subtasks needed for its preparation.

In the following example, a schedule is generated for the preparation of a dish consisting of lamb chop, pasta and runner beans. The main task to be scheduled is assumed to have the name `root-task`. For the given meal, this top-level task is divided into the subtasks `prepare-lamb-chop`, `prepare-pasta` and `prepare-runner-beans`. Subtasks are recursively decomposed until the resources needed by the task are clearly defined. For example to prepare runner beans, water is first boiled in a `kettle`, then transferred to a `saucepan` on a `hob-ring`, where the beans are added and then

cooked. `Kettle`, `saucepan` and `hob-ring` are all resources which are in limited supply in the kitchen. A similar procedure is used for preparing the pasta.

The schedule generated is as follows:

```
           ROOT-TASK:0**************************************20
      PREPARE-PASTA:0******************10
     BOIL-BEAN-WATER:      3*****6
          COOK-BEANS:             6************13
PREPARE-RUNNER-BEANS:      3******************13
          COOK-PASTA:      3*************10
    BOIL-PASTA-WATER:0*****3
     GRILL-LAMB-CHOP:           5****************************20
       PREHEAT-GRILL:0*********5
   PREPARE-LAMB-CHOP:0**************************************20
```

The numbers in the schedule denote the number of minutes elapsed since the start of the `root-task`. Notice that the task `boil-bean-water` does not overlap with the task `boil-pasta-water` because both tasks require full use of the only available `kettle`.

6.3.2.2 Meta–problem-solvers

This section provides an example of meta-level reasoning applied to the designer and scheduler introduced in the previous section. Given some simple culinary knowledge, I shall demonstrate the meta-level reasoning used to infer that there is no such thing as a “quick” dish that includes a jacket potato, nor a “healthy” dish that includes gammon and french fries.

Meta-designer

The inputs and outputs of the designer and meta-designer are shown in figure 50. Although the meta-designer has *access* to all of the knowledge bases used by the designer, my implementation of the meta-designer makes no reference to the *design preferences* or *design constraints*. These two knowledge bases have therefore been ‘greyed out’ in the figure, and the knowledge they contain is not used to test the plausibility of the goal. The idea is that the implementer of the meta–problem-solver asserts truths about the output of the ground-level problem solver, but can also choose the *degree* to which the input knowledge is used. There is a trade-off between the reasoning power given by asserting all that is known about the problem solver’s output, and

the cost of computing that knowledge for a given task instance. In the limit, the meta–problem-solver is simply a new (constraint-based) implementation of the problem solver.

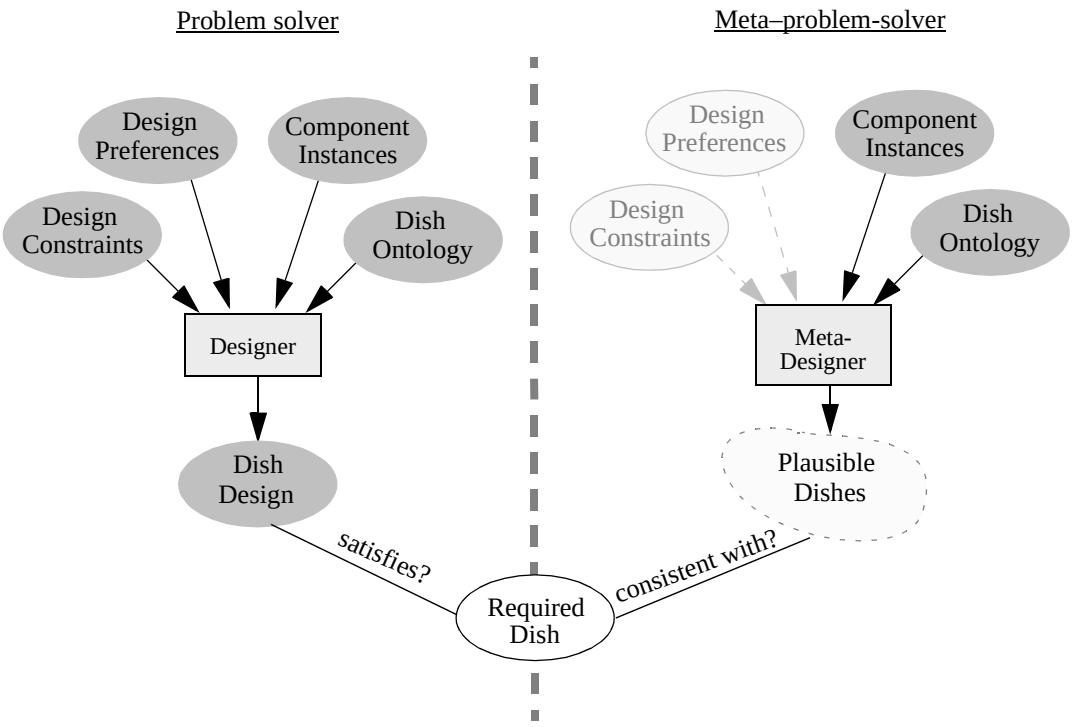


Figure 50: Inputs and Outputs of the Designer and Meta-Designer

My implementation of the meta-designer represents the contents of a CKRL knowledge base as a CLOS object, of class `CKRL-KB`. The `CKRL-KB` class defines slots for each of the CKRL constructs `concept`, `property`, `instance`, `sort`, `relation`, `fact`, `rule`, and so on. When a CKRL knowledge base is parsed, an object is produced whose slots are filled with the CKRL constructs[16] that the knowledge base contains. For example, when a knowledge base of dish components is parsed, an object is returned in which the slots `concepts`, `properties`, `relations`, `rules`, `facts` and `sorts` are all `NIL`, but the `instances` slot contains a list of the given component instances (such as `gammon-steak`, `sweetcorn`, and `peas`), together with *their* respective slot values.

16. The CKRL constructs are in a “lispified” form. That is, they are well-formed *s-expressions*, rather than character strings.

To check the *assignment of knowledge bases to input roles*, I decided to represent roles as CLOS classes too (see “Automatically Constraining Object Instances” on page 171). I defined an abstract class, called `KB-ROLE`, with a subclass dedicated to each problem-solving role. The class `KB-ROLE` is, in turn, a subclass of the `CKRL-KB` class. Each role-describing class asserts truths about the contents of knowledge bases that fulfil that role. In practice, this means that the class asserts constraints on its slots at object creation time. To test whether a given knowledge base satisfies a problem solving role, the knowledge base is first parsed, producing a `CKRL-KB` object. This object is then “merged” into the corresponding `KB-ROLE` object. That is, each of the slot values of the `CKRL-KB` object is asserted to be `equal` to the corresponding slot value of the `KB-ROLE` object. This has the effect of subjecting the knowledge base constructs to the necessary conditions for fulfilling the role. The function to perform the “merging” has already been mentioned: it is called `reconcile`. (Note that I assess the plausibility of the CKRL knowledge base, rather than the LISP knowledge base used by the problem solver. The success of this approach assumes that the automatic translation mechanism from CKRL to LISP is consistent with the direct interpretation of the CKRL.)

In the case of the meta-designer, we should like to ensure that the *dish ontology* contains definitions of the concepts `dish` and `dish-component`. If the variable `concept-names` is constrained to be the set of names of `concepts` contained in the knowledge base, then these two necessary conditions are guaranteed by the following SCREAMER+ assertions:

```
;; Assert that one of those concepts must be "dish"
(assert! (memberv "dish" concept-names))

;; Assert that one of them must also be "dish-component"
(assert! (memberv "dish-component" concept-names))
```

We should also like to ensure that the *component instances* knowledge base really does contain some instances. If the variable `instance-names` is constrained to be the set of names of `instances` contained in this knowledge base, then the existence of some instances is guaranteed by the following assertion:

```
;; Assert that there is at least one instance
(assert! (notv (equalv nil instance-names)))
```

For *inter-role consistency*, the *component instances* knowledge base should contain instances of only *known* concepts. That is, it should contain only `instances` of the concepts defined by the *dish ontology* knowledge base[17]. If the variable `ontology-concepts` is constrained to hold a list of all the names of concepts defined by the *dish ontology*, and the variable `instances` is constrained to hold a list of instances given by the knowledge base, then the necessary condition can be ensured by the following assertion:

```
(assert! (everyv #'(lambda(x) (memberv x ontology-concepts))
                 (mapcarv #'get-instance-type instances)))
```

To enable tests for the plausibility of a particular goal, the meta-designer outputs a *single CLOS object* representing a plausible design. The design is plausible because each *slot* that the object contains is also constrained to be plausible (see figure 51 for an example of a plausible design).

```
#<PLAUSIBLE-DESIGN @ #x8430db2> is an instance of
    #<STANDARD-CLASS PLAUSIBLE-DESIGN>:
 The following slots have :INSTANCE allocation:
  MAIN             [45 nonnumber enumerated-domain:(#<VEGETARIAN @ #x83ccd32>
                                                    #<FISH @ #x83ccd4a>
                                                    #<MEAT @ #x83ccd62>
                                                    #<MEAT @ #x83ccd7a>
                                                    #<MEAT @ #x83ccd92>
                                                    #<MEAT @ #x83ccdaa>)]
  CARBOHYDRATE     [46 nonnumber enumerated-domain:(#<CARBOHYDRATE @ #x83cccd2>
                                                    #<CARBOHYDRATE @ #x83cccea>
                                                    #<CARBOHYDRATE @ #x83ccd02>
                                                    #<CARBOHYDRATE @ #x83ccd1a>)]
  VEGETABLE        ([52 nonnumber enumerated-domain:(#<VEGETABLE @ #x83ccc5a>
                                                     #<VEGETABLE @ #x83ccc72>
                                                     #<VEGETABLE @ #x83ccc8a>
                                                     #<VEGETABLE @ #x83ccca2>
                                                     #<VEGETABLE @ #x83cccba>)]
                    [53 nonnumber enumerated-domain:(#<VEGETABLE @ #x83ccc5a>
                                                     #<VEGETABLE @ #x83ccc72>
                                                     #<VEGETABLE @ #x83ccc8a>
                                                     #<VEGETABLE @ #x83ccca2>
                                                     #<VEGETABLE @ #x83cccba>)])
  COST             [80 integer 5:7 enumerated-domain:(5 6 7)]
  DIETARY-POINTS   [88 integer 8:20 enumerated-domain:(8 9 10 11 12 13 14 15 16 17 18 19 20)]
```

Figure 51: A Plausible Dish Design

The slots of a dish include values for the `main-component`, `carbohydrate-component`, and `vegetables`. Each of the slots of the object representing the plausible design is therefore constrained to contain “sensible” values in accordance with the knowledge bases provided. In addition, slots for the dietary-value of a plausible dish, and the cost of a plausible dish have been calculated as the plausible sum of the corre-

17. A refinement of this condition would be to ensure that each concept instance sets only *known* and *relevant* `properties`.

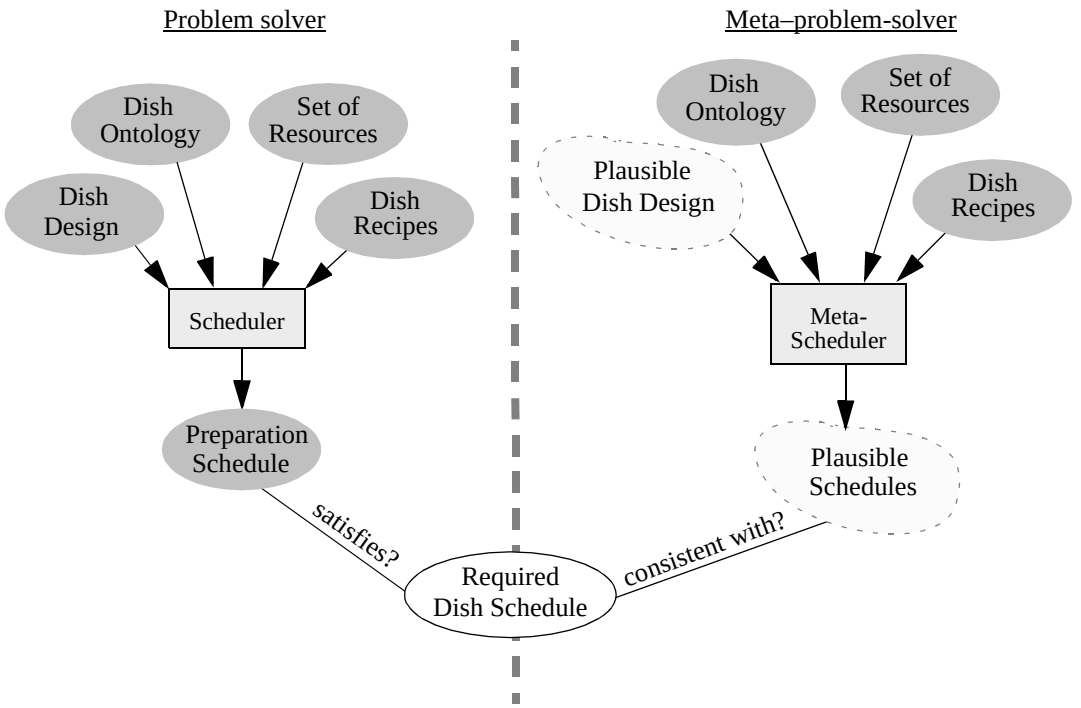


Figure 52: Inputs and Outputs of the Scheduler and Meta-Scheduler

sponding components. In effect, *the plausible overall design has been achieved by constraining its component parts to be plausible.*

Meta-scheduler

The inputs and outputs of the scheduler and meta-scheduler are illustrated in Figure 52.

Main Components		**Carbohydrates**		**Vegetables**	
Ingredient	**Prep. Time**	**Ingredient**	**Prep. Time**	**Ingredient**	**Prep. Time**
Lamb Chop	20	Jacket Potato	60	Runner Beans	10
Gammon Steak	15	French Fries	15	Carrots	15
Fillet of Plaice	15	Rice	15	Cauliflower	10
Sausages	10	Pasta	10	Peas	3
Chicken Kiev	30			Sweetcorn	5
Quorn Taco	10				

Table 10: Some Meal Preparation Knowledge used by the Scheduler

When checking a knowledge base used as an input to the scheduler for its suitability for the *dish ontology* role, the same set of conditions can be used as those used when checking the inputs to the designer. To check a knowledge base for its suitability for the *dish recipes* role, it must be ensured that the knowledge base contains instances of the `task` concept, because the scheduler assumes that recipes are defined as `tasks`. If `r-instances` is constrained to be the set of `instances` contained in the *recipes* knowledge base, then this condition can be guaranteed with the following assertion:

```
(assert! (somev #'(lambda(x) (equalv x 'task))
                (mapcarv #'get-instance-type r-instances)))
```

The scheduler also expects the *recipes* knowledge base to contain a rule called `schedule-genesis`. This rule ensures that no `task` is left without a bound value for its latest start time, because the problem solver would not be able to schedule the set of `tasks` if this slot were not set. If `r-rules` is constrained to hold the names of the rules in the *recipes* knowledge base, then the existence of the `schedule-genesis` rule can be guaranteed with the following assertion:

```
(assert! (memberv 'schedule-genesis r-rules))
```

In the *recipes* knowledge base, each recipe is represented as a set of facts describing some `subtasks` of the `root-task`. For *inter-role consistency*, each `uses` relation may only use known resources, each `precedes` relation may only refer to defined tasks, and each `subtask-of` relation may only refer to other tasks defined in the recipes knowledge base. These conditions are similarly ensured by asserting constraints across the appropriate parts of the role objects.

Goal plausibility is checked by calculating upper and lower bounds on the preparation time of a dish. These bounds are set in accordance with the principles depicted in the Gantt charts of Figure 53. Figure 53(a) depicts a scheme which my scheduler would certainly not generate because it contains a time period (mid-schedule) in which none of the tasks being scheduled is active. Figure 53(a) is detected as being implausible because the overall time for the `task` is greater than the worst case scenario shown in figure 53(b). This depicts a plausible, but worst-case, schedule, in which the tasks are carried out serially. This might happen if all the tasks require full use of some resource

(e.g. the grill, or a saucepan), and only one of those resources is available. This sets the upper bound of the dish preparation time as the *sum* of the durations of the composite tasks. In the best case, however, the tasks are carried out in parallel, as shown in Figure 53(c). This sets the lower bound for the preparation time as the duration of the *single lengthiest task*.

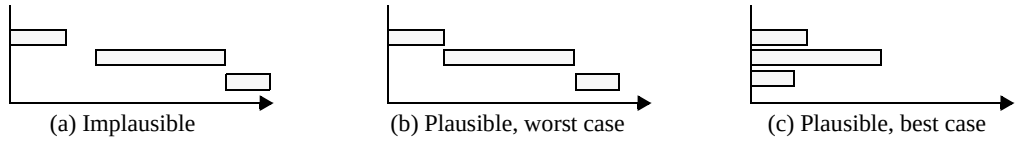


Figure 53: Gantt Charts of Implausible and Plausible Schedules

The meta-scheduler produces a description of a set of plausible dishes together with their plausible preparation times, which are derived from knowledge of the preparation times of dish components. The upper bound of the dish preparation time is the sum of the durations of the composite tasks; the lower bound is the duration of the lengthiest `task`.

6.3.2.3 Goal

Let us inspect some plausible dish preparation times and costs:

```
;;; First, call the meta-designer
;;; This outputs an object whose slots are constrained to
;;; be allowable values
> (setq my-design (meta:designer :instances "instances.ckrl"
                                 :ontology "background.ckrl"))
#<PLAUSIBLE-DISH @ #x99c86a>

;;; Then call the meta-scheduler
;;; This adds scheduling constraints to the design
> (setq my-dish (meta:scheduler :design my-design
                                :ontology "background.ckrl"
                                :recipes "recipes.ckrl"))
#<PLAUSIBLE-DISH @ #x99c86a>

;;; Retrieve the value of the duration slot
> (slot-value my-dish 'duration)
[774 integer 10:120]
```

```
;;; Retrieve the value of the cost slot
> (slot-value my-dish 'cost)
[769 integer 5:7 enumerated-domain:(5 6 7)]
```

This tells us that a plausible dish takes between ten minutes and two hours to prepare, and costs £5, £6, or £7. Now let us restrict our search to a fish dish, and again inspect the plausible values:

```
> (assert! (typepv (slot-valuev my-dish 'main) 'fish))
NIL

> (slot-value my-dish 'preparation-time)
[774 integer 15:105]

> (slot-value my-dish 'cost)
7
```

The plausible range size for the preparation time of the dish has decreased, and the cost of the dish has become bound without having to make any choices of vegetables. If the goal were to have such a fish dish ready in less than quarter of an hour, it would fail without having to know the details of the dish preferences or kitchen resources and without needing to run the scheduler:

```
> (assert! (<v (slot-valuev my-dish 'preparation-time) 15))
Error: Attempt to throw to the non-existent tag FAIL
  [condition type: CONTROL-ERROR]
>
```

In a similar way, if I define a quick dish to be one that takes less than half an hour to prepare, and a 'healthy' dish to be one that has a dietary points count of 16 or less, then it is easy to discover that there is no such thing as a quick dish that includes a jacket potato, or a 'healthy' dish that includes gammon and french fries!

6.4 Discussion

6.4.1 Desirable Approximations

Schaerf & Cadoli suggested that a desirable method of approximate reasoning should be *semantically well-founded*, *computationally attractive*, *improvable*, *dual*, and *flexible* (Schaerf & Cadoli, 1995). If the method is **semantically well-founded**, it can provide clear information about the problem at hand. As a result of this information, the method should enable a user to work more effectively. A method of approximate rea-

soning is only **computationally attractive** when the approximate answers it generates are easier to compute than answers to the original problem. Approximate answers should be **improvable**, so that, given enough time and motivation, they would eventually converge to the right answer. The issue of **duality** refers to the possibility that an approximation may be either sound or complete. Schaerf & Cadoli propose that an approximation which has a version of both forms is desirable. A **flexible** form of approximate reasoning is applicable to a wide range of reasoning problems.

Semantic Well-Foundedness — To be semantically well-founded, a method for approximating fitness for purpose should provide an output with a clear meaning, which can provide the basis for further human decision-making. This is certainly the case with the proposed approach, because even if the answer generated by the procedure is not completely certain (recall that the procedure generates a definite "no" or a "don't know"), the action to be taken by the user is evident[18]. In fact, my approach can be seen as a reaction against some of the methods used in AI for reasoning with uncertainty. The MLT Consultant, for example, like MYCIN, used *certainty factors* to qualify the degree of belief in its recommendations (see, for example, Winston, 1984; or Russell & Norvig, 1995). Certainty factors can be applied whenever a parameter is assigned a value. A certainty factor (CF) is a real number between -1 and +1. A CF of +1 means that the parameter certainly has the value, whilst -1 means that the parameter certainly does not have the value. A CF of 0 means that there is no evidence to suggest whether the parameter has the value or not. Whenever inferences are made, the certainty factors of the contributing premises are combined with mathematical formulae to generate a new certainty factor for the conclusion. The problem is that whilst CF values of -1, 0, and +1 might be well defined, the interim values are very ill-defined. For example, what would it mean to say that a knowledge base configuration is fit for the purpose of solving a goal with a certainty factor of 0.5? What does it mean for the user? What are the appropriate actions to take? I suggest that since a certainty factor can only be meaningfully compared with other certainty factors, it is an intermediate value for computer consumption, rather than a basis for human decision making.

18. A knowledge base is assumed to be fit for the purpose of solving a goal unless proved otherwise.

Computational Attractiveness — The words of Polya (Chapter 4, page 96) have already highlighted the necessity for the computational attractiveness of an approximation. However, the conceptual framework and LISP-based implementation currently provide no assurance that this requirement has been upheld by the author of a meta-description. This is the responsibility of the meta-description's author. Moreover, I require that the author of a meta-description accept the following three obligations:

- to ensure the computational attractiveness of the approximation;
- to ensure that the approximation is effective in detecting implausible configurations; and,
- to ensure that the approximation faithfully represents properties of the ground-level problem solver.

As we have seen, one way in which a meta-description can provide a computationally attractive approximation of the behaviour of a problem solver is through *abstraction*. A more pragmatic approach to the problem is to provide the user with the opportunity to abort the computation at regular intervals. I have used the multi-processing capabilities of LISP to write a macro called `with-abort-check`, which exhibits this property (see Appendix C). This macro can be applied to the evaluation of an arbitrary LISP expression, including the execution of meta–problem-solvers.

Improvability — Schaerf and Cadoli said of improvability

> 'approximate answers can be improved, and eventually they converge to the right answer (provided we have enough time and motivation)' (Schaerf & Cadoli, 1995).

However, this definition suggests two distinct interpretations of the notion of improvability. In the first interpretation, an approximation is improved by a human as and when required. This is the current status of the approximation described in this chapter. In the second interpretation, an algorithm is able to improve the approximation in an iterative fashion, in a similar way to the Newton-Raphson method for finding the roots of a polynomial. Eventually it would converge on some "optimal" value. I believe that finding such an algorithm for fitness-for-purpose approximations would be a considerable challenge, not only because the approximation must be guaranteed to contain all the possible solutions (how can an algorithm prove this?), but also because the algo-

rithm must respect the three obligations of the meta-description given previously (page 196).

Duality — A desirable property of an approximate form of reasoning is that both sound and complete versions could be described. A *sound* version of the fitness-for-purpose approximation would be able to prove that a KB *is* suitable for the task at hand (in contrast with the complete version, which proves that a KB *is not* suitable for the task at hand). Although one can describe such an approximation, I believe it would be of little use to the problems of MUSKRAT since the proofs of suitability would either be somewhat trivial and uninteresting for the user (e.g., 'file `foo.txt` is suitable as input to the `textedit` text editor because it contains only ASCII characters'), or require significant proof apparatus, such as an interactive theorem prover. Note also that a sound approximation might be unable to prove that a particular input is, indeed, usable.

So far, I've been describing *necessary* conditions; the sound approximation would encode *sufficient* conditions. The sound approximation could be useful as long as there are some interesting sufficient conditions.

Flexibility — For flexibility, 'the approximation schema should be general enough to be applied to a wide range of reasoning problems' (Schaerf & Cadoli, 1995). I believe this is true of the approach described in this chapter because the conditions to be satisfied are expressed declaratively as constraints. Furthermore, SCREAMER+ provides a powerful notation for the expression of these constraints, and the mechanisms maintaining consistency and searching for solutions is completely domain independent.

6.4.2 Plausibility as a Binary Decision Process

The approach to fitness for purpose described here is a *binary decision* process, that is, a configuration of problem solver and knowledge bases is simply deemed to be either (plausibly) fit or unfit for the purpose of solving a given goal. Critics have suggested that it might be more useful to yield, say, a decimal number between 0 and 1 which indicates the *level* to which a knowledge base is fit for a purpose. This has the advan-

tage that knowledge bases which *almost* satisfy the fitness-for-purpose criteria can be recognised as such, rather than simply being rejected.

However, there were good reasons for making a binary decision as a first step. Firstly, recall the three actions identified in chapter 4. This chapter has attempted to tackle the first of these: to 'determine whether the knowledge required for solving a given problem is available and *in a form suitable for immediate use*'. Since the action makes no mention of being *almost* suitable, the research task I set myself required a binary decision procedure. Any other kind of response would have upset the clear boundary between this and the last of the three actions: to 'modify existing knowledge to meet the requirements of the problem solver'. In Chapter 8, I propose an extension to the current work which enables the recognition of *partially fulfilled* fitness-for-purpose criteria.

6.4.3 Soundness and Completeness

The plausibility model characterises a semi-decision procedure which is semantically well-founded. It is "only" a semi-decision procedure because of the nature of the approximation to fitness for purpose. If the tests can prove that a KB is unsuitable for use with a problem solver to achieve a given goal, then they have provided valuable information; if they cannot prove unsuitability, then no knowledge has been gained. In effect, a plausibility approximation, together with its associated tests, returns either a "no" or a "maybe yes" answer to the question of fitness for purpose. Schaerf & Cadoli state the relationship between such approximations and the important properties of *soundness* and *completeness*:

> '... an approximate solution to a decision problem is a "maybe" answer, equipped with reasons to believe that the "maybe" is actually a "yes" or to believe that it is actually a "no". In a form of approximate reasoning called *sound* reasoning we have two possible answers: "yes" and "maybe no". In the dual form of approximate reasoning (*complete* reasoning), the two possible answers are "no" and "maybe yes".' (Schaerf & Cadoli, 1995)

The plausibility model introduced here is complete, but unsound. However, the fact that it is unsound does not prevent it from being useful. Consider an alarm signal raised by the software which monitors a reactor at a nuclear power station. In the interests of safety, it is preferable that the alarm be sometimes raised unnecessarily (i.e., generate 'false positives') than to miss a real emergency (i.e., allow 'false negatives'). The decision procedure for raising the alarm is therefore also complete, but unsound.

The example also demonstrates that when making a decision, the *cost* of making the wrong choice should also be considered (cf. decision theory). A nuclear power station cannot afford to miss a genuine emergency, because of its serious consequences. This consideration of cost is similar to that of the plausibility approximation to fitness for purpose, but clearly less (safety-) critical: if a MUSKRAT user fails to identify a KB which would help to solve the problem at hand, it is a *missed opportunity for knowledge reuse* rather than a genuine calamity.

6.4.4 Functional and Dynamic Properties

The mechanics of the problem solvers described in Section 6.3 may have been unspectacular, but this does not devalue the merit of their meta-descriptions. Indeed, the meta-descriptions are ignorant of the problem solvers' mechanics, because they are *functional descriptions*. The meta-layer might use "insider", or "deep" knowledge of the problem solver, but essentially provides a black box description of the problem solver's behaviour, because it encodes only the problem solver's input–output relationship and not how it works internally. The latter is a strong argument for the generality of the approach – it does not matter what algorithms the problem solvers employ, the main issue is *whether the language of constraints used for describing the inputs and outputs is sufficiently expressive*. (My constraint-based solutions to the problems discussed in chapter 5 are evidence of expressiveness.)

The designer and scheduler of Section 6.3.2.1 can either be used independently, or in combination – the scheduler can schedule the preparation of the design proposed by the designer. Likewise, the corresponding meta–problem-solvers can also be cascaded. The possibility of cascading problem solvers in this way is interesting because when

analysing a complex problem solver, one might split its functionality into connecting parts and analyse those parts separately.

An important limitation of the described approach to determining fitness for purpose is that it cannot differentiate between problem solvers which are *functionally equivalent*. That is, since they have the same inputs and outputs, and the described approach considers *only* the inputs and outputs, then the descriptions of the problem solvers must also be the same. One way to address this limitation is to supplement the functional descriptions of problem solvers with descriptions of their *dynamic properties*. An example of a dynamic property is the CPU time required by a computation. Recently researchers have begun to consider how these dynamic properties of problem solvers might be described and verified (Groot, ten Teije & van Harmelen, 1999).

Chapter 7

Acquiring Knowledge which is Fit for a Purpose

'The only failure a man ought to fear is failure of
cleaving to the purpose he sees to be best.'
George Eliot

Chapter Summary

To acquire knowledge which is fit for a specific purpose, it is very desirable to have a structured, declarative expression of the knowledge which is needed. This chapter introduces a stand-alone knowledge acquisition tool, called COCKATOO (Constraint-Capable Knowledge Acquisition Tool), which uses constraint technology to help specify its requirements. The language in which these specifications are given is based on the meta-language notation of context-free grammars. Since the implementation was realised using the technology of constraints, however, it also provided the opportunity to build a more exacting tool by augmenting context-free grammars with the expressiveness of constraints. COCKATOO was implemented using the declarative constraints package of SCREAMER+.

7.1 Introduction and Motivation

Chapter 4 stated three actions concerning fitness-for-purpose (see page 93). The first of these, to *determine whether the knowledge required for solving a given problem is available and in a form suitable for immediate use*, was discussed in chapter 6, and illustrated with a solution which applied the constraint technology developed in chapter 5. This chapter addresses the second of the three actions concerning fitness for pur-

pose; namely, to *acquire knowledge ab initio such that it meets the requirements of the problem solver*. Typically, a knowledge engineer would use a knowledge acquisition tool, or other elicitation methods, to acquire the knowledge needed by a problem solver. Usually the knowledge must then be transformed, because the output of the knowledge acquisition tool cannot be used directly as input to the problem solver.

In this chapter, I demonstrate how the declarative constraints package of SCREAMER+ can be used to build a knowledge acquisition tool which specifies the knowledge required by a problem solver as a context-free grammar. Later, I also show how constraints can be integrated into such a grammar, thereby increasing the expressive power of the tool. The tool itself is *generic*, in the sense that it is independent of task, problem solver, and domain. On the other hand, it can also be configured to acquire knowledge suited to a particular purpose (i.e., a problem-solving role — see section 7.5.2). When custom-tailored in this way, the need for post-acquisitional transformations is minimised.

7.2 Formal Grammars

Informally, a **grammar** is a mechanism for describing the set of **sentences** which belong to a **language**[1]. Most of us can remember school grammar lessons which purported to teach the difference between admissible and inadmissible sentences of natural language. This chapter is concerned with **formal grammars**, which are often used to define the admissible and inadmissible sentences of **formal languages**. When dealing with *formal grammars*, a *sentence* is a sequence of symbols, and a **production clause** (also called simply a **production** or a **clause**) expresses the possibility of rewriting one sequence of symbols as another sequence. A *formal grammar* (hereafter referred to as a 'grammar') is defined by a finite set of *production clauses*.

1. In natural language, grammars are predominantly *descriptive*; that is, the grammar attempts to *describe* the sentences of an existing language. With formal languages, grammars are chiefly *prescriptive*; that is, they *define* the admissible and inadmissible sentences of the language.

```
[1]   expr   → <value> AND <value>
[2]   expr   → <value> OR <value>
[3]   expr   → <value> NAND <value>
[4]   expr   → <value> XOR <value>
[5]   expr   → NOT <value>
[6]   expr   → <value>
[7]   value  → TRUE
[8]   value  → FALSE
[9]   value  → (<expr>)
```

Figure 54: Production Clauses Specifying a Language of Computable Boolean Expressions

For example, Figure 54 contains a set of (numbered) clauses which specify a syntactically restricted "language" of computable Boolean logic expressions. The first clause states that an expression (`expr`) can be written as a `value` followed by the symbol '`AND`', followed by another `value`. By referring to clauses 7, 8 and 9, we see that a `value` can be one of `TRUE`, `FALSE`, or a bracketed expression. So, for example, by using clauses 1 and 7, we can confirm that '`TRUE AND TRUE`' is an admissible sentence of the grammar.

The production clauses consist of **terminal symbols** and **non-terminal symbols**. Whilst a *terminal symbol* captures part of the language being specified, *non-terminal symbols* are only part of the language description and do not appear in any expression generated by the grammar. *Non-terminal symbols* are enclosed in angled brackets < >, and refer to other clauses of the grammar. Examples of *terminal symbols* are `AND`, `OR`, and `TRUE` (in clauses 1, 2 and 7, respectively); the only *non-terminal symbols* are `<expr>` and `<value>`. Note that although the language is *syntactically* restricted (i.e., it cannot generate all well-formed Boolean expressions[2]), the number of admissible sentences is still infinite because there is mutual recursion on the *non-terminal symbols* `<expr>` and `<value>`.

2. For example, it cannot generate the sentence '`TRUE AND TRUE AND TRUE`'.

Chomsky classified grammars into different types according to the form of their clauses. The most important of these types[3] are **context-sensitive grammars**, **context-free grammars**, and **regular-expression grammars**.

Definition 15: Context-Sensitive Grammar

'A *type 1* or *context-sensitive grammar* contains only productions of the form

A → alpha

where A *is a non-empty string of non-terminal symbols*, alpha is a non-empty string of terminal and/or non-terminal symbols and the length of the right-hand-side string alpha is not less than the length of the left-hand-side string A.' (Bornat, 1986).

Definition 16: Context-Free Grammar

'A *type 2* or *context-free grammar* contains only productions of the form

A → alpha

where A *is a single non-terminal symbol* and alpha is a string of terminal and/or non-terminal symbols.' (Bornat, 1986).

Definition 17: Regular-Expression Grammar[4]

'A *type 3* or *regular expression grammar* contains only productions which are one of the forms A → a

A → aB

in which A and B are single non-terminal symbols, a is a single terminal symbol.' (Bornat, 1986).

---

3. Actually, Chomsky defined only one other type – type 0, or *unrestricted grammars*, whose production clauses contain a sequence of non-terminal symbols on their left-hand side, and a mixture of terminal and non-terminal symbols on their right-hand side.

4. There is an alternative definition of a regular expression grammar in which the second form of production is A → Ba. This would be a *left-recursive* definition, instead of the *right-recursive* definition given above. A regular expression grammar may be either left-recursive or right-recursive, but not both.

Notice that the productions of type 3 grammars have a more restricted form than those of type 2 grammars, and the productions of type 2 grammars are more restricted than those of type 1 grammars[5]. Programming languages are often described with type 2 grammars because they represent a compromise between simplicity and efficiency of processing, and the expressiveness of the language. (Regular expressions can be parsed very efficiently with a finite state machine, but cannot perform bracket matching to an arbitrary depth. On the other hand, parsers for context-sensitive grammars tend to be more complex than those for context-free grammars because they need to look beyond the current symbol before deciding which production to use.) The work described here also concentrates on context-free grammars.

7.2.1 Backus-Nauer Form (BNF)

The grammar given in Figure 54 was specified using a particular notation, called BNF (Backus-Nauer Form[6]). This is a popular way of formally specifying a context-free grammar. The productions of BNF use **sequential composition** to specify which symbols follow which other symbols. For example, the production

```
<expr> → NOT <value>
```

sequentially composes the terminal symbol `NOT` with the non-terminal symbol `<value>`. Sometimes, a single BNF clause also contains **alternative expansions**. This can help to keep the language specification concise because the left-hand side of the clause does not have to be repeated for every possible expansion.

For example, an alternative, and much more concise, formulation of the language for Boolean expressions is given in Figure 55.

```
[1]   expr   → <value> <op> <value> | <value> | NOT <value>
[2]   op     → AND | OR | NAND | XOR
[3]   value  → (<expr>) | TRUE | FALSE
```

Figure 55: An alternative formulation of the Restricted Language for Boolean Expressions

5. Type 2 grammars are allowed only a *single* non-terminal symbol on the left-hand side of a production, whereas type 1 grammars are allowed a *string* of non-terminal symbols.
6. Also known as the Backus-Normal Form.

7.2.2 Extended Backus-Nauer Form (EBNF)

Two extensions have been added to BNF for further convenience, resulting in an equally powerful, but more concise, notation called EBNF (Extended Backus-Nauer Form). The two additions were operators for **optionality** and **repetition**. A symbol which is *optional* may or may not be included in the expansion of a production's left-hand side. *Optional symbols* are denoted by (square) brackets '`[]`'. *Repetitions* are denoted by either '`+`' or '`*`'. A '`+`' indicates that the preceding term occurs one or more times in succession; a '`*`' indicates that the preceding term occurs zero or more times. Parentheses can also be used to specify the operator precedence.

For example, an `integer` could be defined as:

`integer` → `[-]<digit>+`

and an identifier, such as those used for variable names in programming languages, could be defined as:

`identifier` → `<letter> (<letter> | <digit>)*`

where the non-terminals `<digit>` and `<letter>` should also be defined appropriately.

Although EBNF may still appear to be simple, it contains all the features needed to build up some very complex language structures. Moreover, a representation of these features can be expressed as SCREAMER+ constraint variables, and further used to drive a knowledge elicitation process.

7.3 Grammar-Driven Knowledge Elicitation

When eliciting knowledge, it is very desirable to have a structured, declarative specification of the body of knowledge that needs to be acquired. This can be used as both the target of the knowledge acquisition process and the criterion by which the acquired knowledge is assessed. It is desirable for the specification to be *structured*, so that it can be machine-interpreted, and its coverage easily ascertained. The specification

should be *declarative* because such statements are human-readable, concise, and contain little of the low-level, solution-specific detail.

Formal grammars provide a means for specifying knowledge to be acquired. Furthermore, they provide a means which is not only structured and declarative, but also widely understood by knowledge engineers and computer scientists. However, there is an important difference in the way that formal grammars are "traditionally" used, and the way that they have been applied here. Traditionally, grammars are used to solve the **parsing problem**; that is, to determine whether some *given* text conforms to some *given* formal grammar. For example, a C compiler must determine whether the program it should compile consists entirely of legal C syntax. In grammar-driven knowledge elicitation, however, one attempts to *acquire* structured text *such that it conforms to the given grammar*.

7.3.1 Representing a BNF Grammar

The representation of a BNF grammar must be able to deal with the terminal and non-terminal symbols that may be contained in a clause, as well as the operators for sequential composition and the expression of alternatives. To meet these requirements, the COCKATOO tool represents each production clause of a grammar as a CLOS object. This object contains (amongst others) a slot for the name of the production, and another slot for a list expression that defines the expansion of the clause. This expression may contain terminal symbols (i.e., LISP atoms and sequences), non-terminal symbols (enclosed by angled brackets <>), and the functors `seq` and `one-of` to represent sequential composition and rewrite alternatives, respectively. Clauses are introduced by the new macro `defclause`.

For example, the single clause BNF grammar

`snack` → `(bread <snack> bread) | cheese`

is expressed to the COCKATOO tool as follows (you may also like to refer to the full examples of grammar elicitation given on pages 217-227):

```
(defclause snack ::=
  (one-of (seq 'bread <snack> 'bread) 'cheese)
  )
```

Internally, a `clause` object has been created. This can be confirmed by calling the method `find-clause` with the name of the clause. Two values are returned: the clause object, and an object representing the grammar in which the clause was found.

```
> (find-clause 'snack)
#<CLAUSE:SNACK>
#<GRAMMAR:MISC>
```

When acquiring a value for the clause, the expression is first evaluated, and any remaining unknowns in the result are also acquired. These unknowns can also be clauses (returned by evaluating non-terminal symbols). The acquisition method therefore performs a depth-first search of the grammar clauses, acquiring values along the way. For example, evaluation of the expression slot of the `snack` clause yields a constraint variable with an enumerated domain of size two: either the list `(BREAD [#<CLAUSE:SNACK>] BREAD)` or `CHEESE`. If the user chooses the former, then a `snack` clause is again acquired, and the process recurses.

Note that COCKATOO permits only one clause with any given name. This restriction implies no loss of generality since a set of clauses

$$\texttt{my-clause} \rightarrow \texttt{expansion}_1.$$
$$\texttt{my-clause} \rightarrow \texttt{expansion}_2.$$
$$\ldots$$
$$\texttt{my-clause} \rightarrow \texttt{expansion}_n.$$

can always be rewritten as

$$\texttt{my-clause} \rightarrow \texttt{expansion}_1 \mid \texttt{expansion}_2 \mid \ldots \mid \texttt{expansion}_n.$$

7.3.2 Representing an EBNF Grammar

Since EBNF is an extension of BNF, it can be represented by extending the representation of BNF. COCKATOO introduces three additional functors, `repeat+`, `repeat*`, and `optional`. The `repeat` functors represent ‘zero or more’ and ‘one or more’ repetitions of their arguments, respectively. The argument to an `optional` functor may or may not be included in the clause’s expansion. As an example, consider the following EBNF grammar, which describes a special offer for shoppers in a café: for the same

price customers may choose a bottomless pot of tea with a selection of biscuits (from a large supply), or a cup of coffee with some apple pie and optional cream.

```
shoppers-break →(tea+ biscuit*) | (coffee apple-pie [cream])
```

This can be expressed to COCKATOO as

```
(defclause shoppers-break ::=
  (one-of (seq (repeat+ 'tea) (repeat* 'biscuit))
       (seq 'coffee 'apple-pie (optional 'cream)))
  )
```

7.3.3 Adding Questions and Comments

It is unrealistic to expect users to base their interaction with a knowledge acquisition tool on their understanding of an EBNF grammar. It should be clear *what* information is required, *why* it is required, and *how* it can be supplied (i.e., what is the format of the input). Ideally, the consequences of their actions for the rest of the knowledge acquisition session should also be clear.

In response to these requirements, COCKATOO allows each clause of a grammar to be associated with both a *question* and a *comment*. These are pieces of canned text that provide additional information, and can be integrated with the grammar as part of the data that drives the KA process. A *question* should be a request for feedback which is directed at the user, such as "`Do you want to claim a repayment if you have paid too much tax?`". A *comment* provides additional information, such as the meaning of particular terms, the exact format of the input, or other explanatory or "small-print" material. For the tax claim example, a comment might read "`If you do not claim a repayment, then any amount you are owed will be set against your next tax bill`".

Questions and comments are provided as keyword arguments to the `defclause` macro. Here is a clause definition for the tax claim example.

```
(defclause repayment-claim? ::= (one-of 'yes 'no)
  :question "Do you want to claim a repayment if you have paid
too much tax?"
  :comment "If you do not claim a repayment, then any amount you
are owed will be set against your next tax bill."
 )
```

When the user's response is acquired according to this clause, either `YES` or `NO` is returned as the value of the acquisition.

```
> (acquire (find-clause 'repayment-claim?))

If you do not claim a repayment, then any amount you are owed
will be set against your next tax bill.

Do you want to claim a repayment if you have paid too much tax?

The possible values are:
  A.  YES
  B.  NO
Which value? : yes
YES
```

7.3.4 Postprocessing of Acquired Knowledge

Although the user may input the *knowledge content* required by a problem solver to the knowledge acquisition tool, it would be a rare case if the format of the user's inputs were exactly the same as the format required by the problem solver. Usually, some kind of syntactic transformation would need to be performed[7]. To achieve this functionality, COCKATOO allows a postprocessing function to be specified for each clause. This is a single argument function that is applied to the value acquired by the clause.

For example, the tax claim clause asks a question of the user which must be answered with either `yes` or `no`. For a LISP program to process this answer, however, it might have to be converted to `t` or `nil`, respectively. Therefore the postprocessing function supplied is simply a lambda function[8] which performs this conversion.

```
(defclause repayment-claim? ::= (one-of 'yes 'no)
  :question "Do you want to claim a repayment if you have paid
too much tax?"
  :comment "If you do not claim a repayment, then any amount you
are owed will be set against your next tax bill."
  :postproc (lambda(x) (if (equal x 'yes) t nil))
  )
```

When COCKATOO acquires this clause, either `t` or `nil` is returned, as appropriate:

```
> (acquire (find-clause 'repayment-claim?))
```

7. The alternative is to demand that the user enters knowledge in the exact format required by the problem solver, but this is not usually very "user-friendly".
8. An unnamed function.

```
If you do not claim a repayment, then any amount you are owed
will be set against your next tax bill.

Do you want to claim a repayment if you have paid too much tax?

The possible values are:
  A.  YES
  B.  NO
Which value? : yes
T
```

Note that the mechanism accommodates *arbitrary* post-acquisitional transformations. I have demonstrated that it can be used for simple *syntactic* transformations; I believe it could also be used as the call-out mechanism for supporting deeper *semantic* transformations of the sort needed to properly address the third action of chapter 4 (see page 93).

Since no restriction is made on the nature of the postprocessing function, the output of a clause does not have to depend on the clause itself. Although arguably inelegant, this is a useful feature that allows the programmer to incorporate general processing capabilities with the grammar. Assuming that the function `date` is already defined, the following clause actually ignores the grammar definition, and returns a string containing the current date:

```
(defclause dated-header ::= 'dummy
  :postproc (lambda(x) (format nil ";;; Today is ~a" (date)))
  )
```

For example,

```
> (acquire (find-clause 'dated-header))
";;; Today is 07-AUG-99"
```

7.4 Constraint-Augmented Grammars

This section shows how a knowledge elicitation grammar can be augmented with constraint expressions. This can improve the conciseness and readability of the grammar, reduce its development time, and enhance its expressiveness. Note that this view of knowledge elicitation is not inconsistent with the definition of a constraint satisfaction problem (see page 46). For example, consider a structured interview (see “Interviewing” on page 18) in which the answers to the knowledge engineer’s agenda of questions

are the variables of the problem, and there are concrete expectations about what their allowable values (the variable's domain) might be.

As we have seen, Grammar-Driven Knowledge Elicitation is a precise and powerful mechanism for acquiring knowledge. However, by combining the grammar-driven approach with constraint technology, we gain the following advantages.

Concise Specifications – Knowledge specifications for some tasks can be written much more concisely, thus giving a more readable specification, and also saving development time.

Single-Input Property Checking – The required properties of each user input can be checked at *acquisition time*, rather than prior to problem solving or at problem-solving time. That is, inadmissible values are identified early in the knowledge acquisition cycle. A property is expressed naturally as a constraint on the input.

Multiple-Input Property Checking – Required properties of *multiple* inputs can also be checked at *acquisition time*. A property of this kind is expressed as a constraint among *multiple* inputs.

Reactive User-Interfaces – Constraints across multiple inputs can be constructed in such a way that the user-interface appears to react to the user's inputs. For example, selection of a particular value for one input might reduce the number of choices for a second input.

7.4.1 Concise Specifications

Suppose that we wished to acquire knowledge of historical events that occurred between the years 1066 and 2000. The date of each event is given by an integer within that range. However, with a purely grammar-driven approach, a part of the acquisition grammar would have to be dedicated to accepting either the sequence of characters that compose the integers of the range, or the enumeration of all of the acceptable values. Devising a grammar for this purpose would appear somewhat foolhardy, given that a range test on an integer input would be very simple.

We have already seen how a clause can be defined by combining values with the keywords `seq`, `one-of`, `optional`, `repeat+` and `repeat*`. Until now, it has not been clear that a clause can be defined to be an *arbitrary* LISP expression. So, for example, the following is acceptable COCKATOO syntax (if somewhat futile):

```
(defclause foo ::= (+ 3 4))
```

The value of the clause can be acquired, as one would expect:

```
> (acquire (find-clause 'foo))
7
```

The usefulness of the concept starts to become clear when constraint expressions are written as the value of a clause. For example, the following concise clause accepts only an integer in the (inclusive) range 1066 to 2000:

```
(defclause event-year ::= (an-integer-betweenv 1066 2000)
   :comment "Enter a year between 1066 and 2000"
   )
```

A non-integer value, or an integer which is out of that range, is not accepted as a value of the clause:

```
> (acquire (find-clause 'event-year))
Enter a year between 1066 and 2000
Input a value: 999

That value causes a conflict. Please try another value...
Input a value: 1665
1665
>
```

For problems such as this, the simple constraint-based clause is much more maintainable than the equivalent grammar-based solutions without constraints.

7.4.2 Single-Input Property Checking

In the previous section, I argued that a grammar would be capable of describing the set of integers in the range 1066-2000, but the introduction of constraints made the solution much more concise. For that problem, the constraint-based approach was no more *powerful* (in terms of expressiveness) than the purely grammar-based approach[9],

9. Both approaches solved the problem.

```
(deftype prime ()
  "Defines a type for prime numbers"
  '(and integer (satisfies primep)))

(defun primep (x)
  "Returns Boolean value indicating whether argument is prime"
  (cond
   ((< x 2) nil)                ; 2 is the smallest prime
   ((= x 2) t)                  ; 2 is a prime
   ((evenp x) nil)              ; all other evens are not prime
   ((divideable x (sqrt x)) nil) ; not prime if divider exists
   (t t)))                      ; else prime

(defun divideable (n limit)
  (do (
       (counter 3 (+ 2 counter))
       (divides nil)
       )
      ((> counter limit) divides)
    (when (zerop (mod n counter))
      (setq divides t))))
```

Figure 56: Defining a type for prime numbers

though it clearly offered advantages. In this section, I shall demonstrate that a constraint-augmented grammar is also more *powerful* than a purely grammar-based approach.

Let us first observe that it is not possible to write a context-free grammar that describes the set of all prime numbers. The property of primeness is simply too complex to be incorporated into a grammar. However, such a property *can* be checked by a constraint-augmented grammar. To write a clause to acquire a prime number using COCKATOO, I would first define a type called `prime`, as given by the code in figure 56. Then I could write a clause that acquired a value of that type:

```
(defclause prime-num ::= (a-typed-varv 'prime)
  :comment "Enter a prime number"
  )
```

Then, when acquiring a value according to that clause, any value that is not a prime number will be rejected, viz.

```
> (acquire (find-clause 'prime-num))
Enter a prime number
Input a value: 25
```

```
That value causes a conflict. Please try another value...

Input a value: 29

29
>
```

Recall that membership of a type can be subject to satisfaction of a *arbitrary* LISP predicate, so the mechanism for checking the properties of a single input value is a general one.

7.4.3 Multiple-Input Property Checking

Another way of specifying values that could not be expressed by a context-free grammar is to assert constraints across multiple input values. For example, a context-free grammar would not be able to constrain two variables to have different values unless it *explicitly* represented all those situations in which the values were different. At best, this represents much work for the implementor of the grammar. If the variables do not have a finite domain, however, it is not even possible. The following clause returns a sequence of two variables which are constrained to be different.

```
(defclause two ::=
  (let (
        (a (make-variable))
        (b (make-variable))
        )

     (assert! (not-equalv a b))
     (seq a b)
     )
  )
```

When acquiring a value for the clause, the second value must be different to the first:

```
> (acquire (find-clause 'two))
Input a value: tweedle-dum
Input a value: tweedle-dum
That value causes a conflict. Please try another value...

Input a value: tweedle-dee

(TWEEDLE-DUM TWEEDLE-DEE)
>
```

7.4.4 Reactive User-Interfaces

Constraints can also be used to modify the behaviour of the acquisition tool, depending on the values supplied by the user. This is the issue of **reactive knowledge acquisition** mentioned earlier. As an example, suppose that a medical application needs to acquire some personal details about a patient prior to a course of treatment. These details could include the patient's sex, and whether he or she is pregnant. Clearly, if the patient is male, then the pregnancy question is redundant. This redundancy cannot be realised by a context-free grammar because it is not known whether the patient is male or female until acquisition time. A context-free grammar cannot build in such conditions at 'compile time'. Such conditions *can* be realised by a constraint-augmented grammar, however, as illustrated by the following clause.

```
(defclause personal-details ::=
  (let (
        (sex (a-member-ofv '(male female)))
        (pregnant? (a-member-ofv '(pregnant not-pregnant)))
        )

    (assert! (impliesv (equalv sex 'male)
                       (equalv pregnant? 'not-pregnant)))
    (seq sex pregnant?)
    )
  )
```

When the clause is acquired and the sex of the patient is female, the question of pregnancy arises:

```
> (acquire (find-clause 'personal-details))
The possible values are:
  A.  MALE
  B.  FEMALE
Which value? : female

The possible values are:
  A.  PREGNANT
  B.  NOT-PREGNANT
Which value? : b
(FEMALE NOT-PREGNANT)
```

When the patient is male, however, the logical implication 'fires' and the value of the variable `pregnant?` becomes bound. The second question therefore need not be asked:

```
> (acquire (find-clause 'personal-details))
```

```
The possible values are:
  A.  MALE
  B.  FEMALE
Which value? : male
(MALE NOT-PREGNANT)
>
```

In the example, the determination of a variable's value at acquisition time caused another variable to become bound. This is one type of reactive behaviour. Another possible situation is that when a variable becomes bound at acquisition time, the domain size of a related variable is reduced. Consider an on-line shopping facility for flowers, where a choice of the largest bouquet allows the customer to select from a wide range of flowers, including those which are expensive. Choosing a smaller bouquet might limit the choice of flowers to those which are less expensive.

7.5 Evaluation

The examples presented in this section have been chosen to verify and illustrate the features of COCKATOO. The first example acquires a simple design (a breakfast order) by configuring a set of known components. The second example demonstrates how post-processing can be applied to the acquired knowledge so that it is immediately usable by a program. The third example shows the application of COCKATOO to a more realistic domain – the acquisition of a case base for oil well drilling.

7.5.1 Acquiring a Breakfast Order

Suppose that the rooms in a large hotel provided a computer terminal through which customers could place their room service orders. With such a system installed, orders could be electronically delivered *directly* to the hotel kitchens, and the hotel receptionist need not be involved in their acquisition and processing. Furthermore, if a grammar is used for specifying the acquisition of breakfast orders, then the choices can be modified very easily – perhaps on a weekly, or even daily, basis.

Figure 57 contains a grammar which could be used in this situation for acquiring precise customer orders for breakfast. It offers three kinds of breakfast: a `fried-breakfast`, a `continental-breakfast`, and a `cereal-breakfast`. These three

```
;;; A grammar for Acquiring Breakfast Orders.
(defgrammar breakfast
    :start-with 'brekky
    )

;;; Assert that the following clauses belong to the breakfast grammar
(in-grammar 'breakfast)

;;; Now the grammar clauses
(defclause brekky ::= (one-of <fried-breakfast>
                              <continental-breakfast>
                              <cereal-breakfast>)
          :question "Would you like a fried, continental, or cereal breakfast ?"
          :comment "CHOOSING YOUR BREAKFAST...~%~%A fried breakfast consists of:~% -
fruit-juice~% - bacon & eggs~% - beverage.~%~%A continental breakfast consists of:~% -
fruit juice~% - cereal~% - toast & preserve~% - beverage.~%~%A cereal breakfast consists
of:~% - fruit juice~% - cereal~% - beverage."
  )

(defclause fried-breakfast ::= (seq <fruit-juice>
                                    <bacon-and-eggs>
                                    <beverage>))

(defclause continental-breakfast ::= (seq <fruit-juice>
                                          (seq <bread> <filling>)
                                          <beverage>)
          )

(defclause cereal-breakfast ::= (seq <fruit-juice>
                                     <cereal>
                                     <beverage>))

(defclause fruit-juice ::= (one-of 'orange-juice
                                   'grapefruit-juice
                                   'pineapple-juice)
          :question "What kind of fruit juice would you like ?"
          )

(defclause cereal ::= (one-of 'Kornflakes 'Nice-krispies 'Weetybix)
          :question "What kind of cereal would you like ?"
          )

(defclause bread ::= (one-of 'white-roll 'granary-roll 'brown-toast 'white-toast 'cr-
oissants)
          :question "What kind of bread would you like ? "
          :comment "(You may choose a sweet or savoury filling.)"
          )

(defclause filling ::= (one-of 'ham 'salami 'cheese 'jam 'marmalade)
          :question "Which of the fillings would you like ? "
          )

(defclause beverage ::= (one-of 'coffee 'tea 'fruit-tea)
          :question "What would you like to drink ? "
          )

(defclause bacon-and-eggs ::=
  (seq 'bacon
       (optional <eggs>
                 :question "Would you like eggs ? ")
       'sausage
       (optional 'black-pudding
                 :question "Would you like black pudding ? ")
       'tomato)
  )

(defclause eggs ::=
  (seq <number-of-eggs> <egg-type> 'eggs)
  )

(defclause number-of-eggs ::= (one-of 1 2)
          :question "How many eggs would you like?"
          )

(defclause egg-type ::= (one-of 'sunny-side-up 'poached 'scrambled)
          :question "How would you like your eggs to be cooked?"
          )
```

Figure 57: A Grammar for Acquiring Breakfast Orders

options are introduced by the grammar's top clause, and the details are provided by grammar subclauses. For example, the top-clause, `brekky`, makes reference to the `<fried-breakfast>` clause, and the `fried-breakfast` clause states that a fried breakfast consists of fruit juice, bacon & eggs, and a beverage. In the grammar, the reference to fruit juice is also a non-terminal symbol (`<fruit-juice>` rather than `'fruit-juice`). The `fruit-juice` clause provides a choice of three different juices, represented by the terminal symbols `orange-juice`, `grapefruit-juice`, and `pineapple-juice`. The acquisition of a choice of fruit juice therefore returns one of these symbols as its result. Acquiring the whole grammar returns a list structure whose composition is determined by the shape of the grammar and the use of the `seq` operator. The `seq` operator returns a sequence of given grammar terms in the same way that LISP's list function returns a list of given LISP terms. When a clause makes reference to a non-terminal symbol, its value is also returned as a sequence.

Here is an example acquisition session using the grammar of Figure 57:

```
>(cockatoo)
CHOOSING YOUR BREAKFAST...

A fried breakfast consists of:
 - fruit-juice
 - bacon & eggs
 - toast
 - beverage.

A continental breakfast consists of:
 - fruit juice
 - cereal
 - toast & preserve
 - beverage.

A cereal breakfast consists of:
 - fruit juice
 - cereal
 - beverage.

Would you like a fried, continental, or cereal breakfast ?
The possible values are:
  A.  <FRIED-BREAKFAST>
  B.  <CONTINENTAL-BREAKFAST>
  C.  <CEREAL-BREAKFAST>
Which value? : a

What kind of fruit juice would you like ?

The possible values are:
  A.  ORANGE-JUICE
  B.  GRAPEFRUIT-JUICE
```

```
  C.  PINEAPPLE-JUICE
Which value? : b

Would you like eggs ? y

How many eggs would you like?

The possible values are:
  A.  1
  B.  2
Which value? : 1

How would you like your eggs to be cooked?

The possible values are:
  A.  SUNNY-SIDE-UP
  B.  POACHED
  C.  SCRAMBLED
Which value? : a

Would you like black pudding ? n

What would you like to drink ?

The possible values are:
  A.  COFFEE
  B.  TEA
  C.  FRUIT-TEA
Which value? : coffee

(GRAPEFRUIT-JUICE (BACON (1 SUNNY-SIDE-UP EGGS) SAUSAGE TOMATO)
COFFEE)
>
```

Notice that the part of the grammar which asks whether the user would like eggs, and if so, their number and cooking method, can be improved. Currently, three questions are posed to the user; but this can be reduced to two (in a scalable way[10]) by employing the constraint mechanisms of COCKATOO. The trick is to ask how many eggs are required, and include zero as an allowable value. If zero is chosen, however, the question of how to cook the eggs should not arise. This behaviour is achieved by the following three clauses:

```
(defclause eggs ::=
  (let ((number-of-eggs (find-clause 'number-of-eggs))
        (egg-type (find-clause 'egg-type)))

    (assert!
```

10. Acquiring an egg specification in the given grammar could be reduced to one question with seven possible responses, but this approach would not be scalable if the number of eggs and ways of cooking them were to change. As these two variables increase, the number of possible responses grows with their product. So even with small changes to these variables, the user could become bewildered by the number of possible responses to the single question. Alternatively, there may be insufficient screen space for displaying the possible responses.

```
      (impliesv (equalv (acquired-valuev number-of-eggs) 0)
                (equalv (acquired-valuev egg-type) 'none))
            )

    (assert!
      (impliesv (not-equalv (acquired-valuev number-of-eggs) 0)
                (not-equalv (acquired-valuev egg-type) 'none))
            )
    (seq number-of-eggs egg-type 'eggs)
    )
  )

(defclause number-of-eggs ::= (one-of 1 2)
    :question "How many eggs would you like?")

(defclause egg-type ::=
    (one-of 'none 'sunny-side-up 'poached 'scrambled)
    :question "How would you like your eggs to be cooked?")
```

The first implication in the `eggs` clause ensures that if no eggs are required, then the variable `egg-type` immediately becomes bound to `'none`. The second implication ensures that if some eggs are required, then `'none` is not an admissible value for the `egg-type` variable.

7.5.2 Acquiring a Simple Fact Base

The problem solving scenario presented in chapter 6 described two problem solvers: a dish-designer and a scheduler. The output of the dish designer could be used as input to the scheduler. Unfortunately, no other means was provided for generating the required input to the scheduler. If we wanted to investigate the scheduling of a task which was *not* the output of the designer, we would either have to modify the configuration of the designer so that the 'right' design was generated, or edit the design "by hand" with a text editor. With a suitably defined grammar, such as that shown in Figure 58, COCKATOO can provide a more comfortable alternative.

The following shows an example interaction with COCKATOO running this grammar. The small fact base which is generated has been output to the terminal, but could equally easily have been output to a file.

```
> (load "task")
; Loading C:\Allegro\Cockatoo\task.gmr
T
> (cockatoo)

What would you like as the main component of the dish?
```

```
;;; Define the name space of the grammar
(defgrammar 'task
    :start-with 'task-file)

;;; Grammar starts here!
(defclause task-file ::= (seq <header> <subtask-rel> <tasknames>))

(defclause header ::= t
  :postproc (lambda(x) (format nil "# This file acquired by COCKATOO on ~a~%" (today)))
  )

(defclause subtask-rel ::= t
  :postproc
    (lambda(x)
     (format nil "~%~%defrelation subtask-of conceptref task, conceptref task;~%~%"))
  )

(defclause tasknames ::= (seq <main-component> <carbohydrate> <vegetable>)
  :postproc (lambda(x)
              (concatenate 'string
                (format nil "~%~%deffacts ")
                (format nil "subtask-of(PREPARE-~a, ROOT-TASK),~%" (first x))
                (format nil "subtask-of(PREPARE-~a, ROOT-TASK),~%" (second x))
                (format nil "subtask-of(PREPARE-~a, ROOT-TASK);~%~%" (third x))))
  )

(defclause main-component ::= (one-of 'lamb-chop 'gammon-steak 'sausages 'chicken-kiev
'plaice 'quorn-taco)
  :question "What would you like as the main component of the dish?")

(defclause carbohydrate ::= (one-of 'jacket-potato 'french-fries 'rice 'pasta)
  :question "What would you like as the carbohydrate component of the dish?")

(defclause vegetable ::= (one-of 'runner-beans 'carrots 'cauliflower 'peas 'sweetcorn)
  :question "Which vegetable would you like?")
```

Figure 58: A Grammar for Acquiring a Simple Fact Base

```
The possible values are:
  A.  LAMB-CHOP
  B.  GAMMON-STEAK
  C.  SAUSAGES
  D.  CHICKEN-KIEV
  E.  PLAICE
  F.  QUORN-TACO
Which value? : a

What would you like as the carbohydrate component of the dish?

The possible values are:
  A.  JACKET-POTATO
  B.  FRENCH-FRIES
  C.  RICE
  D.  PASTA
Which value? : a

Which vegetable would you like?

The possible values are:
  A.  RUNNER-BEANS
  B.  CARROTS
  C.  CAULIFLOWER
  D.  PEAS
  E.  SWEETCORN
Which value? : d
```

```
# This file acquired by COCKATOO on 11-AUG-99

defrelation subtask-of conceptref task, conceptref task;

deffacts subtask-of(PREPARE-LAMB-CHOP, ROOT-TASK),
subtask-of(PREPARE-JACKET-POTATO, ROOT-TASK),
subtask-of(PREPARE-PEAS, ROOT-TASK);

NIL
>
```

7.5.3 Acquiring a Case Base of Oil Well Drilling

A possible application for COCKATOO in the oil industry is the acquisition of knowledge to support case-based reasoning for oil well drilling. One of the major problems in this area is the selection of the best drill bit for cutting through any given set of rock strata. Before any automated reasoning can be applied to the bit selection task, however, a case base of past experience must be acquired.

COCKATOO could be used for acquiring this case base, as illustrated by the grammar structure shown in Figure 59 (a real application would expand the choice of rock types, and acquire other ancillary information). The following is an example of a short acquisition session using that grammar:

```
> (cockatoo)

    WELL DRILLING CASE BASE ACQUISITION
    -----------------------------------

Acquiring a new case...

A case consists of the drill bit used, together with the depth of each
rock stratum encountered.

Enter the name of the contact person for any questions relating to this case.

What was the first name of the contact person?
Input a value: fred

What was the last name of the contact person?
Input a value: dibnah

Which bit was used for drilling?

The possible values are:
  A.  BIT-TYPE-8765
  B.  BIT-TYPE-8760B
  C.  BIT-TYPE-8543
Which value? : b

Acquiring the rock composition of the different strata, starting from ground level.

What was the rock type?

The possible values are:
  A.  CHALK
  B.  CLAY
  C.  SANDSTONE
  D.  GRANITE
Which value? : c
```

```
(defgrammar drilling
    :start-with 'cases)

(in-grammar 'drilling)

(defclause cases -> (repeat+ <case>
                             :question "Another case? ")
           :comment "    WELL DRILLING CASE BASE ACQUISITION~%    ------------------
-----------------")

(defclause case ->
  (seq <contact>
       <drill-bit>
       <well-strata>
       (optional <remark>
                 :question "Any further remarks on this case? "))
  :comment "Acquiring a new case...~%~%A case consists of the drill bit used, together
with the depth of each ~%rock stratum encountered.")

(defclause contact -> (seq 'contact <contact-first-name> <contact-last-name>)
          :comment "Enter the name of the contact person for any questions relating
to this case.")

(defclause contact-first-name -> (a-symbolv)
           :question "What was the first name of the contact person?")

(defclause contact-last-name -> (a-symbolv)
           :question "What was the last name of the contact person?")

(defclause well-strata ->
  (repeat+ <stratum>
           :question "Another stratum? ")
  :comment "Acquiring the rock composition of the different strata, starting from
ground level.")

(defclause stratum ->
  (seq 'stratum :rock <rock> :depth <depth>))

(defclause rock -> (one-of 'chalk 'clay 'sandstone 'granite)
           :question "What was the rock type?")

(defclause depth -> (an-integer-betweenv 1 2000)
           :question "What was the depth (in metres) of this rock stratum?")

(defclause drill-bit -> (seq 'bit-type
                             (one-of 'bit-type-8765 'bit-type-8760b 'bit-type-8543))
           :comment "Which bit was used for drilling?")

(defclause remark -> (a-stringv)
           :comment "Please enter your remarks as a string.")
```

Figure 59: A Grammar for Acquiring a Case Base of Oil Well Drilling

```
What was the depth (in metres) of this rock stratum?
Input a value: 100

Another stratum? y

What was the rock type?

The possible values are:
  A.  CHALK
  B.  CLAY
  C.  SANDSTONE
  D.  GRANITE
Which value? : granite

What was the depth (in metres) of this rock stratum?
Input a value: 150

Another stratum? n

Any further remarks on this case? y

Please enter your remarks as a string.
Input a value: "The bit suffered excessive wear during the latter stages of drilling"

Another case? n
```

```
(((CONTACT FRED DIBNAH) (BIT-TYPE BIT-TYPE-8760B)
  ((STRATUM :ROCK SANDSTONE :DEPTH 100)
   (STRATUM :ROCK GRANITE :DEPTH 150))
  "The bit suffered excessive wear during the latter stages of drilling"))
>
```

7.5.4 Acquiring a Set of Simple Rules

This example demonstrates that COCKATOO can be used to acquire rule sets such as the following simple example (first given on page 32 of Chapter 2):

```
IF BarkColour is silvery_white
THEN TreeName is silver_birch;

IF BarkTexture is smooth AND BarkColour is grey
THEN TreeName is beech;

IF BarkTexture is fissured AND BarkColour is brown
THEN TreeName is oak;
```

Note that the exact syntax of the rule set depends on the target inference engine or interchange format. In CKRL, for example, the same rule set might look as follows:

```
DEFRULE silver_birch
  WHEN (conceptref tree ?x)
  IF (?x.barkColour EQ silvery_white)
  THEN (?x.name = silver_birch)

DEFRULE beech
  WHEN (conceptref tree ?x)
  IF ((?x.barkTexture EQ smooth) AND (?x.barkColour EQ grey))
  THEN (?x.name = beech)

DEFRULE oak
  WHEN (conceptref tree ?x)
  IF ((?x.barkTexture EQ fissured)
      AND (?x.barkColour EQ brown))
  THEN (?x.name = oak)
```

A grammar to acquire such a rule set is given in figure 60. Note that the grammar defines and uses types that describe admissible antecedents and consequents of simple CKRL rules.

The following is an example acquisition session using this grammar:

```
USER(26): (cockatoo)

Acquiring a set of rules...

What is the name of the rule?
Input a value: silver_birch

Which concept does the rule concern?
Input a value: tree

What is the condition for the rule to fire?
```

```
;;; Define a grammar called rule-set whose top-level clause is also rule-set.
(defgrammar rule-set
    :start-with 'rule-set)

;;; Assert that the following clauses belong to the rule-set grammar
(in-grammar 'rule-set)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define the types associated with rules
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Type for a rule antecedent
(deftype antecedent ()
  '(and list (satisfies antecedentp))
  )

;;; Condition associated with the antecedent type
(defun antecedentp(x)
  ;; Assumes that x is a list
  (assert (listp x))
  (cond
   ;; Most boolean expressions have three components
   ((= (length x) 3) (member (second x) '(and or eq < > <= >=)))
   ;; Except negations
   ((= (length x) 2) (member (car x) '(not conceptof)))
   ;; Otherwise cannot be an antecedent
   (t nil)
   )
  )

;;; Type for a rule consequent
(deftype consequent ()
  '(and list (satisfies consequentp))
  )

;;; Condition associated with the consequent type
(defun consequentp(x)
  ;; Assumes that x is a list
  (assert (listp x))
  ;; Expect a conclusion
  (or (and (= (length x) 3) (equal (second x) '=)) ; derive a value
      (and (= (length x) 2) (listp (second x)))    ; assert a relation
      )
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Grammar clauses start here...
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclause rule-set ::= (repeat+ <rule>
  :question "Acquire another rule? ")
  :comment "Acquiring a set of rules..."
  )

(defclause rule ::= (seq 'defrule <rule-name>
                         (seq 'when (seq 'conceptref <scope> '?x)
                                    (seq 'if <antecedent>
                                         'then <consequent>)))
  )

(defclause rule-name ::= (a-symbolv)
  :question "What is the name of the rule? ")

(defclause scope ::= (a-symbolv)
  :question "Which concept does the rule concern? "
  )

(defclause antecedent ::= (a-typed-varv 'antecedent)
  :question "What is the condition for the rule to fire? ~%(Use ?x to represent the
concept of interest.)"
  )

(defclause consequent ::= (a-typed-varv 'consequent)
  :question "What is the consequent? ")
```

Figure 60: Grammar for Acquiring a Simple Rule Set

```
(Use ?x to represent the concept of interest.)
Input a value: (?x.barkColour eq silvery_white)

What is the consequent?
Input a value: (?x.name = silver_birch)
Acquire another rule? y

What is the name of the rule?
Input a value: beech

Which concept does the rule concern?
Input a value: tree

What is the condition for the rule to fire?
(Use ?x to represent the concept of interest.)
Input a value: ((?x.barkTexture eq smooth) and (?x.barkColour eq grey))

What is the consequent?
Input a value: (?x.name = beech)
Acquire another rule? y

What is the name of the rule?
Input a value: oak

Which concept does the rule concern?
Input a value: tree

What is the condition for the rule to fire?
(Use ?x to represent the concept of interest.)
Input a value: ((?x.barkTexture eq fissured) and (?x.barkColour eq brown))

What is the consequent?
Input a value: (?x.name = oak)
Acquire another rule? n

((DEFRULE SILVER_BIRCH
  (WHEN (CONCEPTREF TREE ?X)
    (IF (?X.BARKCOLOUR EQ SILVERY_WHITE)
        THEN
      (?X.NAME = SILVER_BIRCH))))
 (DEFRULE BEECH
  (WHEN (CONCEPTREF TREE ?X)
    (IF ((?X.BARKTEXTURE EQ SMOOTH) AND (?X.BARKCOLOUR EQ GREY))
        THEN
      (?X.NAME = BEECH))))
 (DEFRULE OAK
  (WHEN (CONCEPTREF TREE ?X)
    (IF ((?X.BARKTEXTURE EQ FISSURED) AND (?X.BARKCOLOUR EQ BROWN))
        THEN
      (?X.NAME = OAK)))))
```

7.6 Discussion

This chapter has argued the value of a declarative specification of the knowledge to be acquired, and introduced the COCKATOO tool which acquires pieces of knowledge by following a context-free (EBNF-like) grammar. This approach offers enhanced readability, eased maintenance, and a reduced initial development effort compared with the construction of multiple customised tools for different domains. Augmenting a context-free grammar with constraints increases both the expressiveness and conciseness of the notation. The power of the tool which interprets the notation is also increased because in some situations its behaviour can be altered by the user's responses to questions. Conciseness of the notation is improved because admissible values do not always have to be detailed down to the level of individual characters or symbols.

Readability and Maintainability of Knowledge Specifications

It is important for a knowledge specification to be easily readable so that persons other than the KA tool developer can understand it. Readable specifications tend to be easier to write, discuss, and maintain.

COCKATOO has two main features which enhance both the readability and maintainability of its knowledge specifications. Firstly, it has developed an approach based on the use of (EBNF-like) formal grammars, which are a well-known type of formal specification, and in widespread use. Secondly, COCKATOO makes a clear separation between the knowledge specification and the acquisition engine which acquires the knowledge. This leads to concise specifications which contain only pertinent material, as well as a general purpose acquisition tool which is reusable across domains. By way of contrast, consider a custom-tailored acquisition tool which embeds the knowledge specification within its program's source code. Such a tool would not be reusable across different problem domains, and even with optimal coding style, only those with knowledge of the programming language would be able to understand the specification.

Development Effort

COCKATOO already provides the means by which to specify the required knowledge at a "high level". That is, during (grammar) development, the knowledge engineer can concentrate on the nature of the knowledge to be acquired, rather than the program which acquires it. Grammar development using COCKATOO is a (cyclic) refinement process which includes the following chronological stages.

Knowledge Analysis – Deciding which knowledge "elements" (pieces of knowledge) need to be acquired.

Grammar Construction – Devising a grammar structure which captures the basic knowledge requirements in terms of those elements and their multiplicities (e.g., one-one, one-many, many-many).

Adding Constraints – An optional stage to enhance the grammar with *constraints*. When performed, the aim of the stage is to

- remove unwanted or nonsensical input combinations from the specification, and
- eliminate redundant questions.

Embellishment – Embellishing the grammar with questions and comments to improve the user-interface.

Notice that the user-interface is not considered until the final stage of development, reflecting the attention paid to the correctness of the knowledge specification in the early stages.

The examples presented throughout this chapter have demonstrated COCKATOO's flexibility, and shown that it can be reused in many different domains. COCKATOO can also be configured *quickly* for use in a new domain. For example, each of the three example grammars presented in section 7.5 were developed (and refined) in less than a day! The main reason for the ease with which COCKATOO can be reused is the clear separation between the data that drives knowledge acquisition (the grammar) and the more generic tool which processes the data (the COCKATOO acquisition engine). COCKATOO is, in effect, a knowledge acquisition *shell* which supports the building of custom-tailored KA tools.

Although it was developed to acquire *knowledge bases* for use within the MUSKRAT toolbox, a KA tool such as COCKATOO has the potential to be applied to a very wide range of application domains. Not only can it acquire simple knowledge elements, it can manage complex constraint relationships between them, and postprocess user inputs for further compatibility with other tools. With regard to COCKATOO's suitability for different acquisition tasks, it could be used in most situations which involve a substantial amount of textual or symbolic user input. It is particularly well-suited to configuration (or limited design) tasks, such as the breakfast ordering example (section 7.5.1). For these tasks, the building blocks of the design are well-known, but their combinations may be explored. Although all the design decisions are made by the human

user, the output is nevertheless constrained to be within the "space" specified by the grammar.

7.6.1 Related Work

COCKATOO is an automated knowledge elicitation tool, but differs considerably from current knowledge elicitation tools based on repertory grids, sorting, and laddering (see "Automated Knowledge Elicitation" on page 21), because they cannot be tightly coupled with an application. The knowledge acquired using these tools must be post-processed "by hand" before they can be used by a problem solver or other application program. COCKATOO, on the other hand, provides a very general post-processing facility which allows the acquired knowledge to be packaged in a form suitable for subsequent use. In a sense, knowledge bases acquired by COCKATOO are the "take-away meals" of the knowledge acquisition world.

Note that GDMs (already discussed under "Critiquing the MLT Consultant" on page 87) use grammars for a different purpose to COCKATOO. COCKATOO uses a grammar to guide the acquisition of *domain knowledge* from a *domain expert*, such that the knowledge can be used by an *existing* problem solver. GDMs, on the other hand, apply grammars to assist *knowledge engineers* with *task decomposition* when *building* a knowledge-based system. The purpose of a GDM grammar is to guide the knowledge engineer to classify the task(s) at hand, so that an appropriate knowledge elicitation tool can be selected. Thus, the grammars of the two approaches describe sentences of quite different natures: a COCKATOO sentence describes a domain structure, whereas a GDM sentence describes the decomposition of a task into subtasks and their respective types. It may also help to consider the meanings of the terminal symbols in each of the two formalisms. A terminal symbol of a COCKATOO grammar represents a domain concept or value; a terminal symbol of a GDM represents a task type whose association with a knowledge elicitation tool is known. The two approaches are complementary, and could even be used together – a terminal node of the GDM grammar could be associated with COCKATOO as the most appropriate elicitation tool for the node's task type.

It is interesting to compare COCKATOO with the **Protégé** project and, in particular, the **Maître** tool (Eriksson et al., 1994). Protégé, like COCKATOO, is a general tool (a "knowledge acquisition shell") whose output is in a format that can be read by other programs (**CLIPS** expert systems). Also, before using Protégé to acquire knowledge, the knowledge engineer must first configure it with an "Ontology Editor" subsystem, called *Maître*. This tool enables the knowledge engineer to define an ontology which is then used as the basis for knowledge acquisition. The ontology plays the same role in Protégé as the constraint-augmented grammar in COCKATOO – it specifies the knowledge to be acquired. Another subsystem of Protégé, called **Dash**, can be used interactively to define a graphical user-interface for the KA tool. Although such graphical user interfaces are very appealing, I do not believe that Protégé has the same knowledge specification power as COCKATOO, since it is not supported by an underlying constraints engine.

The idea of minimising the number of questions asked of the user was inspired by the questioning techniques of MOLE (Eshelman, 1988), and the intelligent mode of the MLT Consultant (see "The MLT Consultant" on page 84). However, the approach taken by COCKATOO is different from both of these systems. MOLE reduces the amount of questioning by making intelligent guesses about the values of undetermined variables, and subsequently requesting the user's feedback. MLT Consultant uses an information theoretic measure to guide the order in which questions are asked. Those questions which gain the most information are asked first. In contrast, COCKATOO uses local propagation techniques to identify redundant questions.

A so-called *Adaptive Form* (Frank & Szekely, 1998) is a graphical user-interface for acquiring structured data that modifies its appearance depending on the user's inputs. For example, a form for entering personal details would only show a field for entering the spouse's name if the user had entered *married* in the *marital status* field. Although this kind of behaviour is very similar to the reactive knowledge acquisition of COCKATOO, Adaptive Forms are driven by (context-free and regular-expression) grammars alone, and do not support more complex constraints. The system uses *look-ahead parameters* to decide which unbound fields to display at any given time. I also note that Frank and Szekely extended their grammar notation to include 'labels', which have the

same function as the questions of COCKATOO. They do not provide the equivalent of comments, name spaces (see Appendix D), or post-processing. The Amulet system (Myers et al., 1997) *does* use a constraint solver to manage its user-interface, but employs it to control the positioning and interactions of graphical objects, rather than to support knowledge acquisition.

Finally, I acknowledge that in other research fields, constraint-based approaches have been combined with generative grammars. For example in the field of Computer-Aided Design, Carlson (1993) describes Grammatica, a tool that combines grammars and a CLP(R) constraint solver, and can be used to describe and explore architectural design spaces[11]. Brown, McMahon and Williams (1994) describe an approach that combines constraints with a formal grammar, and show how it can be applied to the manufacture of objects on a lathe. In the field of natural language processing, constraints have been combined with grammars and used, for example, to describe the agreement between nouns and verbs (e.g., Shieber, 1986, p. 25).

7.6.2 Limitations

Backtracking

COCKATOO currently does not allow the user to backtrack from a given user input, and try something else. Once an input has been entered, the user is committed to that value and cannot change it later. This is a serious limitation because not only does it not allow for the mistyping of inputs, it also prevents the user from experimenting with the COCKATOO grammar and using it as a kind of 'what-if' scenario. Of course, COCKATOO can always be aborted and the acquisition restarted from the beginning, but a more flexible backtracking capability is a desirable feature which should be addressed in a later version of the system. Ideally, at each stage of the acquisition process the user should be given the option to go back to the previous stage, retract the previous input, and input a new value. The problem is that retracting an input is not simple, because the constraint-based assertions associated with that input may already have caused propagation to other constraint variables. To retract an input, one must be able to

11. In particular, Carlson shows how the approach can be applied to the description of Gothic-style windows and arches.

recover the states of *all* constraint variables before the assertion was made. One option to achieve this functionality is to record the states of all constraint variables before each user input; another option is to record the changes that occur after each user input, so that they may be reversed. Ideally, however, SCREAMER+ would provide a retract facility of its own (see discussion on page 118).

Recursion

COCKATOO makes no checks for circularity in the supplied grammar. Therefore the following grammar is valid COCKATOO syntax, but leads to a stack overflow and entry into the LISP debugger:

```
(defclause circle ::= (seq <circle> 'end))
```

This is not a serious limitation, since a circular grammar is indeed an error, but COCKATOO should be able to deal with this type of error in a more user-friendly manner. It should check the grammar and generate an appropriate warning if the grammar exhibits its circularity.

Inter-KB Propagation

There are two types of propagation that should be accommodated by a KA tool for MUSKRAT; namely *intra-KB propagation* and *inter-KB propagation*. In the former case, knowledge is propagated from one part of a knowledge base to a different part of the *same* knowledge base. In the latter case, knowledge is propagated from one knowledge base to a *different* knowledge base.

COCKATOO already provides an adequate mechanism for intra-KB propagation through the functions `find-clause` and `acquired-valuev`. The function `find-clause` can be used from within a grammar to search for a known clause, and `acquired-valuev` can be used to make assertions about the value that a clause acquires. When the functions are combined, they can have the effect of intra-KB propagation. Consider the following grammar, which acquires an order for a three-course meal. It propagates knowledge from one part of the order to other parts of the same order at acquisition time. If the user chooses a pasta starter, then spaghetti is not offered as a main-course. If the user chooses pizza as the main-course, then bread pudding is

not offered as a dessert. Although the example may appear trivial, the same mechanisms can be applied to larger problems where intra-KB propagation is required.

```
(defgrammar intra-kb
    :start-with 'meal)

(defclause meal ::=
  (let (
        (starter (find-clause 'starter))
        (main-course (find-clause 'main-course))
        (dessert (find-clause 'dessert))
        )

    (assert! (impliesv (equalv (acquired-valuev starter) 'pasta)
                       (not-equalv (acquired-valuev main-course) 'spaghetti)))

    (assert! (impliesv (equalv (acquired-valuev starter) 'garlic-bread)
                       (not-equalv (acquired-valuev main-course) 'pizza)))

    (assert! (impliesv (equalv (acquired-valuev main-course) 'pizza)
                       (not-equalv (acquired-valuev main-course) 'bread-pudding)))

    (seq starter main-course dessert)
    )
  )

(defclause starter ::= (one-of 'pasta 'soup 'garlic-bread)
           :question "What would you like as a starter?")

(defclause main-course ::= (one-of 'spaghetti
                                   'pizza
                                   'meat-pie)
           :question "Which main course would you like?"
           )

(defclause dessert ::= (one-of 'bread-pudding 'fruit-salad)
           :question "Which dessert would you like?")
```

For inter-KB propagation, a mechanism must be provided by which the knowledge contained in an *existing* knowledge base is made available to COCKATOO for acquiring a different, but related, knowledge base. In its crudest form, this could involve reading expressions from a file into some global variables which are subsequently accessed by the clauses of a COCKATOO grammar. This idea will work already, but is unsatisfactory because a program would have to be written to read each knowledge base that might potentially influence the KB to be acquired. A neater concept should be developed to solve this problem.

Chapter 8

Conclusions and Further Work

'Finally, in conclusion, let me say just this.'

Peter Sellers

Chapter Summary

This dissertation introduced the notion of *fitness-for-purpose*, applied to a problem solver, a set of knowledge bases (assigned problem-solving roles), and a goal. It presented an approach to the automatic determination of fitness-for-purpose, and a knowledge acquisition tool which can be configured to reject knowledge that is not fit for a particular problem solving role or purpose. Both ideas have been supported by an extension to the LISP-based constraints package, SCREAMER. This chapter reviews the dissertation by summarising these ideas, explaining the contributions which the work has made, and proposing some promising extensions.

8.1 Summary

The problems discussed in this dissertation arose from a desire to build an advanced knowledge acquisition toolbox, called MUSKRAT (Multistrategy knowledge refinement and acquisition toolbox). MUSKRAT acknowledges that knowledge bases are acquired to solve problems, and that when a new problem arises, automatically generated advice on the reusability of available knowledge bases could save a toolbox user much time. A major requirement of the MUSKRAT Advisor was therefore that it should be able to perform the following three actions (first given on page 93).

★ **Determine** whether the knowledge required for solving a given problem is available, and in a form suitable for immediate use; [Action 1]

If the required knowledge is not available and immediately applicable, then either

★ **Acquire** knowledge *ab initio* such that it meets the requirements of the problem solver; [Action 2]

or

★ **Modify** existing knowledge to meet the requirements of the problem solver. [Action 3]

Each of these three actions is a difficult task which would normally require intelligence if solved by a human. The dissertation addressed the first two of these actions, employing constraint technology for the solutions to both tasks. It was discovered that in both cases, constraint technology yielded an elegant notation for expressing the problem, and provided efficient mechanisms for solving them. Since much of contemporary knowledge acquisition research is concerned with getting the 'right' knowledge for the task at hand, and constraint technology proved to be advantageous in my own work, the benefits of the technology to knowledge acquisition became a main theme of my work. I believe that the knowledge acquisition community can learn much by further embracing constraint technology.

My consideration of Action 1 demanded a clarification of the phrase 'a form suitable for immediate use'. I therefore defined the *fitness for purpose* of a problem solving configuration which consists of a single problem solver, knowledge bases, and a goal. I argued that fitness for purpose is related to the notion of competence, and that the *plausibility* of a configuration is a tractable approximation to fitness for purpose. I noted that any test for the plausibility of a configuration should be in some sense 'easier' than running a problem solver. Constraint programming offered a promising approach to the implementation of plausibility tests because it enables a knowledge engineer to write a declarative meta–problem-solver, which makes statements about the relationship between the inputs and outputs of the ground-level problem solver. When this knowledge is coupled with knowledge of a goal which the user is trying to satisfy, it reduces the space of plausible results in a way that can lead to early failures. A failure

denotes the *implausibility* of the task. In addition, meta–problem-solvers can accept the *outputs of other meta–problem-solvers* as their *inputs*, and propagate this input knowledge to their respective outputs. This ability allows us to reason about the plausibility of a goal which is to be solved using a *combination* of problem solvers.

To satisfy the requirements of Action 2, I devised a generic knowledge acquisition tool which acquires knowledge by expanding the terms of a context-free grammar. This is a declarative notation which is very easy to learn (especially for those already familiar with formal grammars). However, limitations in the expressiveness of a purely grammar-driven approach led me to enhance the tool to accommodate much of the work I had already done using constraint technology. For the purposes of specifying the knowledge which is to be acquired, a *constraint-augmented grammar* has the advantages of improved conciseness and additional expressive power compared to the purely grammar-driven approach.

8.2 Contributions

This section summarises the contributions of the work described in this dissertation, which include the following.

- A notation for representing the capabilities of problem solvers, together with their relationships to the contents of knowledge sources and a desired goal. The identification of mismatches between these entitties contributes to the assessment of a knowledge base's reusability. (The constraint-based notation is described in chapter 5 and appendix A; examples of capability descriptions using that notation are given in section 6.3 of chapter 6, from page 176. The approach to identifying mismatches is described in section 6.2.3 of chapter 6, from page 166.)
- An investigation of the relationship between 'economical' problem solving and the description of problem solving methods. (See section 4.4.2 from page 93 of chapter 4, and section 6.2 from page 160 of chapter 6.)

- Significant progress towards the provision of a framework which unifies problem solving, knowledge acquisition and knowledge transformation/refinement. (Chapters 5, 6 & 7 work towards the MUSKRAT vision presented from page 86, in section 4.3 of chapter 4).
- The development of a domain-independent knowledge acquisition tool that can be configured to acquire knowledge to satisfy a given problem solving role. (See from page 98 for section 4.4.3 of chapter 4, from page 206 for sections 7.3 and 7.4 of chapter 7, and from page 330 for appendix D.)
- The further development of constraint technology through my extension to the SCREAMER package. (See from page 115 for section 5.3 of chapter 5, and from page 268 for appendix A.)

My notation for representing the capabilities of problem solvers, and the technology chosen for proving their properties differs considerably from the related works of Wielinga et al. (Wielinga, Akkermans & Schreiber, 1998), Fensel et al. (Fensel & Schönegge, 1997; Fensel & Schönegge, 1998), and Pierret-Golbreich (Pierret-Golbreich, 1998). The differences have evolved because I have been working towards a different, but associated, objective. Unlike the methodology of Wielinga et al., for example, I assume that problem solvers exist and seek to describe them as they are, rather than attempting to construct them as I would like them to be. Furthermore, since I am building an advisory system for novice users of problem solvers, there are strong requirements for *operational* descriptions, and a fully *automatic* proof technique. Pierret-Golbreich argues that formal methods should be used to describe problem-solving methods and that this offers a means to decide on a component's reusability. Since such methods are not operational, however, this approach cannot be applied to my problem. Fensel & Schönegge have operationalised their proof technique, but its powerful proof mechanisms are necessarily interactive (Fensel & Schönegge, 1997), and not suitable to be driven by novices. My notation, on the other hand, is declarative, operational, and sufficiently expressive to enable the automatic derivation of interesting properties of a problem solver and the knowledge it employs.

The contribution to the topic of knowledge base reuse is most closely related to Puppe's work (Puppe, 1998) and the Protégé project (Gennari, Cheng, Altman &

Musen, 1998). Puppe has realised, as do O'Hara et al. (O'Hara, Shadbolt & van Heijst, 1998) that as experts learn more about the domain, their perspectives on the task may change, and hence it is highly desirable that acquired knowledge can be used with a different algorithm from the one for which it was initially acquired. Puppe confines his attention to classification tasks, and notes that there are a number of different classification algorithms which process the same information in different ways. He discusses a number of algorithms including CATEGORICAL (which produces decision trees/tables), heuristic classifiers (which use heuristic rules) and case-based reasoners. Puppe's important insight was to reimplement several classifiers so that they have common data files which can be readily reused. In MUSKRAT, however, I have set up a more general framework in which a knowledge base initially used with a PSM for a classification task might be reused with a different PSM for synthesis or planning.

Protégé's long term goal is to build a tool-set and methodology for the construction of domain-specific KA tools and knowledge-based systems from reusable components. The project plans to develop a variety of methods and knowledge bases, all packaged as CORBA (Common Object Request Broker Architecture) components. Knowledge bases are made available on a server which uses the OKBC (Open Knowledge Base Connectivity) protocol, enabling developers to query the frame-based knowledge with functions such as `get-class-all-subs` to retrieve classes, `get-frame-slots` to retrieve slots, and `get-class-all-instances` to retrieve the instances of a class. A single method, *propose-and-revise*, is available within the framework to date, but there are several knowledge bases which have been used for the VT (elevator-configuration) task, U-Haul (truck selection), as well as the Ribosome and tRNA configuration tasks. In order for a particular method to reuse knowledge from a particular knowledge base a mediator must be manually encoded for each method/KB pair. However, Gennari et al. (Gennari, Cheng, Altman & Musen, 1998) argue that given the similarities in the representations of the knowledge bases, there will be many commonalities between the various mediators needed. This already reduces the workload required to produce the mediators, and the Stanford group envisages partially automating mediator production in the future. MUSKRAT's third case (see section 1 of this chapter) corresponds to the situation for which Protégé currently creates media-

tors. Whilst Protégé *assumes* that a mediator is necessary, MUSKRAT has evolved a test to determine whether an available knowledge base might be *directly* reusable for the given task. If this is the case, no mediation or adaptation is required.

The knowledge acquisition tool COCKATOO makes a contribution to the field of knowledge acquisition because it is able to acquire knowledge *ab initio* such that it meets the requirements of a problem solver. In this respect, it is similar to knowledge acquisition tools such as MOLE and SALT, which also acquire knowledge for a particular problem solver. However, whilst MOLE and SALT will only ever be able to acquire knowledge for *cover-and-differentiate* or *propose-and-revise* systems, respectively, COCKATOO places no restrictions on the target problem-solving method. Instead, COCKATOO is *configured* to acquire knowledge that meets the problem solver's requirements. This configuration takes the form of a constraint-augmented grammar. Although other tools, such as Protégé, also use a configuration approach, they do not apply constraint technology. Without this capability, I doubt that the distinction between acceptable and unacceptable inputs can be stated as precisely as with COCKATOO.

My extension to the SCREAMER package, driven by the requirement for a declarative and operational description of problem solvers and knowledge sources, has reached a level of expressiveness that contributes to the field of constraint technology. SCREAMER+ can be used to provide elegant solutions to many of the non-trivial problems cited as soluble by commercially available systems such as CHIP (Simonis, 1995) and ECLIPSE (Wallace, Novello & Schimpf, 1997). The LISP-based nature of SCREAMER+, however, has also helped it to gain advantages over its PROLOG-based counterparts, such as the ability to assert constraints on expressions which take *functions* as arguments. This is especially important in the context of this dissertation, because a function might be the realisation of a PSM. (But see also "Constraint Variables with Enumerated Domains of Functions", a topic of further work discussed on page 244.)

My investigation of economical problem solving has been motivated by the requirement to generate advice on the suitability of problem solvers "on the fly" at run-time. To meet this objective, I have proposed a form of approximate reasoning, and investigated the use of *abstraction* (Giunchiglia & Walsh, 1992). In general, abstraction is

recognised as a “black art” (Knoblock, 1991; Walsh 2000), and there are few examples of the *automatic* generation of abstractions[1]. My work currently supports the expression of hand-crafted abstractions, but can be extended to support automatically generated abstractions (see “Automatic Generation of Preconditions” on page 245).

The final contribution of MUSKRAT is progress towards a unified framework for problem solving, directed knowledge acquisition, and knowledge transformation. This aim is shared by the generalised directive models (GDM) work of O’Hara, Shadbolt and van Heijst (O’Hara, Shadbolt & van Heijst, 1998), but there are significant differences as their work takes place in the context of analysing source materials and attempting to co-evolve domain ontologies as well as the appropriate model for the task; in some cases a KA task is activated. In the case of MUSKRAT, the task to be solved and the problem solver to be used have already been identified. MUSKRAT seeks to discover whether existing knowledge base(s) can be used without change with the identified problem solver to solve a particular goal, whether it can be transformed before use, or whether it is necessary to acquire a completely new knowledge base using a KA tool. Other work which has sought to provide a common framework for problem solving and KA includes MOBAL (Morik, Wrobel, Kietz & Emde, 1993), VITAL (Motta, O’Hara & Shadbolt, 1996), and NOOS (Arcos & Plaza, 1994; Arcos & Plaza, 1997).

8.3 Further Work

The substantive work of this dissertation can be divided into three parts: ‘Constraint Handling in Common LISP’ (Chapter 5), ‘Determining Fitness for Purpose’ (Chapter 6), and ‘Acquiring Knowledge which is Fit for a Purpose’ (Chapter 7). The consideration of further work also adheres to these headings. In addition, it will reconsider the MUSKRAT vision described in Chapter 4 in the light of the work described here and more recent research directions in knowledge acquisition.

1. ABSTRIPS is one of the few examples.

8.3.1 Constraint Handling in Common LISP

Applying SCREAMER+ to Ship Design

Jayanta Majumder, of the University of Strathclyde, is currently evaluating SCREAMER+ for an application in ship design (Majumder, 2000). The proposed system assists in the ship design process with an integrated software environment that includes the following major components.

- An object-oriented *repository* that models the space of design parameters.
- A knowledge base of *design rules*. The rule base comprises three kinds of rules: *diagnostic*, *generative* and *triggering*. The *triggering* rules invoke different software tools that act upon the repository through a well-defined software interface.
- A *transaction command language* that uploads data to the repository.
- A *format specification meta-language* that is used to export data from the repository in different formats. This will help to link other software components to the system.

SCREAMER+ is being evaluated as a representation for the design rules. It is of particular interest because it combines flexible constraint representations with constraint solvers for both continuous[2] and finite domains. Conditional constraints using `ifv` will be evaluated as a possible approach to the representation of the triggering design rules.

Combining Constraints with Lazy List Evaluation

I believe it would be advantageous to combine the constraint programming of SCREAMER+ with lazy list evaluation. This includes the ability to store and propagate knowledge of a list *before its length is known*. SCREAMER+ cannot do this currently, despite my stated aims to propagate knowledge as early as possible. The problem is that some list manipulation functions produce results that contain no information about the lengths of their (list) arguments. For example, consider the expression *z* = `(carv` *x*`)`, in which *z* is constrained to be the head of the list *x*. As we have seen, as soon as *x*

2. SCREAMER already contained a solver for real-valued variables in closed ranges, and this has not been removed by SCREAMER+.

becomes bound, *z* can be computed, thus propagating knowledge in a "forward" direction with respect to the evaluation of the function. However, if *z* were to become bound before *x*, then the knowledge of the head of the list cannot be propagated to *x* unless the implementation allows lazy lists. For example, if *z* were to become bound to the symbol `'foo`, then *x* should acquire the value `'(foo ...)`, in which the tail is unknown. Then, if some other variable is constrained to be the `car` of *x*, it may acquire its value immediately, rather than having to wait until all the values of the list become bound.

The same idea applies to the function `nthv`. For example, if the third element of the list *x* is constrained to be `'foo`, i.e., *z* = `(nthv 2 x)` and *z* = `'foo`, then *x* should acquire the value `(? ? foo ...)`, where its length would already be known to be greater than or equal to 3. The advantage would be that `(third x)` could be evaluated without knowing the length of *x*, and propagation of this knowledge could therefore occur earlier than in the current implementation.

Within LISP, lazy lists could be implemented as an abstract data type with the same set of basic operations as conventional LISP lists. That is, the details of the representation are hidden from the lazy list's user, but a set of lazy list functions may operate on this type to return the desired results. Suppose the three most basic functions are called `lazy-cons`, `lazy-car` and `lazy-cdr`. Then if *x* is a list, these lazy implementations must uphold the condition that:

`(lazy-cons (lazy-car` *x*`) (lazy-cdr` *x*`))` = *x*

The data structure of the representation could consist of a set of mappings, one mapping for each known index of the list, as well as an integer constrained to hold the length of the bound list. For example, the list `'(? ? foo ? bar ...)` would be represented by the mappings $\{2 \rightarrow \texttt{foo}, 4 \rightarrow \texttt{bar}\}$ (assuming the indexing starts with zero), and the length would be an integer whose lower bound is 5. With such a representation, seeking the third element of the list (index 2) would mean searching the set of mappings for an index of 2. The corresponding value, `foo`, could be returned immediately without having to know the length of the list.

Constraint Variables with Enumerated Domains of Functions

Currently, domains of SCREAMER+ constraint variables are restricted to *data values*, such as LISP atoms, strings, and lists. However, it would not be difficult to modify the code to also allow LISP *functions* to be included in the enumerated domain of a constraint variable. This would be in keeping with the spirit of LISP, which allows functions to be treated as data values (and data values to be interpreted as functions). By allowing functions in the enumerated domain of constraint variables, it would be possible to constrain a data value to be the output of one of a set of functions. For instance, it could work as shown in the following (hypothetical) example:

```
;;; Create a constraint variable to hold a function
> (setq f (make-variable 'func))
[func]

;;; Assert the function to be one of the arithmetic operators
;;; i.e., addition, subtraction, multiplication, or division
> (assert! (memberv f (list #'+ #'- #'* #'/)))
[func function enumerated-domain:(#<Function +>
                                  #<Function ->
                                  #<Function *>
                                  #<Function />)]

;;; Constrain x to be the result of applying func to the
;;; arguments 2 and 3
> (setq x (applyv f '(2 3)))
[23 number]

;;; Constrain y to be the result of applying func to the
;;; arguments 1 and 2
> (setq y (applyv f '(1 2)))
[24 number]

;;; Constrain y to be 2
> (assert! (equalv y 2))
NIL
;;; Inspect the value of f
> f
<Function *>

;;; Inspect the value of x
;;; It should automatically become bound because the previous
;;; assertion caused the function f to become bound
> x
6
```

8.3.2 Determining Fitness for Purpose

Applying Fitness for Purpose to a Real-World Problem

The ideas of *fitness for purpose* have so far only been applied to a problem solving scenario in the culinary domain. This scenario, coupled with the qualitative evaluation of chapter 6, was sufficient to demonstrate that the approach can be operationalised. However, the usefulness of the approach would be better gauged through its application to a real-world problem. Moreover, further work of this kind is more likely to highlight the inadequacies of the method, and therefore provide the impetus to drive forward research.

Applying the fitness-for-purpose method to a real-world problem would also provide the opportunity to further investigate the domain/task dependencies of such tests. I believe that some fitness-for-purpose tests can only be used within a particular application domain (e.g., the Königsberg Bridges test in Graph Theory), whilst others are dependent only on the task (e.g., data requirements for preforming Heuristic Classification). Further investigation of this topic would seek to provide a clean separation between these aspects, so that the tests may be both reused and combined. One can imagine a plausibility test toolkit, which reduces the effort required for building meta–problem-solvers.

Another important aspect of fitness-for-purpose testing that could be evaluated empirically is the *quality* of a fitness-for-purpose test. Although one can theoretically analyse the trade-off between checking the plausibility of a goal and running a problem-solver to try to solve the goal, there are related issues that cannot be analysed theoretically. For example, how credible is the advice generated by an advisory system for its users, and how effective is it in changing its users' behaviour? Such effects can only be measured empirically.

Automatic Generation of Preconditions

The arithmetic example of chapter 4 (see page 94) showed how properties of *parity* in integer arithmetic can be used to assess the plausibility of an arithmetic equation. Arithmetic expressions were abstracted to their equivalent parity expressions before

the precondition was applied. Moreover, there was a *very close relationship between the precondition and the original arithmetic expression*. In such cases, it is possible for the precondition to be generated automatically (Robertson, 1999). This represents a class of tasks to be investigated.

As an example, suppose again that we would like to compute the plausibility of the arithmetic expression $22 \times 31 + 11 \times 17 + 13 \times 19 = 1097$. That is, the left-hand side forms the input to the meta–problem-solver, and the *goal* is that the expression plausibly evaluates to the right-hand side. We can first rewrite the expression into LISP's prefix notation, giving

```
(= (+ (* 22 31) (* 11 17) (* 13 19)) 1097)
```

Then, since the meta-level in this case applies a *different* algebra to the *same* numbers, the corresponding meta-level expression can be generated ("lifted") by replacing the ground-level operators (+, * and =) by their meta-level counterparts (meta+, meta* and meta=), giving

```
(meta= (meta+ (meta* 22 31) (meta* 11 17) (meta* 13 19)) 1097)
```

Using the constraint-based meta-level functions provided in Figure 61, the expression fails, signifying that the goal is inconsistent with the value of the arithmetic expression:

```
> (assert! (meta= (meta+ (meta* 22 31) (meta* 11 17) (meta* 13
19)) 1097))
Error: Attempt to throw to the non-existent tag FAIL
  [condition type: CONTROL-ERROR]

Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart)
[1] >
```

Moreover, the same functions can be used to identify *which* goals are plausible. In the example below, SCREAMER+ has identified that 1097 and 1109 are implausible as values of the arithmetic expression.

```
> (setq goal (a-member-ofv '(1108 1109 1116 1097)))
[327 integer 1097:1116 enumerated-domain:(1108 1109 1116 1097)]

> (assert! (meta= goal (meta+ (meta* 22 31) (meta* 11 17) (meta*
13 19))))
NIL

;;; Check that implausible values were removed from the domain
```

```
> goal
[327 integer 1108:1116 enumerated-domain:(1108 1116)]
>
```

The call to `meta=` represents the consistency check between the problem solver's output and the goal. Note that the precondition to apply to arithmetic expressions varies with the task instance, so it is *imperative* that the precondition be *automatically* computed as part of the meta–problem-solver.

```
(defun parity (x)
  (if (or (equal x 'even) (and (integerp x) (evenp x)))
      'even
    'odd)
  )

(defun parityv (x)
  (let ((z (a-member-ofv '(even odd))))
    (assert! (equalv z (funcallv #'parity x)))
    z)
  )

(defun madd (x y)
  (let ((z (a-member-ofv '(even odd))))
    (assert! (ifv (equalv (parityv x) (parityv y))
                  (equalv z 'even)
                  (equalv z 'odd)))
    z)
  )

(defun mmultiply (x y)
  (let ((z (a-member-ofv '(even odd))))
    (assert! (ifv (andv (equalv (parityv x) 'odd)
                        (equalv (parityv y) 'odd))
                  (equalv z 'odd)
                  (equalv z 'even)))
    z)
  )

;;; This function meta-adds any number of arguments
(defun meta+ (x &rest args)
  (cond
   ((endp args) (parityv x))
   (t (madd (parityv x) (apply #'meta+ args)))
   )
  )

;;; This function meta-multiplies any number of arguments
(defun meta* (x &rest args)
  (cond
   ((endp args) (parityv x))
   (t (mmultiply (parityv x) (apply #'meta* args)))
   )
  )

;;; This function meta-compares two arguments
(defun meta= (x y)
  (equalv (parityv x) (parityv y))
  )
```

Figure 61: Functions Used when Mapping Arithmetic Expressions to the Meta-Level

Recognising Partial Fitness for Purpose

The approach to fitness for purpose described in this dissertation is a *binary* decision process (see "Plausibility as a Binary Decision Process" on page 197). That is, a configuration of problem solver and knowledge bases is deemed to be either (plausibly) fit for solving a goal, or unfit. This is somewhat unsatisfactory, because it provides no information about how much work would need to be done to transform an unfit configuration into a fit one. Rather than introduce a range of numbers that indicate the level of fitness, but do not state where the problems lie, I propose to extend the current model by introducing a *lattice* (i.e., a hierarchy with multiple inheritance) of problem solving roles. As we have seen, problem solving roles constrain the knowledge they contain to be fit for consumption by a problem solver. The extension would mean organising the criteria, represented by constraint objects, into a multiple inheritance hierarchy. The objects near the top of the hierarchy are weakly constrained, whereas those nearer the leaves are strongly constrained. A fitness-for-purpose search would traverse this hierarchy, looking for the strongest set of criteria which are fulfilled by a knowledge base. That is, if a knowledge base fails to satisfy a given role, the search could move to a weaker role (parent object), to see if it is satisfied by the knowledge base. If this role succeeds, the approach would be able to identify those criteria which are satisfied by the knowledge base, and those which are not. This provides significant information to a user who faces the prospect of fixing a knowledge base to solve the problem.

For example, consider a simplified view of the roles of people within a university department. The roles concern the abilities to *administer*, *teach*, and *research*. However, a secretary administers, but does not teach or research; a research student researches, but does not teach or administer; and a lecturer's main tasks are to teach and research. A head of department should be able to administer and lecture (see figure 62).

Given a knowledge source (in this case a person) who can teach and research, but cannot administer; and a head of department role to be filled, we would first try to assign the knowledge source to the head of department role (e.g., using `reconcile`; see page 293). This would fail because the knowledge source does not meet the criteria of the role. We would then look to the parents of the role to see if they might be satisfied.

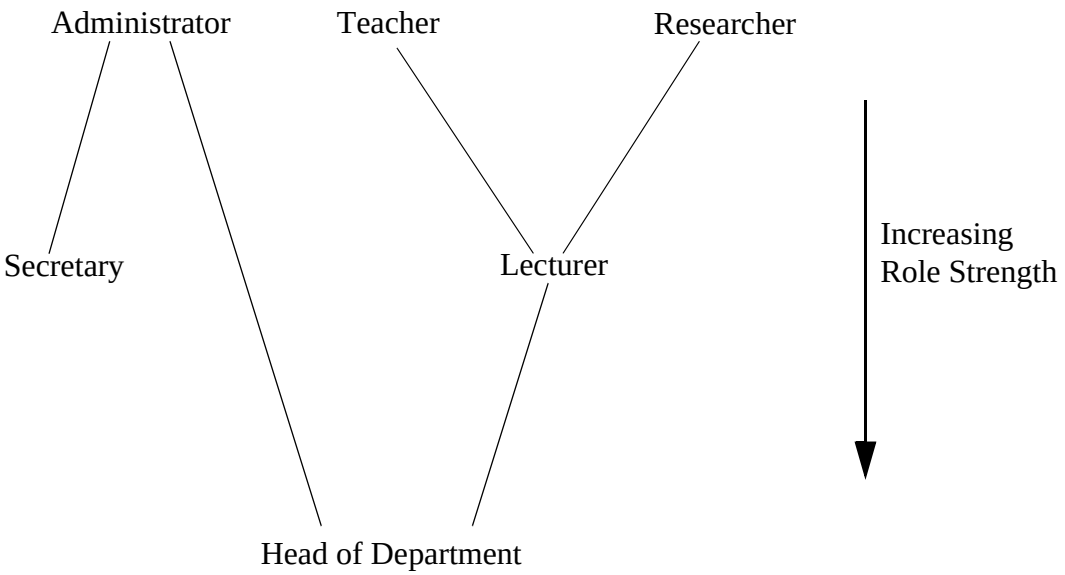


Figure 62: Simplified View of Roles within a University Department

In this case, the knowledge source meets the criteria for being a lecturer but not an administrator, thus providing information on the *reason* for failure.

Plausible Outputs of Rule Bases

I have not yet demonstrated how my approach can be applied to knowledge bases which contain sets of rules. Since such **rule bases** are subject to interpretation when "executed" by an inference engine, I believe they are more like *programs* than other knowledge bases containing purely structural knowledge. Each rule base should therefore be modelled by a **meta–rule-base** in the same way that a problem solver is modelled by a meta–problem-solver. Similarly, since sets of facts are the "inputs" to the rule base, they should also be made available as inputs to the meta–rule-base.

For example, let us reconsider the set of rules first introduced in chapter 2:

```
IF BarkColour is silvery_white
THEN TreeName is silver_birch;

IF BarkTexture is smooth AND BarkColour is grey
THEN TreeName is beech;

IF BarkTexture is fissured AND BarkColour is brown
THEN TreeName is oak;
```

These rules can be modelled by the following meta–rule-base:

```
(defun meta-rules (&key (bark-colour (make-variable))
                        (bark-texture (make-variable)))
  (let ((tree-name (make-variable)))
    (assert! (impliesv (equalv bark-colour 'silvery_white)
                       (equalv tree-name 'silver_birch)))
    (assert! (impliesv (andv (equalv bark-texture 'smooth)
                             (equalv bark-colour 'grey))
                       (equalv tree-name 'beech)))
    (assert! (impliesv (andv (equalv bark-texture 'fissured)
                             (equalv bark-colour 'brown))
                       (equalv tree-name 'oak)))
    tree-name) ; return the tree-name variable
  )
```

The input facts `bark-colour` and `bark-texture` are represented by constraint variables. A meta–rule-base can be used to answer queries about the outputs of the ground-level rule base:

```
;;; Nothing is known about the output when no inputs
;;; are supplied (“empty” constraint variable returned)
> (meta-rules)
[3]

;;; However, the output can become bound when enough
;;; information is supplied
> (meta-rules :bark-colour 'grey :bark-texture 'smooth)
BEECH

;;; It is also possible to test the output against some
;;; “goal” property. For example, to test whether a tree
;;; with silvery_white bark could be an oak
> (equalv (meta-rules :bark-colour 'silvery_white) 'oak)
NIL
```

Note that caution must be exercised when the rule set uses non-monotonic logic; that is, it is allowed to “overwrite” values which have already been inferred. This behaviour cannot easily be modelled by a constraint program, and certainly not one without the power of constraint retraction. Instead, the meta–rule-base should model only those inferred values which are not “overwritten”. This approach is consistent with the plausibility approximation introduced in chapter 6, because the output of the meta–rule-base still subsumes the output of the ground-level rule base.

Plausible Outputs of KA Tools

One of the original intentions of this work was to describe the capabilities of KA tools (see page 77). As you have seen, the focus changed to the description of the capabilities

of problem solvers. However, the issue of describing KA tools remains, and we are now in a much better position to attempt this task. As a start, I suggest using the SCREAMER+ notation to describe the plausible outputs of repertory grids and decision trees.

The central idea is that if you are using a repertory grid, and you know the elements whose constructs are to be acquired, then you already know something about the output that will be generated. If you also know the constructs, but not the values, then even more can be said about the output. The statements of what are known can be made in terms of constraints, and the resulting plausible data structure can be used for reasoning in the same way as the plausible outputs of problem solvers. For example, if one were to use the repertory grid to add to the knowledge shown in Figure 5 on page 20, then the following data structure might be generated *prior* to knowledge acquisition:

```
((:ELEMENT VECTRA
  ((:CONSTRUCT SIZE :LEFT SMALL :RIGHT LARGE :VAL 4)
    (:CONSTRUCT COSTS :LEFT ECONONICAL :RIGHT EXPENSIVE :VAL 3)
   (:CONSTRUCT PRESTIGE :LEFT LOW :RIGHT HIGH :VAL 4)
    ...
    )
  )
 (:ELEMENT ESCORT
  ((:CONSTRUCT SIZE :LEFT SMALL :RIGHT LARGE :VAL 3)
    (:CONSTRUCT COSTS :LEFT ECONONICAL :RIGHT EXPENSIVE :VAL 2)
   (:CONSTRUCT PRESTIGE :LEFT LOW :RIGHT HIGH :VAL 2)
    ...
    )
  )
 (:ELEMENT MINI
  ((:CONSTRUCT SIZE :LEFT SMALL :RIGHT LARGE :VAL 1)
    (:CONSTRUCT COSTS :LEFT ECONONICAL :RIGHT EXPENSIVE :VAL 1)
   (:CONSTRUCT PRESTIGE :LEFT LOW :RIGHT HIGH :VAL 1)
    ...
    )
  )
  ...
 )
```

Note that the ellipses "`...`" have been used to represent lists whose tails are unknown. This requirement suggests an application for the lazy list work suggested above.

To represent plausible decision trees, one can look at the input data to the machine learning algorithm, and make statements about the relationship between the input and the output (see Table 2 on page 24; and Figure 6, also on page 25). The output is always a decision tree, which is a special kind of graph. Each node in the graph contains

a label which must be one of the attributes of the data set. The connections between the nodes are unknown prior to learning, but the nature of those connections are known. They are directed links that are labelled with the possible values of the nodes attribute. They may lead either to another node, or to the classification of the example (in the example, either +ve or -ve). I believe that such qualitative knowledge can be captured by constraints and represented by a data structure – a "plausible decision tree".

8.3.3 Acquiring Knowledge which is Fit for a Purpose

A Graphical User-Interface to COCKATOO

A graphical user-interface could be built on as a "front-end" to the existing COCKATOO system, and enable a faster and slicker interaction between the user and the program. For example, rather than typing the letter which labels a desired choice and then pressing `ENTER`, the user could simply point at the desired option and click the mouse. Although this would not make any fundamental contribution to this work, it might help to fulfil the expectations of modern computer users.

Scheduling the Acquisition of Knowledge Elements

The order in which questions are asked by a knowledge acquisition tool are very important. An *inefficient* ordering would lead to questions being answered unnecessarily by the user (e.g., asking whether someone is pregnant before eliciting their sex). Alternatively, a bad ordering could simply appear illogical, or even violate the rules of etiquette (consider a medical application which attempts to elicit intimate, but apparently irrelevant, details from a patient). Consider again the `fruit-juice` clause of the breakfast grammar on page 218. Since fruit-juice is a component of each of the given breakfasts, it could be 'factored out', so that it featured in the `brekky` clause, but in none of the clauses `fried-breakfast`, `continental-breakfast`, or `cereal-breakfast`. This has not been done in the example because it leads to a grammar which asks the user for their choice of fruit juice before they have chosen which breakfast they would like.

COCKATOO's elicitation ordering is currently implicit in the structure of the grammar (recall that the acquisition engine performs a depth-first expansion of the grammar).

Small changes to the grammar can substantially alter this order, and therefore also greatly affect the reactions of users to the tool. One idea to tackle this problem is to allow the assertion of "meta-constraints", which help to schedule parts of the acquisition session. In the above example, the sex of patients should be acquired before asking whether they are pregnant. Rather than just organising the grammar so that this occurs, an explicit scheduling constraint could also be added. That way, the behaviour of the grammar is more resistant to subsequent change. The order of acquisition might also depend on other information that is known about the variables, such as their type, boundness, domain size, domain values, etc.

Meta-Grammars for Acquiring Grammars

I have already argued the flexibility of COCKATOO for acquiring textual data, and stated some of the advantages offered by a clear separation of COCKATOO's run-time performance into a knowledge specification (the constraint-augmented grammar) and an acquisition engine (the COCKATOO shell itself). Given that a knowledge specification is provided as a textual file, another possible application of COCKATOO might be to acquire these files. The COCKATOO acquisition engine would use a kind of 'meta-grammar' to acquire domain-specific grammars. This is only likely to work if the domain-specific grammar required is relatively straightforward, but even so, using COCKATOO reflectively in this way could prove to be a labour-saving idea.

8.3.4 Extending the MUSKRAT Vision

Elicitation of the Problem-Solving Goal

Although MUSKRAT has been described as a *goal-driven* architecture for problem solving, no mechanism has been proposed thus far in this dissertation for the elicitation and subsequent representation of the user's problem-solving goal. In chapter 6, the goal was represented as a constraint expression, which was combined with another constraint expression representing the output of a problem solver. However, it would be unreasonable to expect a MUSKRAT user to input a SCREAMER+ constraint expression directly.

To overcome this problem, I suggest that COCKATOO could be configured to first elicit the user's goal, then apply its postprocessing capabilities to output a corresponding constraint expression. The constraint expression could then be combined with the (plausible) outputs of problem solvers to assess the overall plausibility of the task.

A Distributed MUSKRAT

MUSKRAT was envisaged as a multistrategy toolbox installed on a single machine. Since the system was specified before the popularity explosion of the Internet, its originators had not conceived of a multistrategy system whose functionality is *distributed over many machines*. Such a change in requirements may not contribute significantly to the science of knowledge acquisition, but it could have a considerable impact on the number of users of the MUSKRAT system, and therefore also on the technology transfer of knowledge acquisition to other domains. The advantages of a distributed system are that a *multi-platform* client can be offered, such as one written in Java; and MUSKRAT's tools need not be installed on the local machine. Instead, they can be offered as a remote *service* across the network.

A distributed MUSKRAT system could be built by connecting a lightweight Java client to a heavyweight LISP server. In its most basic form, the Java client would be simply a remote LISP listener which sends LISP commands across the network to the server, receives responses for each, and prints them to the screen. In a more secure version, the LISP protocol would be hidden from the user, but steered by the client's graphical user-interface. The user-interface would enable a user to specify a goal, and apply the problem solving methods held by the MUSKRAT server. However, the knowledge bases to be tested for fulfilment of problem-solving roles could be distributed over a set of known, contributing sites *on the Internet*. A distributed framework would enable researchers to maintain their own knowledge bases, but also make them available to others through the extended MUSKRAT. I believe this approach could act as a catalyst to knowledge reuse.

8.4 Conclusions

In this dissertation, I have claimed that constraint technology can be usefully applied in the field of knowledge acquisition. To support the claim, I presented a constraint-based notation that can be used for *specifying the fitness-for-purpose requirements* for using a problem solver to satisfy a particular problem-solving goal. The constraint notation is also supported by mechanisms which *examine whether requirements are satisfied*. A constraint-based knowledge acquisition tool was also developed which only acquires knowledge that is consistent with a set of fitness-for-purpose requirements.

To demonstrate my claims, I made significant further developments to the SCREAMER constraint-handling package, calling the extended version SCREAMER+. SCREAMER+ embraced the symbolic, functional and object-oriented spirit of LISP, and was shown to be able to solve a selection of benchmark CSP problems.

In this chapter, I have suggested several possible extensions to my work. I believe that these extensions will prove to be both significant and interesting challenges for future research.

References

ANGELE, J., DECKER, S., PERKUHN R., STUDER R., (1996), “Modeling Problem-Solving Methods in New KARL”, in Proceedings of the Tenth Knowledge Acquisition Workshop for Knowledge-Based Systems (KAW–96), pp. 1-18, Banff, Canada.

ARCOS, J. L., PLAZA, E., (1994), “Integration of Learning into a Knowledge Modelling Framework”, in Proceedings of the Eighth European Knowledge Acquisition Workshop (EKAW–94), LNCS, Springer Verlag.

ARCOS, J. L., PLAZA, E., (1997), “NOOS: An Integrated Framework for Problem Solving and Learning”, Research Report 97-02, Institut d'Investigació en Intelligència Artificial (IIIA), Barcelona, Spain.

ARMENGOL, E., BENJAMINS, V. R., DECKER, S., FENSEL., D., MOTTA, E., PLAZA, E., STUDER, R., WIELINGA, B., (1998), “State of the Art Deliverable”, IBROW3 Project Deliverable (ESPRIT Project 27169).

BACKHOUSE, R. C., (1986), “Program Construction and Verification”, Prentice Hall International, Hemel Hempstead, UK.

BARR, A., FEIGENBAUM, E. A., (1982), “The Handbook of Artificial Intelligence”, Volume 2, William Kaufmann, Los Altos, California.

BEASLEY, D., BULL, D. R., MARTIN, R. R., (1993), “An Overview of Genetic Algorithms: Part 1, Fundamentals”, University Computing, Vol. 15, No. 2, pp. 58-69.

BENJAMINS, V. R., PIERRET-GOLBREICH, C., (1996), “Assumptions of Problem-Solving Methods”, in Proceedings of the Ninth European Knowledge Acquisition Workshop (EKAW–96), LNCS, Springer Verlag, pp. 1-16.

BENJAMINS, V. R., PLAZA, E., MOTTA, E., FENSEL, D., STUDER, R., WIELINGA, B., SCHREIBER, G., ZDRAHAL, Z., DECKER, S., (1998), “IBROW3 – An Intelligent Brokering Service for Knowledge Component Reuse on the World-Wide Web”, in proceedings of the Eleventh Banff Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW–98), Banff, Alberta, Canada.

BENJAMINS, V. R., WIELINGA, B., WIELEMAKER, J., FENSEL, D., (1999), “Towards Brokering Problem Solving Knowledge on the Internet”, in Fensel, D., Studer, R., (Eds.), in Proceedings of the Eleventh European Workshop on Knowledge Acquisition, Modeling, and Management (EKAW–99), LNCS, Springer Verlag, pp. 33-48.

BIÉBOW, B., SZULMAN, S., (1999), “TERMINAE: A Linguistics-Based Tool for the Building of a Domain Ontology”, in Fensel, D., Studer, R., (Eds.), Proceedings of

the Eleventh European Workshop on Knowledge Acquisition, Modeling, and Management (EKAW–99), LNCS, Springer Verlag, pp. 49-66.

BISSON, G., (1992), "Learning in FOL with a Similarity Measure", in Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI–92), pp. 82-87.

BOOKER, L. B., GOLDBERG, D. E., HOLLAND, J. H., (1990), "Classifier Systems and Genetic Algorithms", in (Carbonell, 1990a).

BOOSE, J. H., (1990), "Uses of Repertory Grid-Centered Knowledge Acquisition Tools for Knowledge-Based Systems", in Boose, J., Gaines, B., (Eds.), *Foundations of Knowledge Acquisition,* Knowledge-Based Systems Book Series, Volume 4, pp. 61-83, London: Academic Press.

BORNAT, R., (1986), "Understanding and Writing Compilers", Macmillan Press, London.

BOSWELL, R., CRAW, S., ROWE, R., (1997), "Knowledge Refinement for a Design System", in Proceedings of the Tenth European Knowledge Acquisition Workshop, (EKAW–97), Sant Feliu de Guixols, Spain, Springer Verlag, pp. 49-64.

BRACHMAN, R. J., (1979), "On the Epistemological Status of Semantic Networks", in Findler, N. V. (Ed.), *Associative Networks: Representation and Use of Knowledge by Computers*, New York: Academic Press, pp. 3-50.

BROWN, K. N., MCMAHON, C. A., SIMS WILLIAMS, J. H., (1994), "A Formal Language for the Design of Manufacturable Objects", in Gero, J. S., Tyugu, E., (1994), *Formal Design Methods for CAD*, Elsevier, pp. 135-156.

BUCHANAN, B. G., BARSTOW, D., BECHTEL, R., BENNET, J., CLANCEY, W., KULIKOWSKI, C., MITCHELL, T. M., and WATERMAN, D. A., (1983), "Constructing an Expert System", in Hayes-Roth, F., Waterman, D. A., Lenat D., (1983), *Building Expert Systems*, Teknowledge Series in Knowledge Engineering, Addison Wesley, Reading, Massachusetts, USA.

CARBONARA, L., SLEEMAN, D., (1996a), "Improving the Efficiency of Knowledge Base Refinement", in Proceedings of the Thirteenth International Machine Learning Conference (ICML–96), San Francisco: Morgan Kaufmann, pp 78-86.

CARBONARA, L., SLEEMAN, D., (1996b). "STALKER: an Efficient Knowledge Base Refinement System", in Proceedings of the Verification, Validation and Refinement of Knowledge Based Systems Workshop at the Pacific Rim International Conference on Artificial Intelligence, Antoniou G., (Ed). Technical Report CIT-96-05, Griffith University, ISSN: 0818-3457, pp. 7-16.

CARBONELL, J., (Ed.) (1990a), "Machine Learning – Paradigms and Methods", MIT Press, Cambridge, Massachusetts, USA.

CARBONELL, J., (1990b), "Paradigms for Machine Learning", in (Carbonell, 1990a).

CARBONELL, J. G., KNOBLOCK, C. A., MINTON, S., (1991), "Prodigy: An integrated architecture for planning and learning", in (VanLehn, 1988), pp. 241-278.

CARLSON, C., (1993), "Grammatical Programming: An Algebraic Approach to the Description of Design Spaces", Department of Architecture, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

CASEAU, Y., LABURTHE, F., (1995), “Introduction to the CLAIRE Programming Language”, LIENS Report 96-15, Department of Mathematics and Informatics, Ecole Normale Supérieure, Paris.

CASEAU, Y., LABURTHE, F., (1996), “CLAIRE: a brief overview”, LIENS Working Paper, Department of Mathematics and Informatics, Ecole Normale Supérieure, Paris; available from `http://www.ens.fr/~laburthe/papers.html`

CHANDRASEKARAN, B., (1983), “Towards a Taxonomy of Problem Solving Types”, AI Magazine, Vol. 4, No. 1, pp. 9-17.

CLANCEY, W. J., (1985), “Heuristic Classification”, Artificial Intelligence, Vol. 23, No. 3, pp. 289-350.

COLLINS, A., MICHALSKI, R. S., (1989), “The Logic of Plausible Reasoning: A Core Theory”, Cognitive Science, Vol. 13, pp. 1-49.

CRAW, S., (1991), “Automating the Refinement of Knowledge-Based Systems”, PhD Thesis, Computing Science Department, University of Aberdeen, Scotland, UK.

CRAW, S., SLEEMAN, D., GRANER, N., RISSAKIS, M., SHARMA, S., (1992), “CONSULTANT: Providing advice for the Machine Learning Toolbox”, in Proceedings of the 1992 BCS Expert Systems Conference, Bramer, M. (Ed.), Cambridge University Press.

CRAW, S. M., SLEEMAN, D., GRANER, N., RISSAKIS, M., SHARMA, S., (1992), “CONSULTANT: Providing Advice for the Machine Learning Toolbox”, AUCS/TR9210, Aberdeen University Computing Science Technical Report, 1992.

DECHTER, R., FROST, D., (1998), “Backtracking algorithms for constraint satisfaction problems – a tutorial survey”, Information- and Computer Science Technical Report R56, University of California at Irvine, USA.

DEJONG, G., MOONEY, R., (1986), “Explanation-Based Learning: An Alternative View”, Machine Learning, Vol. 1, pp. 145-176.

DIAPER, D., (1989), “Knowledge Elicitation: Principles, Techniques and Applications”, Ellis Horwood, Chichester, England, UK.

DIETTERICH, T. G., HILD, H., BAKIRI, G., (1990), “A Comparative Study of ID3 and Backpropagation for English Text-to-Speech Mapping”, in Porter B. W., Mooney R., (Eds.), *Machine Learning*, Morgan Kaufmann, Los Altos/Palo Alto/San Francisco, pp. 24-31.

DINCBAS, M., SIMONIS, H., VAN HENTENRYCK, P., (1988), “Solving the Car-Sequencing Problem in Constraint Logic Programming”, in proceedings of ECAI–88.

DOMINGUE, J., (1998), “Tadzebao and WebOnto: Discussing, Browsing and Editing Ontologies on the Web”, in Gaines, B., Musen, M., (Eds.), Proceedings of the Eleventh Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff, Canada.

DUKE UNIVERSITY, (1995), “HL7 Frequently Asked Questions”, `http://www.mcis.duke.edu/standards/HL7/faq/HL7FAQ10.HTML`, version 1.1, 10/12/95.

ERICSSON, K. A., SIMON, H. A., (1984), "Protocol Analysis: Verbal reports as data", Cambridge, MA, USA. MIT Press.

ERIKSSON, H., PUERTA, A. R., GENNARI, J. H., ROTHENFLUH, T. E., SAMSON, W. T., MUSEN, M., (1994), "Custom-Tailored Development Tools for Knowledge-Based Systems", Technical Report KSL-94-67, Section on Medical Informatics, Knowledge Systems Laboratory, Stanford University, California, USA.

ESHELMAN, L., (1988), "MOLE: A Knowledge-Acquisition Tool for Cover-and-Differentiate Systems", in Marcus, S., (Ed.), "Automating Knowledge Acquisition for Expert Systems", Kluwer Academic Publishers, pp. 37-80.

FEIGENBAUM, E. A., (1977), "The Art of Artificial Intelligence: Themes and Case Studies in Knowledge Engineering", in Proceedings of IJCAI–77, pp. 1014-29.

FENSEL, D., ANGELE, J., STUDER, R., (1995), "The Knowledge Acquisition and Representation Language KARL", Research Report 316, AIFB, University of Karlsruhe, Germany.

FENSEL, D., (1995b), "Assumptions and Limitations of a Problem Solving Method: A Case Study", Research Report 312, AIFB, University of Karlsruhe, Germany.

FENSEL, D., (1997), "The Tower-of-Adapter Method for Developing and Reusing Problem-Solving Methods", in Proceedings of the Tenth European Workshop on Knowledge Acquisition, Modeling and Management, Lecture Notes in Artificial Intelligence Series (LNAI–1319), Springer-Verlag, Berlin, pp. 97-112.

FENSEL, D., GROENBLOOM, R., (1997), "Specifying Knowledge-Based Systems with Reusable Components", in Proceedings of the Ninth International Conference on Software Engineering & Knowledge Engineering (SEKE–97), Madrid, Spain.

FENSEL, D., MOTTA, E., BENJAMINS, V. R., DECKER, S., GASPARI, M., GROENBOOM, R., GROSSO, W., VAN HARMELEN, F., MUSEN, M., PLAZA, E., SCHREIBER, G., STUDER, R., TEN TEIJE, A., WIELINGA, B., (1999), "The Unified Problem-Solving Method Development (sic.) Language UPML", Deliverable 1.1 (Version 2), ESPRIT Project 27169.

FENSEL, D., SCHÖNEGGE, A., (1997), "Using KIV to Specify and Verify Architectures of Knowledge-Based Systems", in Proceedings of the Twelfth International Conference on Automated Software Engineering (ASEC–97), Incline Village, Nevada.

FENSEL, D., SCHÖNEGGE, A., (1998), "Inverse Verification of Problem Solving Methods", International Journal of Human-Computer Studies, Vol. 49, No. 4, pp. 339-361.

FENSEL, D., STUDER, R., (1999), Preface to the Proceedings of the Eleventh European Workshop on Knowledge Acquisition, Modelling and Management (EKAW–99), Springer Verlag.

FERNÁNDEZ, A., HILL, P., (1997), "Finite Domain Solvers Compared Using Self Referential Quizzes", Technical Report 97.03, School of Computer Studies, University of Leeds, UK.

FIRLEJ, M., HELLENS, D., (1991), "Knowledge Elicitation: A Practical Handbook", Prentice Hall, Hemel Hempstead, UK.

FISCHER, B., SNELTING, G., (1997), "Reuse by Contract", Proceedings of the ESEC/FSE Workshop on Foundations of Component-Based Systems, Zurich, pp. 91-100.

FRANK, M. R., SZEKELY, P., (1998), "Adaptive Forms: An Interaction Technique for Entering Structured Data", Knowledge-Based Systems, Vol. 11, pp. 37-45.

FREUDER, E. C., (1997), "In Pursuit of the Holy Grail", Constraints Journal, Kluwer Acadmic Publishers Vol. 2, No. 1, pp. 57-61.

GAINES, B., SHAW, M. L. G., (1997), "Knowledge acquisition, modelling and inference through the World-Wide Web", International Journal of Human–Computer Studies, Vol. 46, pp. 729-759.

GANASCIA, J-G., THOMAS, J., LAUBLET, P., (1993), "Integrating models of knowledge and machine learning", in *Machine Learning: ECML–93*, Brazdil, P. B. (Ed.), Springer-Verlag.

GARDNER, A., (1981), "Search", in Barr, A., Feigenbaum, E.A., (Eds.), *Handbook of Artificial Intelligence*, Vol. 1, Los Altos, California, William Kaufmann, pp. 19-139.

GENESERETH, M. R., FIKES R. E., (1992), "Knowledge Interchange Format Reference Manual, Version 3.0", Computer Science Department, Stanford University.

GENNARI, J. H., CHENG, H., ALTMAN, R. B., MUSEN, M. A., (1998), "Reuse, CORBA, and Knowledge-based Systems", International Journal of Human-Computer Studies, Vol. 49, No. 4, pp. 523-546.

GENT, I. P., WALSH, T., (1999), "CSPLIB: A Benchmark Library for Constraints", Technical Report APES-09-1999, APES (Algorithms, Problems and Empirical Studies) Group, Department of Computer Science, University of Strathclyde, Glasgow, UK. Available at `http://csplib.cs.strath.ac.uk`.

GIRARDI, M. R., IBRAHIM, B., (1995), "Using English to Retrieve Software", The Journal of Systems and Software, Vol. 30, No. 3, pp. 249-270.

GIUNCHIGLIA, F., WALSH, T., (1992), "A Theory of Abstraction", Artificial Intelligence, Vol. 56, No. 2-3, pp. 323-390.

GOLDSCHLAGER, L., LISTER, A., (1982), "Computer Science: A Modern Introduction", Prentice Hall International, London.

GÓMEZ-PÉREZ, A., ROJAS-AMAYA, M. D., (1999), "Ontological Reengineering for Reuse", in Fensel, D., Studer, R., (Eds.), Proceedings of the Eleventh European Workshop on Knowledge Acquisition, Modeling, and Management (EKAW–99), LNCS, Springer Verlag, pp. 139-156.

GRANER, N., SHARMA, S., SLEEMAN, D., RISSAKIS, M., CRAW, S. M., (1992), "The Machine Learning Toolbox Consultant", Technical Report AUCS/TR9207, Department of Computing Science, University of Aberdeen, Scotland, UK.

GRANER, N., SLEEMAN, D., (1993), "MUSKRAT: a Multistrategy Knowledge Refinement and Acquisition Toolbox", in *Proceedings of the Second International Workshop on Multistrategy Learning*, Michalski, R. S., Tecuci, G., (Eds.), pp. 107-119.

GRAY, P. M. D., PREECE, A., FIDDIAN, N. J., GRAY, W. A., BENCH-CAPON, T. J. M., SHAVE, M. J. R., AZARMI, N., WIEGAND, M., ASHWELL, M., BEER, M., CUI, Z., DIAZ, B., EMBURY, S.M., HUI, K., JONES, A. C., JONES, D. M., KEMP, G. J. L., LAWSON, E. W., LUNN, K., MARTI, P., SHAO, J., VISSER, P. R. S., (1997), "KRAFT: Knowledge Fusion from Distributed Databases and Knowledge Bases", Conference on Database and Expert System Applications (DEXA–97), Toulouse, France.

GROOT, P., TEN TEIJE, A., VAN HARMELEN, F., (1999), "Formally Verifying Dynamic Properties of Knowledge Based Systems", in Proceedings of the Eleventh Workshop on Knowledge Acquisition, Modelling and Management (EKAW–99), Springer Verlag.

HAMMOND, K. J., (1993), "Explaining and Repairing Plans That Fail", in Readings in Knowledge Acquisition and Learning, Buchanan, B. G., Wilkins, D. C., (Eds.), Morgan Kaufmann, San Mateo, California, USA.

HARVEY, W., (1995), "Nonsystematic Backtracking Search", PhD Thesis, Department of Computer Science, Stanford University, California.

HENZ, M., MÜLLER, T., (2000), "An Overview of Finite Domain Constraint Programming", to appear in proceedings of the Fifth Conference of the Association of Asia-Pacific Operational Research Societies, APORS–2000.

HINRICHS, T., (1992), "Problem Solving in Open Worlds: A Case Study in Design", Erlbaum, Northvale, New Jersey, USA.

HINTON, G., (1990), "Connectionist Learning Procedures", in (Carbonell, 1990a).

HUTCHINSON, A., (1994), "Algorithmic Learning", Oxford University Press, Oxford, UK.

JACKSON, P., (1990), "Introduction to Expert Systems", Second Edition, Addison-Wesley Publishing, Wokingham, UK.

JOHNSON, J., (1997), "Mathematics, Representation, and Problem Solving", Mathematics Today, Bulletin of the Institute of Mathemeatics and its Applications, Vol. 33, No. 3, pp. 78-80.

JONES, C. B., (1986), "Systematic Software Development Using VDM", Prentice Hall International, Hemel Hempstead, UK.

KELLY, G. A., (1955), "The Psychology of Personal Constructs", W. W. Norton & Company Inc., New York, USA.

KODRATOFF, Y., SLEEMAN, D., USZYNSKI, M., CAUSSE, K., CRAW, S., (1992), "Building a Machine Learning Toolbox", in *Enhancing the Knowledge Engineering Process*, Steels, L., Lepape, B., (Eds.), North-Holland, Elsevier Science Publishers, pp. 81-108.

KNOBLOCK, C., (1991), "Automatically Generating Abstractions for Problem Solving", PhD Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, USA.

KUMAR, V., (1992), "Algorithms for Constraint Satisfaction Problems: A Survey", AI Magazine, Vol. 13, No. 1, pp. 32-44.

LABURTHE, F., SAVÉANT, P., DE GIVRY, S., JOURDAN, J., (1998), "ECLAIR, a library of constraints over finite domains", Technical Report ATS 98-2, Thomson CSF, Corporate Research Laboratory, Paris.

LAIRD, J., HUCKA, M., HUFFMAN, S., ROSENBLOOM, P., (1991), "An analysis of Soar as an integrated architecture", in SIGART Bulletin Special Section on Integrated Cognitive Architectures, Vol. 2, No. 4, pp. 98-103.

LARSON, L. C., (1983), "Problem-Solving Through Problems", Springer Verlag, New York.

MACKENZIE, D., (1995), "The Automation of Proof: A Historical and Sociological Exploration", IEEE Annals of the History of Computing, Vol. 17, No. 3, pp. 7-29.

MAJUMDER, J., (2000), Personal Communication.

MARCUS, S., (1988), "SALT: A Knowledge-Acquisition Tool for Propose-and-Revise Systems", in Marcus, S., (Ed.), "Automating Knowledge Acquisition for Expert Systems", Kluwer Academic Publishers, pp. 81-124.

MARLING, C. R., STERLING, L. S., (1996), "Designing Nutritional Menus Using Case-Based and Rule-Based Reasoning", in *Artificial Intelligence in Design '96*, Kluwer Academic Publishers.

MCDERMOTT, J., (1982), "R1: A Rule-Based Configurer of Computer Systems", Artificial Intelligence, Vol. 19, No. 1, pp. 39-88.

MCDERMOTT, J., (1988), "Preliminary Steps Toward a Taxonomy of Problem-Solving Methods", in Marcus, S., (Ed.), "Automating Knowledge Acquisition for Expert Systems", Kluwer Academic Publishers, pp. 225-256.

MERELO, J. J., (1996), "Genetic Mastermind, a case of dynamic constraint optimization", Technical Report G-96-1, Department of Electronics and Computer Technology, University of Granada, Spain.

MICHAIL, A., NOTKIN, D., (1999), "Assessing Software Libraries by Browsing Similar Classes, Functions and Relationships", in Proceedings of the 21st International Conference on Software Engineering (ICSE–-99), Los Angeles, CA, USA. Also available as Technical Report UW-CSE-TR-98-08-05, (1998), Department of Computer Science & Engineering, University of Washington.

MICHALSKI, R. S., TECUCI, G. (Eds.), (1991), Proceedings of the First International Workshop on Multistrategy Learning (MSL–91), George Mason University, Fairfax, VA.

MINTON, S., CARBONELL, J. G., KNOBLOCK, C. A., KUOKKA, D. R., ETZIONI, O., GIL, Y., (1990), "Explanation Based Learning: A Problem Solving Perspective", in (Carbonell, 1990a).

MINTON, S., JOHNSTON, M. D., PHILIPS, A. B., LAIRD, P., (1992), "Minimizing Conflicts: a heuristic repair method for constraint satisfaction and scheduling problems", Artificial Intelligence, Vol. 58, pp. 161-205.

MITCHELL, F., (1998), "An Introduction to Knowledge Acquisition", Technical Report AUCS/TR9804, Department of Computing Science, University of Aberdeen, UK.

MITCHELL, T. M., KELLER, R. M., KEDAR-CABELLI, S. I., (1986), "Explanation-Based Generalisation: A Unifying View", Machine Learning, Vol. 1, pp. 47-80.

MITCHELL, T. M., ALLEN, J., CHALASANI, P., CHENG, J., ETZIONI, O., RINGUETTE, M., AND SCHLIMMER, J.C., (1991), "Theo: A framework for self-improving systems" in (VanLehn, 1988), pp. 323-355.

MLT CONSORTIUM, (1992a), "Specification of the Common Knowledge Representation Language of the MLToolbox", Technical Report MLT/LRI/wp20392, ESPRIT Project 2154.

MLT CONSORTIUM, (1992b), "Final Specifications of the Common Knowledge Representation Language of the MLToolbox", Technical Report MLT/LRI/wp20392, ESPRIT Project 2154, March 1992.

MLT CONSORTIUM, (1993), "Final Report on CKRL", Technical Report LRI/MLT/wp2/0393, ESPRIT Project 2154.

MORIK, K., CAUSSE, K., BOSWELL, R., (1991), "A common knowledge representation integrating learning tools", in (Michalski and Tecuci, 1991), pp. 81-91

MORIK, K., WROBEL, S., KIETZ J-U., EMDE, W., (1993), "Knowledge Acquisition and Machine Learning: Theory, Methods and Applications", Academic Press, London.

MOTTA, E., O'HARA, K., SHADBOLT, N., (1994), "Grounding GDMs: A Structured Case Study", International Journal of Human-Computer Studies, Vol. 40, No. 2, pp. 315-347.

MOTTA, E., O'HARA, K., SHADBOLT, N., (1996), "Solving VT in VITAL: A Study in Model Construction and Knowledge Reuse", International Journal of Human-Computer Studies, Vol. 44, No. 3, pp. 333-371.

MÜLLER, M., SMOLKA, G., (1996), "Oz: Nebenläufige Programmierung mit Constraints", Künstliche Intelligenz, Themenheft Logische Programmierung, Scientec Publishing, Bad Ems., pp. 55-61.

MUSEN, M., (1998), Personal Communication, (Feedback on PhD Thesis Proposal).

MYERS, B. A., MCDANIEL, R. G., MILLER, R. C., FERRENCY, A. S., FAULRING, A., KYLE, B. D., MICKISH, A., KLIMOVITSKI, A., and DOANE, P., (1997), "The Amulet Environment: New Models for Effective User Interface Software Development", IEEE Transactions on Software Engineering, Vol. 23, No. 6, pp. 347-365.

NÉDELLEC, C., CAUSSE, K., (1992), "Knowledge refinement using Knowledge Acquisition and Machine Learning Methods", in Proceedings of EKAW–92, Springer Verlag.

NEWELL, A., (1982), "The Knowledge Level", Artificial Intelligence, Vol. 18, No. 1, pp. 87-127.

O'HARA, K., SHADBOLT, N., (1996), "The Thin End of the Wedge: Efficiency and the Generalised Directive Model Methodology", in Shadbolt, N., O'Hara, K., Schreiber, G., (Eds), *Advances in Knowledge Acquisition*, proceedings of the Ninth European Knowledge Acquisition Workshop (EKAW–96), Nottingham, UK, pp. 33-47.

O'HARA, K., SHADBOLT, N., VAN HEIJST, (1998), "Generalised Directive Models: Integrating Model Development and Knowledge Acquisition", International Journal of Human-Computer Studies, Vol. 49, No. 4, pp. 497-522.

OROUMCHIAN, F., (1995), "Theory of Plausible Reasoning", in *Information Retrieval by Plausible Inferences: An Application of the Theory of Plausible Reasoning of Collins and Michalski*, PhD Thesis, School of Computer and Information Science, Syracuse University, New York.

PIERRET-GOLBREICH, C., (1998), "Supporting Organization and Use of Problem-solving Methods Libraries by a Formal Approach", International Journal of Human-Computer Studies, Vol. 49, No. 4, pp. 471-495.

POLYA, G., (1957), "How To Solve It: A New Aspect of Mathematical Method", Doubleday Anchor Books, New York.

PREECE, A., (1998), "Building the Right System Right: Evaluating V&V Methods in Knowledge Engineering", Eleventh Banff Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW–98), Banff, Alberta, Canada.

PROSSER, P., (1993), "Hybrid Algorithms for Constraint Satisfaction Problems", Computational Intelligence, Vol. 9, No. 3, pp. 268-299.

PUERTA, A., EGAR, J., TU, S., MUSEN, M., (1992), "A Multiple-Method Knowledge-Acquisition Shell for the Automatic Generation of Knowledge Acquisition Tools", Knowledge Acquisition, Vol. 4, pp. 171-196.

PUGET, J-F., (1994), "A C++ Implementation of CLP", in Proceedings of SPICIS 94, Singapore. Also available from `http://www.ilog.fr/html/products/optimization/research_papers.htm`

PUPPE, F., (1998), "Knowledge Reuse among Diagnostic Problem-Solving Methods in the Shell-Kit D3", International Journal of Human–Computer Studies, Academic Press, Vol. 49, No. 4, pp. 627-649.

QUINLAN, J. R., (1983), "Learning Efficient Classification Procedures and their Application to Chess Endgames", in Michalski, R. S., Carbonell, J. G., Mitchell, T. M., *Machine Learning: An Artificial Intelligence Approach*, Tioga Publishing Company, Palo Alto, California, USA.

QUINLAN, J. R., (1986), "Induction of Decision Trees", Machine Learning, Vol. 1, pp. 81-106.

REICHGELT, H., SHADBOLT, N., (1992), "ProtoKEW: A knowledge-based system for knowledge acquisition", in *Artificial Intelligence*, Sleeman, D, and Bernsen, NO (Eds.), Research Directions in Cognitive Science: European Perspectives, volume 6, Lawrence Erlbaum, Hove, UK.

ROBERTSON, D., (1999), Personal Communication.

RODOSEK, R., (1997), "Generation and Comparison of Constraint-Based Heuristics Using the Structure of Constraints", PhD Thesis, Department of Computing, Imperial College of Science, Technology and Medicine, University of London.

RUMELHART, D. E., HINTON, G. E., WILLIAMS, R. J., (1986), "Learning Internal Representations by Error Propagation, in Parallel Distributed Processing, Vol. 1, Rumelhart, D. E., McClelland J. L., (Eds.), MIT Press, 1986.

RUSSELL, S., NORVIG, P., (Eds.), (1995), "Artificial Intelligence – A Modern Approach", Prentice-Hall International, New Jersey, USA.

SABIN, D., FREUDER, E., (1994), "Contradicting Conventional Wisdom in Constraint Satisfaction", in Cohn, A.G. (Ed.), Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI–94), Wiley, London, UK, pp. 125-129.

SCHAERF, M., CADOLI, M., (1995), "Tractable Reasoning via Approximation", Artificial Intelligence, Vol. 74, No. 2, pp. 249-310.

SHIEBER, S. M., (1986), "An Introduction to Unification-Based Approaches to Grammar", Center for the Study of Language and Information, Stanford University, California, USA.

SCHREIBER, G., WIELINGA, B., BREUKER, J., (Eds.), (1993), "KADS - A Principled Approach to Knowledge-Based System Development", Academic Press.

SCHREIBER, G., AKKERMANS, H., ANJEWIERDEN, A., DE HOOG, R., SHADBOLT, N., VAN DE VELDE, W., WIELINGA, B., (1999), "Knowledge Engineering and Management: The CommonKADS Methodology", MIT Press, Cambridge, MA, USA.

SCHREIBER, G., WIELINGA, B., AKKERMANS, H., VAN DE VELDE, W., ANJEWIERDEN, A., (1994), "CML: The CommonKADS Conceptual Modelling Language", in Proceedings of the Eighth Knowledge Acquisition Workshop (EKAW–94), Hoegaarden, Belgium, Springer Verlag, pp. 1-25.

SIGART Bulletin, (1991), Special section on integrated cognitive architectures, Vol. 2, No. 4, Aug. 1991.

SIMONIS, H., (1995), "The CHIP System and Its Applications", in Montanari, U., Rossi, F., (Eds.), *Principles and Practice of Constraint Programming*, proceedings of the first international conference on the principles and practice of constraint programming, Lecture Notes in Computer Science Series, Springer Verlag, pp. 643-646.

SISKIND, J. M., (1991), "Screaming Yellow Zonkers", Draft Technical Report of 29th September 1991, supplied with the SCREAMER code distribution 3.20 at `http://linc.cis.upenn.edu/~screamer-tools/home.html`.

SISKIND, J. M., MCALLESTER, D. A., (1993), "Nondeterministic LISP as a Substrate for Constraint Logic Programming", in proceedings of AAAI-93.

SISKIND, J. M., MCALLESTER, D. A., (1994), "SCREAMER: A Portable Efficient Implementation of Nondeterministic Common LISP'", Technical Report IRCS-93-03, Uni. of Pennsylvania Inst. for Research in Cognitive Science.

SLEEMAN, D., RISSAKIS, M., CRAW S., GRANER, N., SHARMA, S., (1995), "Consultant-2: Pre- and Post-processing of Machine Learning Applications", International Journal of Human–Computer Studies, Vol. 43, No. 1, pp. 43-63.

SLEEMAN, D., (1992), "Artificial Intelligence: Achievements and Promises", in (Sleeman & Bernsen, 1992).

SLEEMAN, D., BERNSEN, N. O., (1992), "Artificial Intelligence", Research Directions in Cognitive Science, European Perspectives, Volume 5, Lawrence Erlbaum Associates.

SLEEMAN, D., SHADBOLT, N., (1996), "Report on EPSRC Workshop on Software Assisted Knowledge Acquisition (SAKA)", Abingdon, Oxfordshire.

SLEEMAN, D., WHITE, S., (1996), "A Multistrategy Knowledge Refinement and Acquisition Toolbox: Revisited", Aberdeen University Computing Science Dept. Technical Report AUCS/TR9605, October 1996.

SLEEMAN, D., WHITE, S., (1997), "A Toolbox for Goal-driven Knowledge Acquisition", in Proceedings of the Nineteenth Annual Conference of the Cognitive Science Society, (COGSCI–97), Stanford, California, USA.

SPIVEY, J. M., (1992), "The Z Notation: A Reference Manual", (second edition) Prentice Hall International, Hemel Hempstead, UK.

STEELE, G. L., Jr., (1980), "The Definition and Implementation of a Computer Programming Language Based on Constraints", MIT AI Research Laboratory Technical Report AITR-595, MIT, Cambridge, Massachusetts, USA.

STEELE, G. L. Jr., (1990), "Common LISP the Language", Second Edition, Digital Press, Woburn, MA, USA.

STERLING, L., SHAPIRO, E., (1986), "The Art of Prolog: Advanced Programming Techniques", MIT Press, Cambridge, MA, USA.

SUNDERMEYER, K., (1991), "Knowledge-Based Systems: Terminology and References", Bibligraphisches Institut Wissenschaftsverlag, Mannheim, Germany.

TSANG, E., (1993), "Foundations of Constraint Satisfaction", Computation in Cognitive Science Series, Academic Press, London.

TURING, A. M., (1937), "On Computable Numbers, with an Application to the Entscheidungsproblem", in Proceedings of the London Mathematical Society, Vol. 42(ii), pp. 230-265; correction Vol. 43, pp. 544-546.

USCHOLD, M., (1998), "Knowledge Level Modelling: Concepts and Terminology", Knowledge Engineering Review, Vol. 13, No. 1, pp. 5-29.

VAN HARMELEN, F., BALDER, J., (1993), "$(ML)^2$: A Formal Language for KADS Models of Expertise", in (Schreiber, Wielinga & Breuker, 1993).

VAN HEIJST, G., TERPSTRA, P., WIELINGA, B., SHADBOLT, N., (1992), "Using generalised directive models in knowledge acquisition", in Proceedings of EKAW–92, Springer Verlag.

VAN HENTENRYCK, P., (1989), "Constraint Satisfaction in Logic Programming", MIT Press, Cambridge, MA, USA.

VANLEHN, K. (Ed.), (1988), "Architectures for Intelligence", *Proceedings of the 22nd Carnegie Mellon Symposium on Cognition, 1988*, Lawrence Erlbaum, Hillsdale, New Jersey, USA.

VISSER, P. R. S., JONES, D. M., BENCH-CAPON, T. J. M., SHAVE, M. J. R., (1997), "An Analysis of Ontology Mismatches; Heterogeneity versus Interoperability", AAAI 1997 Spring Symposium on Ontological Engineering, Stanford University, California, USA, pp. 164-172.

WALLACE M. G., NOVELLO, S. and SCHIMPF, J., (1997) "ECLiPSe : A Platform for Constraint Logic Programming", in ICL Systems Journal, Vol 12, Issue 1, May 1997.

WALSH, T., (2000), Personal Communication.

WHITE, S., (1997), "Providing Means-Ends Guidance for Knowledge Acquisition & Problem Solving", PhD Thesis Proposal, Department of Computing Science, University of Aberdeen, Aberdeen, UK.

WHITE, S., SLEEMAN, D., (1998a), "Providing Advice on the Acquisition and Reuse of Knowledge Bases in Problem Solving", in the Proceedings of the Eleventh Knowledge Acquisition Workshop (KAW–98), Banff, Canada.

WHITE, S., SLEEMAN, D., (1998b), "Constraint Handling in Common LISP", Department of Computing Science Technical Report AUCS/TR9805, University of Aberdeen, Aberdeen, UK.

WHITE, S., SLEEMAN, D., (1999), "A Constraint-Based Approach to the Description of Competence", in Fensel, D., Studer, R., (Eds.), Proceedings of the Eleventh European Workshop on Knowledge Acquisition, Modeling, and Management (EKAW–99), LNCS, Springer Verlag, pp. 291-308.

WIELINGA, B. J., AKKERMANS, J. M., SCHREIBER A. TH., (1998), "A Competence Theory Approach to Problem Solving Method Construction", International Journal of Human-Computer Studies, Vol. 49, No. 4.

WIELINGA, B. J., SCHREIBER, A. T., BREUKER, J. A., (1992), "KADS: a modelling approach to knowledge engineering", *Knowledge Acquisition*, Vol 4, No. 1, pp. 5-53.

WILKS, Y., (1997), "Information Extraction as a core language technology: What is IE?", in Pazienza, M-T., (ed.), *Information Extraction*, Springer, Berlin, 1997. Also appeared as technical report CS–95–21, University of Sheffield, Department of Computer Science, 1995.

WINSTON, P. H., (1984), "Artificial Intelligence", Second Edition, Addison-Wesley, Massachusetts, USA, pp. 187-192.

WOLF, R., (1999), "Elicitation of Operational Track Grids", in Fensel, D., Studer, R., (Eds.), Proceedings of the Eleventh European Workshop on Knowledge Acquisition, Modeling, and Management (EKAW–99), LNCS, Springer Verlag, pp. 397-402.

Appendix A: The Extension to SCREAMER

A.1 Type Restrictions

`listpv, conspv, symbolpv, stringpv` [Macro]

Synopsis: (**listpv** ***x***) **|** (**conspv** ***x***) **|** (**symbolpv** ***x***) **|** (**stringpv** *x*)

Description: The macros **listpv**, **conspv**, **symbolpv**, and **stringpv** each return a variable, *z*, constrained to indicate with a boolean value whether *x* is of the given type. For example, the variable returned by **listpv** indicates whether its argument is a list as soon as the argument itself becomes bound. Each of these macros is a specialisation of the **typepv** function.

Examples

```
;;; Create a constraint variable
> (setq x (make-variable))
[432]
;;; Constrain z to be the listp of x
> (setq z (listpv x))
[433]
;;; Bind x to a non-list value
> (make-equal x "hello")
"hello"
;;; Inspect the value of z
> z
NIL

;;; Constrain a to be one of the given values
> (setq a (a-member-ofv '(1 nil t (hello))))
[124 enumerated-domain:(1 NIL T (HELLO))]
;;; Assert a to be a list
> (assert! (listpv a))
NIL
;;; The possible values are now only those which are lists
> a
[124 nonnumber enumerated-domain:(NIL (HELLO))]
```

`typepv` [Function]

Synopsis: (**typepv** *x type*)

Description: This function returns a variable, *z*, constrained to indicate with a boolean value whether *x* is of the given type. The supplied *type* must be one of those acceptable to the Common LISP *typep* function. The output variable *z* does not become bound until both *x* and *type* are bound. However, if *z* is bound to true, *type* is bound, and *x* has an enumerated domain, then all values are eliminated from the domain which are not of the supplied type.

Examples

```
> (setq x (make-variable))
[434]
> (setq ty (make-variable))
[435]
> (setq z (typepv x ty))
[436]
> (make-equal x "jolly")
"jolly"
> (make-equal ty 'string)
STRING
> z
T
```

`a-listv`, `a-consv`, `a-symbolv`, `a-stringv` [Macro]

Synopsis: (**a-listv**) **|** (**a-consv**) **|** (**a-symbolv**) **|** (**a-stringv**)

Description: The macros **a-listv**, **a-consv**, **a-symbolv** and **a-stringv** are commonly used specialisations of the function **a-typed-varv**, which are used to constrain a variable to be either a list, cons, symbol, or string, respectively. See also **a-typed-varv**.

Examples

```
> (setq z (a-listv))
[100]
> (make-equal z 'not-a-list nil)
Warning: (make-equal Z 'NOT-A-LIST) failed
```

```
  Z = [100]; 'NOT-A-LIST = NOT-A-LIST
NIL
```

`a-typed-varv` [Function]

Synopsis: (**a-typed-varv** *type*)

Description: This function returns a variable, *z*, constrained to be of the given LISP type. An attempt to bind *z* to a value which is incompatible with the supplied type will fail.

Examples

```
> (setq z (a-typed-varv 'string))
[97]
> (make-equal z '(1 2) 'failed)
Warning: (make-equal Z '(1 2)) failed
  Z = [97]; '(1 2) = (1 2)
FAILED

> (setq a (a-typed-varv 'string))
[98]
> (assert! (memberv a '(1 two "three")))
NIL
> a
"three"
```

A.2 Boolean Values

`impliesv` [Function]

Synopsis: (**impliesv** *x* *y*)

Description: This function is a simple extension to SCREAMER which returns a boolean variable, *z*, constrained to indicate whether *x* implies *y*. Typically, the implication is asserted, and *y* becomes bound to true as soon as *x* becomes true.

```
;;; Create two boolean constraint variables, x and y
> (setq x (a-booleanv))
[360 Boolean]
> (setq y (a-booleanv))
[361 Boolean]
;;; Assert that x implies y
> (assert! (impliesv x y))
```

```
NIL
;;; Set x to be true
> (make-equal x t)
T
;;; Check that y is bound to true
> y
T

;;; The truth table for implies is easy to generate:
> (setq p (a-booleanv))
[3558 Boolean]
> (setq q (a-booleanv))
[3559 Boolean]
> (setq r (a-booleanv))
[3560 Boolean]
> (assert! (equalv r (impliesv p q)))
NIL
> (all-values (solution (list p '=> q 'is r) (static-ordering
#'linear-force)))
((T => T IS T) (T => NIL IS NIL) (NIL => T IS T) (NIL => NIL IS T))
```

A.3 Expressions

`ifv` [Macro]

Synopsis: (**`ifv`** *`condition expression1 expression2`*)

Description: This macro returns a variable, *z*, constrained to be the value of *expression1* or *expression2*. If the boolean constraint variable *condition* becomes bound to **`t`**, then *z* becomes bound to *expression1*; otherwise *z* becomes bound to *expression2* as soon as *condition* becomes bound to **`nil`**. An important property of **`ifv`** is that neither *expression1* nor *expression2* are actually evaluated until the *condition* becomes bound. This means that the macro can be used in recursive constraint definitions, such as that given below.

Examples

```
;;; Create our own recursive definition of memberv
> (defun my-memberv (m ll)
    (when ll
      (ifv (equalv m (carv ll))
           t
           (my-memberv m (cdrv ll))
           )
      )
    )
MY-MEMBERV
```

```
> (setq x (make-variable))
[11731]
> (setq z (my-memberv 1 x))
[11741]
> (make-equal x '(3 2 1))
(3 2 1)
> z
T
```

`make-equal` [Macro]

Synopsis: (**`make-equal`** *x y* [*failure-expression*])

Description: This macro attempts to constrain *x* to be **`equal`** to *y*. Although the arguments are actually symmetric, *x* is usually a constraint variable, and *y* some new (constrained) value, so that the macro amounts to the equivalent of an assignment in imperative language programming. The macro returns the updated value of *x* if the assertion succeeded, which is now constrained to be **`equal`** to *y*. If the assertion fails, then the *failure-expression* is evaluated and returned. Also, unlike a conventional SCREAMER **`(assert! (equalv x y))`**, a failed **`make-equal`** assertion leaves *x* with the same value after the assertion as it had before the assertion was attempted.

Examples

```
;;; First, an example of make-equal succeeding...
> (setq x (make-variable))
[4]
> (make-equal x '(1 2 3))
(1 2 3)
> x
(1 2 3)

;;; Now an example of the assertion failing...
> (setq a (an-integer-betweenv 1 5))
[10214 integer 1:5 enumerated-domain:(1 2 3 4 5)]
> (make-equal a 9 nil)
Warning: (make-equal A 9) failed
  A = [10214 integer 1:5 enumerated-domain:(1 2 3 4 5)]; 9 = 9
NIL
> a
[10214 integer 1:5 enumerated-domain:(1 2 3 4 5)]
>
```

`setq-domains` [Macro]

Synopsis: (**setq-domains** *var-list expr*)

Description: It is often the case that several different variables in a problem have the same set of domain values. This macro provides a short-hand for the repetitive decalrations of domain values which occurs in these cases. The *var-list* should be an unquoted list of variables. The expression *expr* will be evaluated (once for each variable) and assigned to the variable name. For example, the expression `(setq-domains (a b c) (an-integer-betweenv 0 9))` will ensure that each of the three variables is constrained to be one of the integers 0...9. The eventual values (i.e., bindings) of `a`, `b` and `c` can, of course, be different.

Examples

```
;;;
> (setq-domains (x y z) (a-member-ofv '(foo bar)))
[333 nonnumber enumerated-domain:(FOO BAR)]
> x
[333 nonnumber enumerated-domain:(FOO BAR)]
> y
[332 nonnumber enumerated-domain:(FOO BAR)]
> z
[331 nonnumber enumerated-domain:(FOO BAR)]
```

A.4 Lists and Sequences

`firstv, secondv, thirdv, fourthv` [Macro]

Synopsis: (**firstv** *x*) **|** (**secondv** *x*) **|** (**thirdv** *x*) **|** (**fourthv** *x*)

Description: The macros **firstv**, **secondv**, **thirdv**, and **fourthv** each return a variable, *z*, constrained to be either the first, second, third, or fourth element of a list, respectively. If *x* is already bound to a list value at the time of invocation, then the *value* of the constraint variable *z* is returned, rather than the variable itself. Otherwise *x* must be a constraint variable, and *z* becomes bound as soon as *x* becomes bound. If *x* becomes bound to a value which is not a list, so that the core function **first**, **second**,

third, or **fourth** cannot be properly executed, then a failure is generated. Note that **firstv** is identical to **carv**, except that **firstv** is implemented as a macro and **carv** is implemented as a function.

Examples

```
;;; If the argument is bound the answer is returned directly
> (firstv '(a b c))
A

;;; Create an unbound constraint variable, x
> (setq x (make-variable))
[813]
;;; Constrain z to be the first element of x
> (setq z (firstv x))
[814]
;;; Bind x to a specific value
> (make-equal x '(a b c))
(A B C)
;;; Inspect z
> z
A
```

---

nthv [Function]

Synopsis: (**nthv** *n x*)

Description: The function **nthv** returns a variable, *z*, constrained to be the n^{th} element of the list *x*. Both *n* and *x* may be either a bound value or an unbound constraint variable at the time of function invocation. The argument *n*, when bound, is an integer which can range from 0, which indexes the first element of the list, to (`length` *x*) - 1, which indexes the last element. As soon as *x* becomes bound, the value of *z* is computed. Similarly, if *z* and *n* both become bound before *x*, then *x* is constrained (but not bound) to be a list whose n^{th} element is *z*. If the bindings are such that the index *n* is out of range for the list *x*, or *x* is not a list at all, a failure is generated.

Examples

```
;;; This example extracts the nth element of a list
> (setq x (make-variable))
[823]
> (setq z (nthv 2 x))
[824]
> (make-equal x '(a b c d e f))
(A B C D E F)
```

```
> z
C

;;; This example sets the nth element of a list
> (setq a (make-listv 4))
([396] [395] [394] [393])
> (assert! (equalv (nthv 1 a) 'foo))
NIL
> a
([413] FOO [394] [393])
> (secondv a)
FOO
```

`subseqv` [Function]

Synopsis: (**subseqv** *x start* [*stop*])

Description: The function **subseqv** returns a variable, *z*, constrained to be the subsequence of *x* running from the inclusive index *start* up to the exclusive index *stop*. (The index of the first value in a sequence is zero.) If *stop* is not supplied, the subsequence *z* runs from *start* to the end of *x*. The variable *z* becomes bound as soon as its arguments become bound. Also, if *z* should become bound to a list value, for example before the elements of *x* are all bound, then the elements of *z* are unified with their respective elements of *x*.

Examples

```
;;; When the args are bound, subseqv is the same as subseq
> (subseqv '(1 2 3 4 5 6) 2 4)
(3 4)

;;; Create a constraint variable for the index
> (setq n (make-variable))
[199]
;;; Constrain a to be a subsequence of '(1 2 3 4 5 6)
> (setq a (subseqv '(1 2 3 4 5 6) n))
[201]
;;; Bind n
> (make-equal n 3)
3
;;; Check the value of a
> a
(4 5 6)

;;; Create a list of 6 elements
> (setq x (make-listv 6))
([203] [204] [205] [206] [207] [208])
;;; Create another list of 2 elements
> (setq z (make-listv 2))
```

```
([210] [211])
;;; Say that the elements of z are both integers
> (assert! (integerpv (firstv z))
)
NIL
> (assert! (integerpv (secondv z)))
NIL
> z
([210 integer] [211 integer])
;;; Assert that z are the same as the middle two of x
> (assert! (equalv z (subseqv x 2 4)))
NIL
;;; Check that the elements of z have been unified with x
> x
([203] [204] [205 integer] [206 integer] [207] [208])
>
```

`lengthv` [Function]

Synopsis: (**`lengthv`** *`x`* [*`is-list`*])

Description: The function **`lengthv`** returns a variable, *z*, constrained to be the length of the list *x*. If *x* is already bound at the time of function invocation, then the length of *x* is returned directly. Otherwise *z* becomes bound as soon as *x* becomes bound. Although **`lengthv`** will work with other types of sequences, such as strings and one-dimenional arrays, its propagation properties have been optimised for use with lists. If *z* should become bound before *x* and the optional argument *is-list* is not supplied (or is supplied with a non-**`nil`** value), then *x* is assumed to be a list and becomes bound to a list structure of the length given by *z*. If *z* becomes bound before *x*, and *is-list* is supplied with the value **`nil`**, then no assumptions are made about the sequence type of *x*, but no propagation takes place from *z* to *x* either. Also, if *x* has an enumerated domain of possible values, then the lengths of these sequences are propagated through to *z*.

Examples

```
> (setq a (a-listv))
[104]
> (setq b (lengthv a))
[107]
> (make-equal b 4)
4
> a
([112] [111] [110] [109])
```

```
;;; Create a constraint variable called x
> (setq x (make-variable))
[185]
;;; Constrain len to be the length of x, where x is not a list
> (setq len (lengthv x nil))
[187 integer]
;;; Enumerate the possibilities for x
> (assert! (memberv x '("one" "two" "three" "four")))
NIL
;;; Check the domain of x
> x
[185 nonnumber enumerated-domain:("one" "two" "three" "four")]
;;; Check the domain of len
> len
[187 integer 3:5 enumerated-domain:(3 5 4)]
;;; Bind x
> (make-equal x "three")
"three"
;;; Check that len has also been bound
> len
5
```

---

consv [Function]

Synopsis: (**consv** *x y*)

Description: The function **consv** returns a variable, *z*, constrained to be the **cons** of *x* and *y*. If *y* is bound at the time of function invocation then *z* is immediately bound to the **cons** of *x* and *y* (regardless of whether *x* is bound); otherwise *z* becomes bound to the **cons** of *x* and *y* as soon as *y* becomes bound[1]. If *z* should become bound before *y*, then *x* and *y* become bound to the **car** and **cdr** of *z*, respectively.

Examples

```
;;; Create a constraint variable
> (setq x (make-variable))
[1]
;;; Constrain z to be the cons of the symbol 'g and x
> (setq z (consv 'g x))
[2]
;;; Bind x to a value
> (make-equal x '(1 2 3))
(1 2 3)
;;; Inspect the value of z
> z
```

1.We should also like to bind *z* if *x* is bound and *y* is unbound, so that `(car z)` would return *x*. The problem is that if the tail of *z* becomes bound to a constraint variable because *y* is not bound, then LISP stores *z* as a dotted list pair whose `cdr` is an unbound constraint variable (rather than a list). This dotted list pair cannot later be unified with a list when *y* eventually becomes bound.

```
(G 1 2 3)


;;; Create two constraint variables to be cons'ed
> (setq x (make-variable))
[7]
> (setq y (make-variable))
[8]
;;; Constrain z to be the cons of x and y
> (setq z (consv x y))
[9]
;;; Bind the value of z
> (make-equal z '(a b c))
(A B C)
;;; Check that the values of x and y have been derived
> x
A
> y
(B C)
```

`carv` [Function]

Synopsis: (**carv** *x*)

Description: The function **carv** returns a variable, *z*, constrained to be the *car* (i.e. first element) of a *cons*. (A cons is either a normal list or a dotted list.) If *x* is already bound to a cons value at the time of invocation, then the *value* of the constraint variable *z* is returned, rather than the variable itself. Otherwise *x* must be a constraint variable, and *z* becomes bound as soon as *x* becomes bound. If *z* becomes bound before *x*, then *x* is constrained to be a cons in which *z* is the first element. If *x* becomes bound to a value which is not a cons, so that the Common LISP function *car* cannot be properly executed, then a fail is generated.

Examples

```
;;; Create a variable to contain a list
> (setq x (make-variable))
[13]
;;; Constrain the variable z to be the car of x
> (setq z (carv x))
[14]
;;; Bind x to a value
> (make-equal x '(fee fi fo fum))
(FEE FI FO FUM)
;;; Inspect the value of z
> z
FEE
```

```
;;; Create a variable to contain a dotted pair
> (setq x (make-variable))
[153]
;;; Constrain z to be the first element of the pair
> (setq z (carv x))
[154]
;;; Create a binding for x
> (make-equal x (cons 'one 'two))
(ONE . TWO)
;;; Inspect the value of z
> z
ONE
```

See also the examples for **cdrv**.

---

cdrv [Function]

Synopsis: (**cdrv** *x*)

Description: The function **cdrv** returns a variable, *z*, constrained to be the *cdr* (i.e. the tail) of a *cons* (normally a list). If *x* is already bound to a cons at the time of invocation, then the *value* of the constraint variable *z* is returned, rather than the variable itself. Otherwise *x* must be a constraint variable, and *z* becomes bound as soon as *x* becomes bound. If *z* becomes bound before *x*, then *x* is constrained to be a cons which has *z* as its *cdr*. If *x* becomes bound to a value which is not a *cons*, so that the Common LISP function *cdr* cannot be properly executed, then a fail is generated.

Examples

```
;;; Create a constraint variable to contain a list
> (setq x (make-variable))
[133]
;;; Constrain z to be the tail of the list
> (setq z (cdrv x))
[134]
;;; Create a binding for x
> (make-equal x '(1 2 3 4))
(1 2 3 4)
Inspect the value of z
> z
(2 3 4)

> (setq x (make-variable))
[114]
> (setq head (carv x))
[115]
> (setq tail (cdrv x))
```

```
[116]
> (make-equal head 'g)
G
> (make-equal tail '(h i))
(H I)
> x
(G H I)
```

---

appendv [Function]

Synopsis: (**appendv** x_1 x_2 [... x_n])

Description: The function **appendv** returns a variable, *z*, constrained to be the **append** of all its arguments, unless all the arguments are bound at the time of function invocation, when the value of the append is returned directly. The function applies tail recursion to any arguments x_3 ... x_n supplied. Usually, however, **appendv** will be called with only two arguments, so that *z* is constrained to be the **append** of x_1 and x_2. In such circumstances, when any two of x_1, x_2 and *z* become bound, the third value is deduced. If *z* only should become bound, then x_1 and x_2 are each constrained to be one of the sublists taken from the front or back of *z*, respectively. For example, if the append, *z*, of two lists *x* and *y* is known to be **'(a b c)**, then *x* must be one of **'()**, **'(a)**, **'(a b)**, and **'(a b c)**, and *y* must be one of **'(a b c)**, **'(b c)**, **'(c)**, and **'()**.

Examples

```
;;; Create a constraint variable
> (setq x (make-variable))
[165]
;;; Constrain z to be the append of x and '(3 4)
> (setq z (appendv x '(3 4)))
[166]
;;; Bind x to the list '(1 2)
> (make-equal x '(1 2))
(1 2)
;;; Inspect z
> z
(1 2 3 4)


;;; Create two constraint variables
> (setq x (make-variable))
[167]
> (setq y (make-variable))
[168]
;;; Assert that appending x to y gives the result '(a b c)
> (assert! (equalv (appendv x y) '(a b c)))
```

```
NIL
;;; Inspect x and y - note the enumerated domains
> x
[167 nonnumber enumerated-domain:(NIL (A) (A B) (A B C))]
> y
[168 nonnumber enumerated-domain:((A B C) (B C) (C) NIL)]
;;; Bind x to one of its possible values
> (make-equal x '(a))
(A)
;;; Inspect y, which has now automatically been bound
> y
(B C)
```

`make-listv` [Function]

Synopsis: (**`make-listv`** *n*)

Description: The function **`make-listv`** returns a variable which is constrained to be a list of the length given by *n*. More precisely, the function actually generates a list of length *n* with a different constraint variable taking the place of every element. Note that since a constraint variable is generated for every place in the list, there is a memory overhead in generating large lists using this function. For long lists it is preferable to use **`a-listv`** which does not actually construct a list, but instead checks that the variable is a list once it is instantiated.

Examples

```
> (setq n (make-variable))
[28]
> (setq x (make-listv n))
[29]
> (make-equal n 4)
4
> x
([33] [32] [31] [30])
> (assert! (equalv (nthv 1 x) 'foo))
NIL
> x
([36] FOO [31] [30])
>
```

`all-differentv` [Function]

Synopsis: (**all-differentv** x_1 x_2 ... x_n)

Description: This function returns a boolean variable, *z*, constrained to indicate whether all the supplied arguments have distinct values. This is in effect a symbolic equivalent of the existing numeric SCREAMER constraint predicate '**/=v**', which can be used to constrain its numeric arguments to contain distinct values.

Thus, whilst **(assert! (/=v** x_1 x_2 x_3**))** is equivalent to: **(andv (numberpv** x_1**) (numberpv** x_2**) (numberpv** x_3**)**
(notv (=v x_1 x_2**)) (notv (=v** x_2 x_3**)) (notv (=v** x_1 x_3**)))**,

the symbolic counterpart **(assert! (all-differentv** x_1 x_2 x_3**))** is equivalent to **(andv (notv (equalv** x_1 x_2**)) (notv (equalv** x_2 x_3**)) (notv (equalv** x_1 x_3**)))**.

Examples

```
;;; Clearly all distinct
> (all-differentv 'a 'b 'c)
T

;;; Not yet clear whether the values are distinct
> (all-differentv (a-member-ofv '(a b)) 'b 'c)
[10224 Boolean]

;;; Clearly not all distinct
> (all-differentv 'a 'b 'a)
NIL
```

A.5 Sets and Bags

`set-equalv` [Function]

Synopsis: (**set-equalv** *x* *y*)

Description: The function **set-equalv** returns a boolean variable, *z*, which is constrained to indicate whether its two list arguments are equal if they are interpreted as sets. In other words, as soon as both *x* and *y* become bound, *z* becomes bound to true if *x* and *y* have the same members; otherwise *z* becomes bound to false.

Examples

```
> (setq x '(a b c))
(A B C)
> (setq y (make-variable))
[283]
> (setq same-set (set-equalv x y))
[284 Boolean]
> (make-equal y '(b b a c a a))
(B B A C A A)
> same-set
T
;;; In this example the second set has 4 distinct members
> (set-equalv '(a b c) '(b b a c a d))
NIL
```

`intersectionv` [Function]

Synopsis: (**`intersectionv`** *x y*)

Description: The function **`intersectionv`** returns a variable, *z*, constrained to be the intersection of the two sets *x* and *y*. The output variable *z* becomes bound as soon as both *x* and *y* become bound. If either *x* or *y* has duplicate entries, the redundant entries may or may not appear in the result[2]. Otherwise, if *z* becomes bound before *x* or *y*, then the latter are constrained to contain each of the elements of *z*.

Examples

```
;;; Create a variable x
> (setq x (make-variable))
[11]
;;; Constrain i to be the intersection of x and (a a b c c c)
> (setq i (intersectionv x '(a a b c c c)))
[12]
;;; Bind x
> (make-equal x '(a b d e f))
(A B D E F)
;;; Check that the intersection is now also bound
> i
(B A)

;;; Create a variable x
> (setq x (make-variable))
[107]
;;; Constrain i to be the intersection of (a b c) and x
> (setq i (intersectionv '(a b c) x))
```

2. **`intersectionv`** uses the Common LISP function **`intersection`**, which leaves the duplication issue open to the LISP implementation.

```
[108]
;;; Bind i
> (make-equal i '(a b))
(A B)
;;; Try to bind x in an inconsistent way
> (make-equal x '(c b a) 'failed)
Warning: (make-equal X '(C B A)) failed
  X = [107]; '(C B A) = (C B A)
FAILED
```

`unionv` [Function]

Synopsis: (**`unionv`** *x* *y*)

Description: The function **`unionv`** returns a variable, *z*, constrained to be the union of the two sets *x* and *y*. The output variable *z* becomes bound as soon as both *x* and *y* become bound. If either *x* or *y* has duplicate entries, the redundant entries may or may not appear in the result[3]. Otherwise, if *z* becomes bound before *x* or *y*, then the elements of the latter are constrained to be members of *z*.

```
;;; Create a constraint variable
> (setq x (make-variable))
[145]
;;; Constrain u to be the union of x and (a a b c c c)
> (setq u (unionv x '(a a b c c c)))
[146]
;;; Bind x
> (make-equal x '(a b d e f))
(A B D E F)
;;; Check the binding of u
> u
(F E D A A B C C C)

;;; Create a constraint variable
> (setq x (make-variable))
[149]
;;; Constrain u to tbe the union of x and (a b c)
> (setq u (unionv x '(a b c)))
[150]
;;; Bind u first to be (a b c d)
> (make-equal u '(a b c d))
(A B C D)
;;; Try binding x in an inconsistent way
> (make-equal x '(d e f) 'failed)
Warning: (make-equal X '(D E F)) failed
  X = [149]; '(D E F) = (D E F)
FAILED
```

---

3. **`unionv`** uses the Common LISP function **`union`**, which leaves the duplication issue open to the LISP implementation.

`bag-equalv` [Function]

Synopsis: (**`bag-equalv`** *x y*)

Description: The function **`bag-equalv`** returns a boolean variable, *z*, which is constrained to indicate whether its two list arguments are equal when interpreted as bags. In other words, as soon as both *x* and *y* become bound, *z* becomes bound to true if *x* and *y* not only have the same members, but also the same number of each of those members; otherwise *z* becomes bound to false.

Examples

```
> (setq x '(a a b c c c))
(A A B C C C)
> (setq y (make-variable))
[300]
> (setq same-bag (bag-equalv x y))
[301 Boolean]
> (make-equal y '(c a c b c a))
(C A C B C A)
> same-bag
T
;;; In this example, the second list has an extra c
;;; So the lists are set-equalv, but not bag-equalv
> (bag-equalv '(a b c) '(a b c c))
NIL
```

`subsetpv` [Function]

Synopsis: (**`subsetpv`** *x y*)

Description: The function **`subsetpv`** returns a boolean variable which is constrained to indicate whether $x \subseteq y$ if the two lists *x* and *y* are interpreted as sets. Note that the related function **`proper-subsetv`** can easily be defined as (**`andv`** (**`subsetpv`** *x y*) (**`not`** (**`set-equalv`** *x y*))).

```
;;; State the necessary ingredients for some recipe
> (setq ingredients '(eggs flour milk salt sugar))
(EGGS FLOUR MILK SALT SUGAR)
;;; We don't know what ingredients are avaialble yet
> (setq available (make-variable))
[366]
;;; It is only possible to prepare the recipe if the necessary
```

```
;;; ingredients is a subset of the available ingredients
> (setq can-prepare (subsetpv ingredients available))
[378 Boolean]
;;; State which ingredients are available
> (make-equal available '(salt pepper milk juice eggs sugar rai-
sins flour))
(SALT PEPPER MILK JUICE EGGS SUGAR RAISINS FLOUR)
;;; Check that can-prepare has become bound
> can-prepare
T
```

A.6 Arrays

`make-arrayv` [Function]

Synopsis: (**`make-arrayv`** *d*)

Description: The function **`make-arrayv`** returns a variable which is constrained to be an array with dimensions given by *d*. More precisely, the function actually generates an array with dimensions *d* in which every element is automatically assigned to be a new, unique, unbound constraint variable. Note that since a constraint variable is generated for every place in the array, there is a memory overhead in generating large arrays using this function. The argument *d* can be either a list of integers or a constraint variable which later becomes bound to a list of integers.

Examples

```
;;; d is a constraint variable which holds the array dimensions
> (setq d (make-variable))
[262]
;;; a is constrained to be an array of dimensions d
> (setq a (make-arrayv d))
[263]
;;; Bind d, so that a is created as a 2 by 3 array
> (assert! (equalv d '(2 3)))
NIL
;;; Check the binding of a
> a
#2A(([266] [265] [264]) ([269] [268] [267]))
```

`arefv` [Function]

Synopsis: (**arefv** *array* &rest *indices*)

Description: The function **arefv** returns a variable which is constrained to be the element of *array* at the position given by *indices*. The arguments to **arefv** can be either bound values or constraint variables. This is a constraint-based version of the Common LISP function **aref**.

Examples

```
> (setq a (make-array '(2 3) :initial-contents '((p q r) (s t
u))))
#2A((P Q R) (S T U))
> (setq x (make-variable))
[274]
> (setq y (make-variable))
[275]
> (setq val (arefv a x y))
[276]
> (make-equal x 0)
0
> (make-equal y 1)
1
> val
Q
```

A.7 Objects

`make-instancev` [Function]

Synopsis: (**make-instancev** *class-name* x_1 x_2 ... x_n)

Description: This function returns a variable, *z*, constrained to be an instance of the supplied type. An object instance is created as soon as the class-name becomes bound. If *z* should become bound before the *class-name*, then the class-name is derived, an instance is created from the arguments x_1 ... x_n and the slots of this instance are unified with those of the object *z*.

The function also displays an interesting phenomenon, in that the *identity* of the object is not known until it is actually created, and this can happen much later than at function invocation time, depending on the bindings of the variables. Thus, you cannot **(assert! (equalv (make-instancev 'junk) (make-instancev 'junk)))** because each of the newly created object instances has a different identity, and the equality constraint will always fail. However, you *can* assert **(slots-equalv (make-instancev 'junk) (make-instancev 'junk))** since **slots-equalv** does not check the identity of the objects, but only the equality of the slot values.

Examples

```
> (defclass junk () ((top :initarg :top) (bottom :initarg :bot-
tom)))
#<STANDARD-CLASS JUNK #xF0A3BC>
> (setq a (make-variable))
[7]
> (setq b (make-variable))
[8]
> (setq c (make-variable))
[9]
> (setq d (make-instancev a :top b :bottom c))
[10]
> d
[10]
> (make-equal a 'junk)
JUNK
> d
#<JUNK #xF24880>
> (describe (value-of d))
#<JUNK #xF243F4> is a JUNK:
TOP:                          [8]
BOTTOM:                       [9]
```

`slot-valuev` [Function]

Synopsis: (**slot-valuev** *x y*)

Description: The function **slot-valuev** returns a variable, *z*, constrained to contain the value of the slot *y* of either the object *x* or a constraint variable which will later become bound to such an object. If the object contains no such slot, then a *slot-missing* error is generated, as would occur with **slot-value** in conventional Common LISP. If the slot value is already bound at the time of function invocation, then that value is returned instead of a constraint variable. Otherwise the variable becomes bound as

soon as both the object *x* and the constraint variable contained by the slot *y* become bound. Note that if the value of slot *y* in object *x* is a constraint variable, then using **(setf (slot-value x y) value)** would overwrite the constraint variable, and bypass its associated noticers (which perform value propagation). The macro **setf** should therefore be avoided when working with slots containing constraint variables. Instead, the function **slot-valuev** (or any accessor functions defined) should be used for both reading and writing to the slot, as shown by the example below.

Examples

```
;;; Define a very simple CLOS class
> (defclass person ()
    ((name :accessor name :initarg :name))
    )
#<STANDARD-CLASS PERSON>
;;; Create an instance of the class with the 'name
;;; slot set to a constraint variable
> (setq anon (make-instance 'person :name (make-variable) ))
#<PERSON @ #x544362>
;;; Inspect the object. Note the slot value of name
> (describe anon)
#<PERSON @ #x544362> is an instance of #<STANDARD-CLASS PERSON>:
 The following slots have :INSTANCE allocation:
  NAME   [92]
;;; Constrain name-of-anon to be slot-value 'name of object anon
> (setq name-of-anon (slot-valuev anon 'name))
[92]
;;; Set the slot value
> (make-equal name-of-anon "Fred Bloggs")
"Fred Bloggs"
;;; Inspect the object again. Note the slot value has been set
> (describe anon)
#<PERSON @ #x544362> is an instance of #<STANDARD-CLASS PERSON>:
 The following slots have :INSTANCE allocation:
  NAME   "Fred Bloggs"
;;; Read the value back from the slot
> (slot-valuev anon 'name)
"Fred Bloggs"
;;; Another way of reading the value
> (name anon)
"Fred Bloggs"
```

`classpv` [Function]

Synopsis: (**`classpv`** *x y*)

Description: The function **`classpv`** returns a variable, *z*, constrained to contain a boolean value which indicates whether the object *x* is an instance of the class *y*. If *x* is not an object, or if *y* is bound to a non-existent class, then *z* becomes bound to **`nil`**. The value of *z* becomes bound as soon as both *x* and *y* become bound.

Also, if *z* and *y* are both bound and *x* has an enumerated domain, then the domain values of *x* are reduced such that *z* = **`(classp x y)`** is true. For example, if *y* is the classname **`'junk`**, and *z* becomes bound to **`nil`** (meaning that *x* is not allowed to be an instance of **`junk`**), then all instances of the class **`junk`** will be removed from the domain of *x* (see example below).

Examples

```
;;; Define a simple class
> (defclass junk() ())
#<STANDARD-CLASS JUNK #xF08B9C>
;;; Create a new variable
> (setq x (make-variable))
[1]
;;; Constrain z to indicate whether it is of class 'junk
> (setq z (classpv x 'junk))
[4 Boolean]
;;; Bind x
> (make-equal x (make-instance 'junk))
#<JUNK #xF1EB90>
;;; Inspect z
> z
T

> (setq x (a-member-ofv (list 1 'hello (make-instance 'junk) nil
"hello")))
[17 enumerated-domain:(1 HELLO #<JUNK #xF13440> NIL "hello")]
> (assert! (notv (classpv x 'junk)))
NIL
> x
[17 enumerated-domain:(1 HELLO NIL "hello")]
```

`class-ofv` [Function]

Synopsis: (**`class-ofv`** *x*)

Description: This function returns a constraint variable, *z*, constrained to be the class of the object *x*. The output variable *z* becomes bound as soon as *x* is bound. If *x* has an enumerated domain, then the classes of those values are propagated to the enumerated domain of *z*.

Examples

```
> (defclass junk () ())
#<STANDARD-CLASS JUNK #xF14018>
> (setq x (make-variable))
[31]
> (setq z (class-ofv x))
[32]
> (make-equal x (make-instance 'junk))
#<JUNK #xF16B98>
> z
#<STANDARD-CLASS JUNK #xF14018>
>
```

`class-namev` [Function]

Synopsis: (**`class-namev`** *x*)

Description: This function returns a variable, *z*, constrained to be the class name of the class *x*. As soon as *x* becomes bound, the LISP function *class-name* is applied, and *z* becomes bound to the result. Also, if *x* has an enumerated domain, then the class names of those values are propagated to *z*. Note that *x* must be a class, and not an object. See **`class-ofv`** for deriving the class of a variable constrained to be an object.

Examples

```
> (defclass foo () ())
#<STANDARD-CLASS FOO #xF1F9E4>
> (defclass bar () ())
#<STANDARD-CLASS BAR #xF17528>
> (setq x (make-variable))
[22]
> (setq name (class-namev (class-ofv x)))
[24]
```

```
> (assert! (memberv x (list (make-instance 'foo) (make-instance
'bar))))
NIL
> x
[22 nonnumber enumerated-domain:(#<FOO #xF14ABC> #<BAR
#xF14BE0>)]
> name
[24 nonnumber enumerated-domain:(FOO BAR)]
>
```

`slot-exists-pv` [Function]

Synopsis: (**slot-exists-pv** *x y*)

Description: This function returns a variable, *z*, constrained to be a boolean value which indicates whether the slot *y* exists in the object *x*. The output variable *z* becomes bound as soon as both *x* and *y* become bound. If *x* has an enumerated domain and *y* is bound, then the possible values of *x* are checked for the existence of slot *y*, in an attempt to bind *z* as early as possible. Also, if *z* and *y* are both bound, and *x* has an enumerated domain, then this domain (a set of object instances) may be reduced according to the value of *z* and whether each of the instances contains the slot y. This enables some powerful constraint reasoning; for example, "Constrain object *p* not to have a slot called *name*".

Examples

```
;;; Set up two classes, person and house, with different slots
> (defclass person () (name))
#<STANDARD-CLASS PERSON #xF22AB8>
> (defclass house () (number))
#<STANDARD-CLASS HOUSE #xF11968>
;;; Say that x is either a house or a person
> (setq x (a-member-ofv (list (make-instance 'house)
                              (make-instance 'person))))
[38 nonnumber enumerated-domain:(#<HOUSE #xF22D10> #<PERSON
#xF2331C>)]
;;; SCREAMER+ can't know for sure whether the slot 'name
;;; exists in x, so an unbound Boolean is returned
> (slot-exists-pv x 'name)
[39 Boolean]
;;; But x certainly doesn't contain the slot 'town
> (slot-exists-pv x 'town)
NIL
;;; Assert that x should not contain the slot 'name
> (assert! (notv (slot-exists-pv x 'name)))
NIL
;;; This leaves only one possible value for x...
```

```
> x
#<HOUSE #xF2277C>
>
```

`reconcile` [Function]

Synopsis: (**`reconcile`** *x y*)

Description: This function unifies the constraint variables *x* and *y* by trying to make them **`equal`**. For example, if *x* is constrained to be an integer between 0 and 10, and *y* is constrained to be an integer between 5 and 15, then after (**`reconcile`** *x y*), both *x* and *y* will be constrained to be an integer between 5 and 10. If *x* and *y* contain (or are) objects, the function ensures that all their respective *slot values* are **`equal`**, rather than the variables themselves. If the slots themselves contain objects, then **`reconcile`** is called recursively on these respective slot values.

In the example below, notice that after **`reconcile`** has been invoked, the two objects still retain their separate object *identities* (as evidenced by the identifiers #x825b72 and #x825f6a), but that the slot values have been made **`equal`**. This required a bidirectional passing of data – both from **`chalk`** to **`cheese`** and from **`cheese`** to **`chalk`**.

Examples

```
> (defclass thing ()
   ((colour :initarg :colour)
    (odour :initarg :odour)
    (name :initarg :name)))
#<STANDARD-CLASS THING>
> (setq chalk (make-instance 'thing :colour 'white))
#<THING @ #x7a83fa>
> (setq cheese (make-instance 'thing :odour 'strong))
#<THING @ #x7a883a>
> (reconcile chalk cheese)
T
> (describe chalk)
#<THING @ #x825b72> is an instance of #<STANDARD-CLASS THING>:
 The following slots have :INSTANCE allocation:
  NAME     [10117]
  ODOUR    STRONG
  COLOUR   WHITE
> (describe cheese)
#<THING @ #x825f6a> is an instance of #<STANDARD-CLASS THING>:
 The following slots have :INSTANCE allocation:
  NAME     [10117]
```

```
ODOUR    STRONG
COLOUR   WHITE
```

A.8 Higher Order Functions

`mapcarv` [Function]

Synopsis: (**mapcarv** *f* x_1 x_2 ... x_n)

Description: The function **mapcarv** returns a variable, *z*, constrained to be the **mapcar** of *f* applied to the arguments x_1 ... x_n. Note that since the elements of the lists x_1 ... x_n may be constraint variables, rather than bound values, the function *f* should be a SCREAMER (or SCREAMER+) constraint function and not a "conventional" Common LISP function. This means, for example, that to constrain a variable to be the list of sums of respective elements of the two lists *p* and *q*, you should use **(mapcarv #'+v p q)**, and *not* **(mapcarv #'+ p q)**. See also the description of **constraint-fn**, which, given a conventional LISP function, returns a modified version of the same function which can also deal with constraint variables as arguments.

Note that in order to maximise propagation, we have deviated slightly from the Common LISP definition of **mapcar** in that we expect all the lists x_1 ... x_n to be of the same length. This allows us to propagate the length of any of the lists x_1 ... x_n through to the length of *z* as soon as they become bound. (When the list arguments to **mapcar** are of different lengths, some of the arguments are not used in the result. For example, **(mapcar #'+ '(1 2 3) '(10 20))** returns **'(11 22)**.)

Examples

```
> (setq a (a-listv))
[183]
> (setq b (a-listv))
[186]
> (setq sums (mapcarv #'+v a b))
[195]
> (make-equal a '(1 2 3))
(1 2 3)
> sums
([207 number] [209 number] [211 number])
> b
([205 number] [204 number] [203 number])
```

```
> a
(1 2 3)
> (make-equal b '(10 20 30))
(10 20 30)
> sums
(11 22 33)
```

`maplistv` [Function]

Synopsis: (**`maplistv`** *f* x_1 x_2 ... x_n)

Description: The function **`maplistv`** returns a variable, *z*, constrained to be the **`maplist`** of *f* applied to the arguments x_1 ... x_n (**`maplist`** applies the function *f* to successive **`cdr`**s of the arguments x_1 ... x_2). As with **`mapcarv`**, the function *f* should be a SCREAMER (or SCREAMER+) constraint function and not a "conventional" Common LISP function (see **`constraint-fn`**).

Note that in order to maximise propagation, we have deviated slightly from the Common LISP definition of **`maplist`** in that all the lists x_1 ... x_n supplied to **`maplistv`** must be of the same length.

Examples

```
;;; First a reminder of the Common LISP function...
> (maplist #'(lambda(x) (cons 'head x)) '(1 2 3 4))
((HEAD 1 2 3 4) (HEAD 2 3 4) (HEAD 3 4) (HEAD 4))


;;; Now create a constraint variable
> (setq a (make-variable))
[232]
;;; And constrain b to the the maplist of a with given lambda fn.
> (setq b (maplistv (constraint-fn #'(lambda(x) (cons 'head x)))
a))
[233]
;;; Now bind a
> (make-equal a '(1 2 3 4))
(1 2 3 4)
;;; Check the value of b
> b
((HEAD 1 2 3 4) (HEAD 2 3 4) (HEAD 3 4) (HEAD 4))
```

`everyv, somev, noteveryv, notanyv` [Function]

Synopsis: (**everyv** *f x*) **|** (**somev** *f x*) **|** (**noteveryv** *f x*) **|** (**notanyv** *f x*)

Description: The functions **everyv**, **somev**, **noteveryv**, and **notanyv** are the constraint-based counterparts of the Common LISP functions **every**, **some**, **notevery**, and **notany**. Thus, for example, the function **everyv** returns a variable, *z*, constrained to indicate whether the predicate *f* is true of every element of *x*. In each case, the predicate *f* must already be bound to a constraint function (see **constraint-fn**) at the time of function invocation.

Examples

```
;;; Create a list of length 3
> (setq a (make-listv 3))
([189] [188] [187])
;;; Assert that it contains only integers
> (assert! (everyv #'integerpv a))
NIL
;;; Observe the effect of the assertion
> a
([189 integer] [188 integer] [187 integer])

;;; Create a list of length 3
> (setq a (make-listv 3))
([176] [175] [174])
;;; b indicates whether there is a non-integer in the list
> (setq b (noteveryv #'integerpv a))
[182 Boolean]
;;; Bind the first element of a to a non-integer
> (assert! (equalv (firstv a) 'p))
NIL
;;; Inspect a
> a
(P [175] [174])
;;; Check whether b has been bound
> b
T
```

at-leastv, at-mostv [Macro]

Synopsis: (**at-leastv** *n f* x_1 x_2 ... x_m) **|** (**at-mostv** *n f* x_1 x_2 ... x_m)

Description: The functions **at-leastv** and **at-mostv** each return a boolean variable, *z*, constrained to indicate whether the m-argument predicate *f* is true of the arguments x_1 ... x_m *at least* or *at most n* times, respectively. Note that as with the other higher-order functions described in this section, *f* must be a constraint-based function such as **integerpv** or a function returned by **constraint-fn**.

Note also that the effect of the related notion *strictly-less-than* can be achieved with **(notv (at-leastv** *n f* x_1 x_2 ... x_m **))**. Likewise, the effect of the notion *strictly-greater-than* can be achieved with **(notv (at-mostv** *n f* x_1 x_2 ... x_m **))**.

Examples

```
> (at-leastv 2 #'integerpv (list (make-variable)))
NIL
> (at-leastv 2 #'integerpv (list 1 (make-variable)))
[594 Boolean]
> (at-leastv 2 #'integerpv (list 1 (make-variable) 'p 6))
T
> (at-leastv 2 (constraint-fn #'evenp) (list 3 4 5 6))
T
```

exactlyv [Macro]

Synopsis: (**exactlyv** *n f* x_1 x_2 ... x_m)

Description: The **exactlyv** form returns a boolean variable, *z*, constrained to indicate whether the m-argument predicate *f* is true of the respective elements of the lists x_1 ... x_m exactly *n* times. Note that *f* must be a constraint-based function, such as **integerpv**, **equalv**, or the returned value of a call to **constraint-fn**.

Note that **exactlyv** is defined such that:

(**exactlyv** *n f* x_1 ... x_m) $\Leftrightarrow$ (**andv** (**at-leastv** *n f* x_1 ... x_m) (**at-mostv** *n f* x_1 ... x_m))

Examples

```
> (exactlyv 3 #'equalv '(1 1 2 3 5 8) '(0 1 2 3 4 5))
T

> (exactlyv 1 #'integerpv '(1 a 2 b))
NIL
```

`constraint-fn` [Function]

Synopsis: (**`constraint-fn`** *`f`*)

Description: This function takes the conventional LISP function *f*, and returns *f'*, a modified version of the same function which accepts constraint variables as arguments as well as bound LISP values. The function reflects my approach to SCREAMER+, since most of the functions I provide are simply constraint-based counterparts to frequently used LISP functions. In contrast to the other functions of the SCREAMER+ library, however, the functions returned by **`constraint-fn`** are not optimised for their propagation properties. This means, for example, that **`carv`** is preferable to **`(constraint-fn #'car)`**. On the other hand, **`constraint-fn`** can be used with *any* function known to LISP, providing a good basis for general symbolic constraint-based reasoning.

Examples

```
;;; First, define a simple function which returns a string
> (defun hello (x) (format nil "Hello ~a" x))
HELLO
;;; Test the function
> (hello 'fred)
"Hello FRED"
;;; Get the constraint-based version of HELLO
> (setq hellov (constraint-fn #'hello))
#<closure 0 #xF216E8>
;;; Create a variable for testing
> (setq who (make-variable))
[216]
;;; Constrain hello-string to be the HELLO of who
> (setq hello-string (funcall hellov who))
[217]
;;; Bind who
> (make-equal who 'mary)
MARY
;;; Check the binding of hello-string
```

```
> hello-string
"Hello MARY"
```

`funcallinv` [Function]

Synopsis: (**`funcallinv`** *f* f^{-1} x_1 x_2 ... x_n)

Description: This is an extended version of **`funcallv`** (supplied with SCREAMER) which allows the programmer to supply an inverse mapping function f^{-1} as well as the "forward" mapping function *f*. As with **`funcallv`**, if the arguments x_1 ... x_n become bound, the returned constraint variable, *z*, becomes bound to the result of applying the function *f* to those arguments. If, however, *z* becomes bound first, then f^{-1} is applied to it to derive the list of arguments x_1 ... x_n. Also, if the function *f* applies only to a single argument x_1 (rather than multiple arguments x_1 ... x_n), and this is an unbound constraint variable with an enumerated domain, then the possible values of *z* are computed and recorded as its enumerated domain.

Examples

```
> (setq a (make-variable))
[16]
> (setq b (make-variable))
[17]
> (assert! (equalv b (funcallinv #'reverse #'reverse a)))
NIL
> (make-equal b '(one two three))
(ONE TWO THREE)
> a
(THREE TWO ONE)
>

;;; Demonstrate the propagation of enumerated domains for single
;;; argument functions
> (setq a (an-integer-betweenv 1 3))
[116 integer 1:3 enumerated-domain:(1 2 3)]
;;; b is constrained to be one more than a
> (funcallinv #'1+ #'1- a)
[120 integer 2:4 enumerated-domain:(2 3 4)]
>
```

A.9 Miscellaneous

`formatv` [Function]

Synopsis: (**formatv** *stream fstring* x_1 x_2 ... x_n)

Description: This function returns a variable, *z*, constrained to be the result of applying the LISP function **format** to the arguments *stream*, *fstring* and x_1 ... x_n. As with the Common LISP function, the formatting string *fstring* determines how the rest of the arguments x_1 ... x_n will be used and represented in the result. The stream argument determines what happens to the resulting output. If it is a stream value, then output is sent directly to that stream. If it is set to true (in Common LISP '**t**'), however, the output is sent to standard output, and if it is set to **nil**, the output is returned as a string. The output variable *z* does not become bound until all the arguments are also bound. In the case of stream output, this can have the effect of creating "format daemons", which wait until all their arguments are bound, and then comment upon it by writing to the stream (see example below). In addition, a mechanism is also provided for switching such a format daemon off. If the returned constraint variable becomes bound to **nil** *before* all the arguments to **formatv** become bound, then the output will not be generated.

Examples

```
;;; Example of a format daemon
;;; First, set up a couple of constraint variables
> (setq x (make-variable))
[32]
> (setq y (make-variable))
[33]
;;; Then set up the format daemon itself
> (setq z (formatv t "~%*** ~a ~a ***~%" x y))
[34]
;;; Then bind x and y
> (make-equal x 'hello)
HELLO
> (make-equal y 'world)
*** HELLO WORLD ***
WORLD
>

;;; Illustrate switching off format daemons
;;; First, set up constraint variables as before
```

```
> (setq x (make-variable))
[88]
> (setq y (make-variable))
[89]
;;; Set up the daemon
> (setq z (formatv t "*** ~a ~a ***~%" x y))
[90]
;;; Bind x
> (make-equal x 'hello)
HELLO
;;; Switch off the daemon
> (make-equal z nil)
NIL
;;; Now bind y. Note that the HELLO WORLD message is not printed.
> (make-equal y 'world)
WORLD
>

;;; Format daemons also work while searching for solutions:
> (setq a (a-member-ofv '(1 2 3)))
[69 integer 1:3 enumerated-domain:(1 2 3)]
> (formatv t "A is ~d~%" a)
[70]
> (all-values (solution a (static-ordering #'linear-force)))
A is 1
A is 2
A is 3
(1 2 3)
>
```

`*enumeration-limit*` [Variable]

Synopsis: **`*enumeration-limit*`**

Description: This variable is used to limit the scale of value propagation in SCREAMER+ when it might otherwise become too greedy for memory. The problem is that when considering enumerated domains of constraint variables, there are cases in which the domain size of the "output" constraint variable is larger than the "input" constraint variable. This means that as propagation progresses, domain sizes may increase uncontrollably if unchecked. We have therefore used the variable **`*enumeration-limit*`** to control the sizes of domains. If value propagation results in any constraint variable having a domain size of greater than, **`*enumeration-limit*`**, then the propagation is frozen. It may later proceed from the same point, however, if changes elsewhere result in the domain sizes of the output variables remaining within the stated limit. The limit can be changed by the user, but is currently set at 100.

Examples

```
> *enumeration-limit*
100
```

Appendix B: Details of Meal Design & Scheduling

B.1 The CKRL Knowledge Bases

B.1.1 Background.ckrl

```
ckrl begin
file         "background.ckrl"
user "swhite"
date "28/02/98"
origin ""
algorithm "editor"
version 2.0;

################################################################################
# Sorts
################################################################################

defsort posint (range (integer (0:*)));

################################################################################
# Background Knowledge for the Constraint Satisfier and Design Analyst
################################################################################

defconcept dish
relevant (main, carbohydrate, vegetable, cost, dietary-points);

defconcept dish-component
relevant (dietary-points, cost, preparation-time);

defconcept main superconcept dish-component;
defconcept meat superconcept main;
defconcept vegetarian superconcept main;
defconcept fish superconcept main;
defconcept vegetable superconcept dish-component;
defconcept carbohydrate superconcept dish-component;


################################################################################
# Relations
################################################################################

defrelation includes conceptref dish, conceptref ingredient;

################################################################################
# Background Knowledge for the Scheduler
################################################################################

################################################################################
# Concepts
################################################################################

defconcept task
relevant (duration, start-after, end-before);

defconcept resource-type
relevant (how-many);
```

```
################################################################################
# Properties
################################################################################

defproperty dietary-points sortref posint;
defproperty cost sortref integer;
defproperty preparation-time sortref posint;

defproperty duration sortref posint;
defproperty start-after sortref posint;
defproperty end-before sortref posint;
defproperty how-many sortref posint;

################################################################################
# Relations
################################################################################

defrelation uses conceptref task, conceptref resource-type, sortref posint;
defrelation precedes conceptref task, conceptref task;
defrelation subtask-of conceptref task, conceptref task;

end ckrl
```

B.1.2 Derivation.ckrl

```
ckrl begin
file "derivation.ckrl"
user "swhite"
date "21/02/98"
origin "xemacs"
algorithm "editor"
version 2.0;

################################################################################
# This file contains rules for deriving dish attribute values
################################################################################

################################################################################
# Rules
################################################################################

defrule sat-meat
when(conceptref dish ?x)
if(?x.meat eq true)
then(?x.animal = true);

defrule sat-seafood
when(conceptref dish ?x)
if(?x.seafood eq true)
then(?x.animal = true);

defrule sat-soup
when(conceptref dish ?x)
if(conceptof(?x, soup))
then(?x.warm = true);

defrule single-item
when    (conceptref dish ?x)
if      (conceptof(?x, dish))
then    (?x.items = 1);

end ckrl
```

B.1.3 Dish-atts.ckrl

```
ckrl begin
file "dish-atts.ckrl"
user "swhite"
```

```
date "21/02/98"
origin "xemacs"
algorithm "editor"
version 2.0;

##############################################################################
# This file contains the definitions of attributes used for describing
#  dishes
##############################################################################

##############################################################################
# Properties
##############################################################################

defproperty main conceptref main;
defproperty carbohydrate conceptref carbohydrate;
defproperty vegetable conceptref vegetable;

defproperty points sortref posint;
defproperty cost sortref posint;
defproperty preparation-time sortref posint;

end ckrl
```

B.1.4 Dish-Components.ckrl

```
ckrl begin
file "dish-components.ckrl"
user "swhite"
date "07/06/99"
origin ""
algorithm "editor"
version 2.0;

##############################################################################
# Main Components
##############################################################################

definstance lamb-chop conceptref meat
(dietary-points 7, cost 4, preparation-time 20);
definstance gammon-steak conceptref meat
(dietary-points 10, cost 4, preparation-time 15);
definstance sausages conceptref meat
(dietary-points 8, cost 2, preparation-time 10);
definstance chicken-kiev conceptref meat
(dietary-points 8, cost 3, preparation-time 30);

# Fish

definstance plaice conceptref fish
(dietary-points 8, cost 4, preparation-time 15);

# Vegetarian

definstance quorn-taco conceptref vegetarian
(dietary-points 6, cost 3, preparation-time 10);

##############################################################################
# Carbohydrate Components
##############################################################################

definstance jacket-potato conceptref carbohydrate
(dietary-points 5, cost 1, preparation-time 60);
definstance french-fries conceptref carbohydrate
(dietary-points 6, cost 1, preparation-time 15);
definstance rice conceptref carbohydrate
(dietary-points 6, cost 1, preparation-time 15);
definstance pasta conceptref carbohydrate
(dietary-points 2, cost 1, preparation-time 10);
```

```
################################################################################
# Vegetable Components
################################################################################

definstance runner-beans conceptref vegetable
(dietary-points 0, cost 1, preparation-time 10);
definstance carrots conceptref vegetable
(dietary-points 0, cost 1, preparation-time 15);
definstance cauliflower conceptref vegetable
(dietary-points 0, cost 1, preparation-time 10);
definstance peas conceptref vegetable
(dietary-points 2, cost 1, preparation-time 3);
definstance sweetcorn conceptref vegetable
(dietary-points 2, cost 1, preparation-time 5);

end ckrl
```

B.1.5 Dishes.ckrl

```
# This file was generated automatically on 02-MAR-00 at 09:01
# Any changes you make here may be lost
ckrl begin
file "kbs/dishes.ckrl"
user "swhite"
date "02-MAR-00"
origin "MUSKRAT"
algorithm "designer/dish generator"
version 2.0;
include "kbs/recipes";
definstance dish-1 conceptref dish
    (
     main QUORN-TACO,
     carbohydrate PASTA,
     vegetable SWEETCORN
    );
definstance dish-2 conceptref dish
    (
     main QUORN-TACO,
     carbohydrate PASTA,
     vegetable PEAS
    );
definstance dish-3 conceptref dish
    (
     main QUORN-TACO,
     carbohydrate PASTA,
     vegetable CAULIFLOWER
    );
definstance dish-4 conceptref dish
    (
     main QUORN-TACO,
     carbohydrate PASTA,
     vegetable CARROTS
    );
definstance dish-5 conceptref dish
    (
     main QUORN-TACO,
     carbohydrate PASTA,
     vegetable RUNNER-BEANS
    );
definstance dish-6 conceptref dish
    (
     main QUORN-TACO,
     carbohydrate RICE,
     vegetable SWEETCORN
    );
definstance dish-7 conceptref dish
    (
     main QUORN-TACO,
     carbohydrate RICE,
     vegetable PEAS
    );
```

```
definstance dish-8 conceptref dish
    (
     main QUORN-TACO,
     carbohydrate RICE,
     vegetable CAULIFLOWER
    );
definstance dish-9 conceptref dish
    (
     main QUORN-TACO,
     carbohydrate RICE,
     vegetable CARROTS
    );
definstance dish-10 conceptref dish
    (
     main QUORN-TACO,
     carbohydrate RICE,
     vegetable RUNNER-BEANS
    );
definstance dish-11 conceptref dish
    (
     main QUORN-TACO,
     carbohydrate FRENCH-FRIES,
     vegetable SWEETCORN
    );
definstance dish-12 conceptref dish
    (
     main QUORN-TACO,
     carbohydrate FRENCH-FRIES,
     vegetable PEAS
    );
definstance dish-13 conceptref dish
    (
     main QUORN-TACO,
     carbohydrate FRENCH-FRIES,
     vegetable CAULIFLOWER
    );
definstance dish-14 conceptref dish
    (
     main QUORN-TACO,
     carbohydrate FRENCH-FRIES,
     vegetable CARROTS
    );
definstance dish-15 conceptref dish
    (
     main QUORN-TACO,
     carbohydrate FRENCH-FRIES,
     vegetable RUNNER-BEANS
    );
definstance dish-16 conceptref dish
    (
     main QUORN-TACO,
     carbohydrate JACKET-POTATO,
     vegetable SWEETCORN
    );
definstance dish-17 conceptref dish
    (
     main QUORN-TACO,
     carbohydrate JACKET-POTATO,
     vegetable PEAS
    );
definstance dish-18 conceptref dish
    (
     main QUORN-TACO,
     carbohydrate JACKET-POTATO,
     vegetable CAULIFLOWER
    );
definstance dish-19 conceptref dish
    (
     main QUORN-TACO,
     carbohydrate JACKET-POTATO,
     vegetable CARROTS
    );
definstance dish-20 conceptref dish
    (
```

```
     main QUORN-TACO,
     carbohydrate JACKET-POTATO,
     vegetable RUNNER-BEANS
    );

end ckrl
```

B.1.6 Preferences.ckrl

```
defrelation like-component conceptref dish-component;
defrelation like-dish conceptref dish, sortref integer;

defrule like-meat
when (conceptref meat ?x)
if true
then (like-component(?x));

defrule like-quick
when (conceptref dish-component ?x)
if (?x.preparation-time < 15)
then (like-component(?x));

# I award 5 points for a main component which I like
# 3 points for a carbohydrate, and 2 points for a vegetable

defrule like-main
when (conceptref dish ?d)
if (like-component (?d.main))
then (like-dish(?d, 5));

defrule like-carbohydrate
when (conceptref dish ?d)
if (like-component (?d.carbohydrate))
then (like-dish(?d, 3));

defrule like-vegetable
when (conceptref dish ?d)
if (like-component (?d.vegetable))
then (like-dish(?d, 2));
```

B.1.7 Recipes.ckrl

```
ckrl begin

###############################################################################
# Header
###############################################################################

file "recipes.ckrl"
user "swhite"
date "10th June 1999"
origin ""
algorithm "editor"
version 2.0;

###############################################################################
# Original version for menus stems from 26th September, 1996                 #
#                           -- RECIPES --                                     #
#                                                                             #
###############################################################################

###############################################################################
# Prepare Lamb Chop
###############################################################################

definstance prepare-lamb-chop conceptref task;
definstance preheat-grill conceptref task
(duration 5);
```

```
definstance grill-lamb-chop conceptref task
(duration 15);

deffacts subtask-of(preheat-grill, prepare-lamb-chop),
 subtask-of(grill-lamb-chop, prepare-lamb-chop);

deffacts uses(grill-lamb-chop, grill, 1);

deffacts precedes(preheat-grill, grill-lamb-chop);

##############################################################################
# Prepare Gammon Steak
##############################################################################

definstance prepare-gammon-steak conceptref task
(duration 15);

##############################################################################
# Prepare Plaice
##############################################################################

definstance prepare-plaice conceptref task
(duration 15);

##############################################################################
# Prepare Sausages
##############################################################################

definstance prepare-sausages conceptref task
(duration 10);

##############################################################################
# Prepare Chicken Kiev
##############################################################################

definstance prepare-chicken-kiev conceptref task
(duration 30);

##############################################################################
# Prepare Quorn Taco
##############################################################################

definstance prepare-quorn-taco conceptref task
(duration 10);

##############################################################################
# Prepare Jacket Potato
##############################################################################

definstance prepare-jacket-potato conceptref task
(duration 60);

##############################################################################
# Prepare French Fries
##############################################################################

definstance prepare-french-fries conceptref task
(duration 15);

##############################################################################
# Prepare Rice
##############################################################################

definstance prepare-rice conceptref task
(duration 15);

##############################################################################
# Prepare Pasta
##############################################################################

definstance prepare-pasta conceptref task
        (duration 10);
definstance boil-pasta-water conceptref task
        (duration 3);
```

```
definstance cook-pasta conceptref task
        (duration 7);

deffacts subtask-of(boil-pasta-water, prepare-pasta),
         subtask-of(cook-pasta, prepare-pasta);

deffacts uses(cook-pasta, saucepan, 1),
         uses(cook-pasta, hob-ring, 1),
         uses(boil-pasta-water, kettle, 1);

deffacts precedes(boil-pasta-water, cook-pasta);

##############################################################################
# Prepare Runner Beans
##############################################################################

definstance prepare-runner-beans conceptref task
(duration 10);
definstance cook-beans conceptref task
        (duration 7);
definstance boil-bean-water conceptref task
        (duration 3);

deffacts subtask-of(boil-bean-water, prepare-runner-beans),
         subtask-of(cook-beans, prepare-runner-beans);

deffacts uses(cook-beans, saucepan, 1),
         uses(cook-beans, hob-ring, 1),
         uses(boil-bean-water, kettle, 1);

deffacts precedes(boil-bean-water, cook-beans);

##############################################################################
# Prepare Carrots
##############################################################################

definstance prepare-carrots conceptref task
(duration 15);

##############################################################################
# Prepare Cauliflower
##############################################################################

definstance prepare-cauliflower conceptref task
(duration 10);

##############################################################################
# Prepare Peas
##############################################################################

definstance prepare-peas conceptref task
(duration 3);

##############################################################################
# Prepare Sweetcorn
##############################################################################

definstance prepare-sweetcorn conceptref task
(duration 5);


##############################################################################
# Rules
##############################################################################

# Make sure that all tasks start after the beginning of the time line

defrule schedule-genesis
when(conceptref task ?t)
if(conceptof(?t, task))
then(?t.start-after = 0);

end ckrl
```

B.1.8 Resources.ckrl

```
ckrl begin

file "resources.ckrl"
user "swhite"
date "28/02/98"
origin ""
algorithm "editor"
version 2.0;

##############################################################################
# Instances of Resources and Tasks
##############################################################################

# Resource Types

definstance oven conceptref resource-type
(how-many 1);
definstance saucepan conceptref resource-type
(how-many 4);
definstance deep-fat-frier conceptref resource-type
(how-many 1);
definstance frying-pan conceptref resource-type
(how-many 1);
definstance hob-ring conceptref resource-type
        (how-many 2);
definstance kettle conceptref resource-type
        (how-many 1);
definstance grill conceptref resource-type
        (how-many 1);

# Tasks
# Since tasks can be subtasks of other tasks, and tasks may be reused by
# other tasks, the structure is an acyclic directed graph

definstance root-task conceptref task
(start-after 0);

end ckrl
```

B.1.9 Task.ckrl

This is the CKRL file that is generated as the output of the dish designer. It contains the specification of the dish to be scheduled by the scheduler.

```
defrelation subtask-of conceptref task, conceptref task;
deffacts subtask-of(prepare-lamb-chop, root-task),
         subtask-of(prepare-runner-beans, root-task),
         subtask-of(prepare-pasta, root-task);
```

B.2 Translating CKRL to LISP

This section provides an overview of how the knowledge used by the two example problem solvers is encoded in CKRL, and how the CKRL knowledge is translated to LISP, the implementation language of the problem solvers. Note that although the translation is explained in terms of the meal planning example, the mechanisms it employs are entirely generic, and could be applied to any domain.

B.2.1 Concepts and Properties

In chapter 6, I used descriptions of a number of dish components. These were encoded by first devising a concept hierarchy for the various ingredient types, then creating instances which represent the given components. The hierarchy which I devised introduced the concept `dish-component` with the three properties (frame slots), `dietary-points`, `cost` and `preparation-time`. These properties are inherited by the three subconcepts `vegetable-component`, `carbohydrate-component` and `main-component`. The latter is further subdivided into three more specialised concepts, namely `meat-component`, `fish-component`, and `vegetarian-component` (see figure 28 on page 82).

In CKRL, the hierarchy is encoded as follows:

```
defconcept dish-component
        relevant (dietary-points, cost, preparation-time);

defconcept main-component superconcept dish-component;
defconcept meat-component superconcept main-component;
defconcept vegetarian-component superconcept main-component;
defconcept fish-component superconcept main-component;
defconcept vegetable-component superconcept dish-component;
defconcept carbohydrate-component superconcept dish-component;

defproperty dietary-points sortref posint;
defproperty cost sortref posint;
defproperty preparation-time sortref posint;
```

Notice that the values allowable for each of the the properties is given by reference to a *sort*. This example references the sort `posint`, the set of non-negative integers.

Mapping *Concepts* and *Properties* to LISP

The frame-based nature of CKRL concepts suggests a straightforward mapping to equivalent data structures implemented using CLOS, the object-oriented feature of Common LISP. Under such a scheme, CKRL concepts and their respective properties are mapped to Common LISP classes with associated slots. Likewise, a concept *instance* in CKRL becomes an instance of the corresponding LISP class. This is not only an elegant translation mechanism, it is also a practical one – the semantics of CKRL inheritance are automatically realised by the inheritance mechanisms of CLOS.

Recall from the previous section that a CKRL `dish-component` was defined as a concept with three properties: `dietary-points`, `cost`, and `preparation-time`. The translation to CLOS should therefore generate a *class* called `dish-component`, with three *slots* corresponding to the three properties in the CKRL model. The following code fragment, produced by the translator, confirms that this is the case:

```
;;; Define the class dish-component
(defclass dish-component()
  (
    (slot-names :accessor :slot-names
                :initform '(object-name points cost preptime)
                :allocation :class)
    (object-name :accessor :object-name :initarg :object-name
:initform (screamer::make-variable))
    (dietary-points :accessor :dietary-points :initarg :die-
tary-points :initform (screamer::make-variable))
    (cost :accessor :cost :initarg :cost :initform (scream-
er::make-variable))
    (preparation-time :accessor :preparation-time :initarg
:preparation-time :initform (screamer::make-variable))
    )
  )
```

Notice that two additional slots, `slot-names` and `object-name` are also defined. These extra slots have been added by the translator to increase the reflective capabilities of the class. As a result of this addition, an instance of the object is provided with easy access to its own name, as well as the names of the slots it contains.

CKRL subconcepts are translated as CLOS subclasses. For example, the translator produces the following CLOS definition of `dish-component`'s subconcept `main`:

```
;;; Define the class main
(defclass main(dish-component)
  (
    (slot-names :accessor :slot-names
                :initform '(object-name)
                :allocation :class)
    (object-name :accessor :object-name :initarg :object-name
:initform (screamer::make-variable))
    )
  )
```

Notice that the additional slots `slot-names` and `object-name` have again been added, but no other slots have been defined. This is because the slots `dietary-points`, `cost`, and `preparation-time` have been inherited from the parent class

`dish-component`; and `slot-names` and `object-name` are always added as a matter of course.

As you would expect, further subconcepts are similarly defined as subclasses of the new class. For example, the CKRL subconcept `meat` is translated as a subclass of the CLOS class `main`.

B.2.2 Instances

Once the concept hierarchy for our scenario is defined, the component instances can be created as follows:

```
definstance lamb-chop conceptref meat
  (dietary-points 7, cost 4, preparation-time 20);
definstance gammon-steak conceptref meat
  (dietary-points 10, cost 4, preparation-time 15);
...
definstance quorn-taco conceptref vegetarian
  (dietary-points 6, cost 3, preparation-time 10);
definstance jacket-potato conceptref carbohydrate
  (dietary-points 5, cost 1, preparation-time 60);
...
definstance peas conceptref vegetable
  (dietary-points 2, cost 1, preparation-time 3);
definstance sweetcorn conceptref vegetable
  (dietary-points 2, cost 1, preparation-time 5);
```

Mapping *Instances* to LISP

When translating the definition of an instance, the translator creates an instance of the corresponding class, in which the slots contain the required values. For example, the following code fragment creates `lamb-chop`, an instance of a `meat` component:

```
;;; Generate instance lamb-chop

(pushnew
  (setq lamb-chop
    (make-instance 'meat
      :object-name 'lamb-chop
      :dietary-points 7
      :cost 4
      :preparation-time 20
      )
    )
  *instances* :test #'objects-equal
  )
```

Notice that the created object has been saved as the value of the variable `lamb-chop` for the sake of convenience, as well as being added to a global list of known instances.

(The function `pushnew` actually tests whether the value is already contained in the list before adding it.)

B.2.3 Sorts

In the scenario described, it makes no sense to allow costs, dietary-points, and preparation-times to be assigned negative values. For the sake of simplicity, I also decided to restrict them to integer values. I therefore defined a sort for non-negative integers, which I called `posint`:

```
defsort posint (range (integer (0:*)));
```

The definition states that the only allowable values are integers in the open range whose smallest value is zero, i.e., the non-negative integers.

Mapping *Sorts* to LISP

The mapping from CKRL sorts into LISP is surprisingly straightforward. Common LISP already contains a *type hierarchy*, to which the programmer can add new types. A new type is defined by a *type specifier*. Type specifiers are often specialisations of regular LISP data structures, such as integers, arrays, atoms and lists. The declaration of a CKRL sort, when translated, therefore becomes the definition of a Common LISP type.

For example, an integer which is not less than zero (the CKRL sort `posint`) is defined as follows:

```
;;; Define a new type
(deftype posint ()
  '(integer 0 *)
  )
```

Once a new type has been defined, it can be enforced with the LISP special form `the`:

```
> (deftype posint ()
   '(integer 0 *))
POSINT
> (the posint 4)
4
> (the posint -4)
Error: object "-4" is not of type "POSINT".
  [condition type: PROGRAM-ERROR]
```

```
Restart actions (select using :continue):
 0: prompt for a new object.
 1: Return to Top Level (an "abort" restart)
[1c] >
```

B.2.4 Relations and Facts

The evaluation of dish preferences, used in the second phase of the designer, is the result of the combination of the likes and dislikes of their component parts. The approval of a component can be represented as a CKRL relation:

```
defrelation like-component conceptref dish-component;
```

Similarly, the approval of a whole dish can be expressed as a relation, and further qualified with an integer. The higher the value of the integer, the greater the fondness for the dish:

```
defrelation like-dish conceptref dish, sortref integer;
```

Whenever components are known to be liked, then that knowledge can be asserted as a set of CKRL facts:

```
deffacts like-component(lamb-chop),
         like-component(jacket-potato),
         ...
         like-component(peas);
```

Also, the scheduler needs to be provided with knowledge about the tasks to be scheduled and the resources used. This can be expressed as a relation which binds a task instance with the number of resources of a particular type:

```
defrelation uses conceptref task,
                 conceptref resource-type,
                 sortref posint;
```

For example, the following fact states that the task instance `boil-pasta-water` (which should be suitably defined) uses one kettle:

```
deffacts uses(boil-pasta-water, kettle, 1);
```

Mapping *Relations* and *Facts* to LISP

There is no LISP data structure which clearly corresponds to the CKRL notions of relations and facts. However, since a fact is a tuple of entities, it can easily be implemented with LISP lists. The translator produces no output which corresponds to a `defrelation` construct, but instead maintains a record of the types of entity expected by that relation. Whenever a fact is subsequently encountered by the translator, it uses its knowledge to restrict the types of entity contained in the tuple.

For example, the CKRL fact:

```
deffacts uses(grill-lamb-chop, grill, 1);
```

is translated to:

```
;;; Assert a "uses" fact
(assert-fact
  '(uses
    (the task grill-lamb-chop)
    (the resource-type grill)
    (the posint 1)
    )
  )
```

where the function assert-fact adds a tuple to a “fact base” abstract data type.

B.2.5 Rules

Even if factual knowledge is not expressed explicitly, it can still be derived through *default* behaviour, expressed using CKRL rules. For example, the following rule states a preference for components which can be prepared quickly:

```
defrule like-quick
when (conceptref dish-component ?x)
  if (?x.preparation-time < 15)
  then (like-component(?x));
```

Another rule assigns weights to a dish, according to the relative merits of its components. In the three CKRL rules given below, I have stated that the main component of a dish should be assigned a weight of 5 points, compared to those for the carbohydrate and vegetable components of 3 and 2 points respectively.

```
defrule like-main
when (conceptref dish ?d)
```

```
  if (like-component (?d.main))
  then (like-dish(?d, 5));

defrule like-carbohydrate
when (conceptref dish ?d)
  if (like-component (?d.carbohydrate))
  then (like-dish(?d, 3));

defrule like-vegetable
when (conceptref dish ?d)
  if (like-component (?d.vegetable))
  then (like-dish(?d, 2));
```

Note that the elegance of the representation used is not an issue: it is common for problem solvers to use somewhat idiosyncratic representations. This can arise, for example, due to limitations of the representation language, or the inexperience of the developer. The central issue is not what the representation is at the problem solving level, but *whether one can express the requirements of the representation at the meta-level.*

Mapping *Rules* to LISP

A CKRL rule is a short piece of procedural knowledge which is applied to instances of a particular concept. When fired, a rule can either modify the property values of a concept instance or assert a new fact. Since a rule involves iteration over many concept instances, I decided to map them to conventional LISP functions[1]. The functions generated use a small library of other functions to perform common operations such as iterating over instances of a particular concept, matching some known facts, or asserting a new fact.

As an example, the rule which expresses a preference for dish-components which can be prepared quickly is translated into LISP as follows:

```
;;; Define rule LIKE-QUICK
(defun like-quick (&aux (retval nil))
  (foreach dish-component ?x
           (progn (if (and (< (:preparation-time ?x) 15))
                      (setq retval
                            (or (assert-fact
                                 (list
                                  'like-component
                                  (:object-name ?x)))
                                retval)))))
  retval)
```

1.If they were to be applied to only one concept instance, then a CLOS method would also have been a good candidate for the implementation.

```
(pushnew 'LIKE-QUICK *rules* :test #'equal)
```

It does not matter that the LISP version is somewhat obfuscated, because it has been automatically translated for subsequent *machine use*. Despite this, I believe that a LISP programmer who is also familar with the much more readable CKRL equivalent would still be able to recognise the basic pattern of the reasoning.

Notice that the rule is also remembered by the global variable `*rules*`.

B.3 The Meta–Problem-Solver

The following file is the meta–problem-solver I developed for the designer and scheduler of chapter 6. Note that the constraint assertions that are particularly pertinent to the text of chapter 6 have been highlighted in bold.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This file contains a meta-problem solver for the dish design problem
;;; Written by Simon White, November 1998
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; To inspect a plausible dish design:
;;; 1. Load this file
;;; 2. Call (meta::designer)
;;; The returned value is an object containing only plausible slot values
;;;
;;; To inpect a plausible schedule:
;;; 3. Call (meta:scheduler)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Meta problem solving in the dish domain
;;;
;;; There's no such thing as a cheap fish dish
;;; There's no such thing as a quick meal with a jacket potato
;;; There's no such thing as a healthy meal with gammon and french fries
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(unless (find-package :meta)
  (make-package :meta)
  )

(in-package :meta)

#+unix (require :screamer+ "~swhite/screamer+/screamer+")
#-unix (require :screamer+ "C:/allegro/screamer+/screamer+")

(use-package :screamer+)

(unless (find-package :ckrl2clos)
  (load "ckrl2clos")
  )
;;; Need to use the ckrl2clos::knowledge-base class and all its methods
(use-package :ckrl2clos)

(import '(knowledge::main
          knowledge::carbohydrate
          knowledge::vegetable
          knowledge::cost
```

```
          knowledge::preparation-time
          knowledge::dietary-points
          ckrl2clos::*instances*
          ckrl2clos::read-ckrl
          ckrl2clos::knowledge-base
          common-lisp-user::true
          screamer::variables-in))


;;; The default place to look for CKRL files
(defvar *ckrl-dir* "~swhite/dish-designer/kbs/*.ckrl")

(defvar *main* nil)
(defvar *carbo* nil)
(defvar *veg* nil)

(defparameter *ingredients* nil)

;;; Two variables for holding instances of meta problem solvers
(defvar *dish-designer*)
(defvar *dish-scheduler*)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Classes and methods for checking role assignment.
;;;
;;; The basic scheme is that each role is defined as an object CLASS
;;; and the characteristics of the role are realised by constraints
;;; asserted automatically at creation time.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This class describes a problem solver
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass problem-solver ()
  (
   (name :initarg :name :accessor :name)
   (input-roles :initarg :input-roles :accessor input-roles)
   (implementation-lang :initarg :implementation-lang)
   )
  (:default-initargs :implementation-lang 'common-lisp)
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This class describes input roles to problem solvers
;;;
;;; The class role defines no extra slots compared to its
;;; superclass knowledge-base. However, it ensures that each of the slots
;;; (apart from filename) is initialised with a constraint variable
;;; This is an abstract class, reused by specific roles.
;;; It introduces constraint-variables into the class hierarchy for roles.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass role (knowledge-base) ()
  (:default-initargs :concepts (make-variable)
                     :instances (make-variable)
                     :properties (make-variable)
                     :relations (make-variable)
                     :rules (make-variable)
                     :facts (make-variable)
                     :functions (make-variable)
                     :sorts (make-variable)
                     :nominals (make-variable)
                     )
  )

(defmethod initialize-instance :after ((x role) &rest args)
  (when (and (slot-boundp x 'ckrl2clos::filename)
             (bound? (slot-value x 'ckrl2clos::filename)))
    (ckrl2clos::parse-ckrl x)
    )
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;; This class describes an ontology role.
;;;
;;; According to this definition, an ontology must contain at least one
;;; concept
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass ontology (role) ())

(defmethod initialize-instance :after ((x ontology) &rest args)
  (let (
        (concepts (make-variable))
        )

    (assert! (equalv concepts (slot-valuev x 'ckrl2clos::concepts)))

    ;; Assert that an ontology-role must contain at least one concept
    (assert! (notv (equalv nil concepts)))
    )
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; A dish-ontology-role is an ontology role that mentions dishes
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass dish-ontology (ontology) ())

(defmethod initialize-instance :after ((x dish-ontology) &rest args)
  (let (
        (concepts (make-variable))
        (concept-names (make-variable))
        )
    (assert! (equalv concepts (slot-valuev x 'ckrl2clos::concepts)))

   (assert! (equalv concept-names (mapcarv (constraint-fn #'caar) concepts)))

    ;; Assert that one of those concepts must be "dish"
    (assert! (memberv "dish" concept-names))

    ;; Assert that one of them must also be "dish-component"
    (assert! (memberv "dish-component" concept-names))
    )
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; A component-instances-role is a role that contains at least one instance
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass component-instances-kb (role)
  ()
  )

(defmethod initialize-instance :after ((x component-instances-kb) &rest args)
  (let (
        (instance-names (ckrl2clos:ckrl-instances x))
        )

    ;; Assert that there is at least one instance
    (assert! (notv (equalv nil instance-names)))

       (ifv (eqv (ckrl2clos:ckrl-instances x) nil)
         (format t " - ~a CANNOT fulfil this role because it contains no
instances~%" (file-namestring (value-of (ckrl2clos:ckrl-filename x)))))
    )
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; A design preferences role is a role that contains at least one rule
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass design-preferences-kb (role)
  ()
  )

(defmethod initialize-instance :after ((x design-preferences-kb) &rest args)
```

```
  (assert! (notv (equalv nil (ckrl2clos:ckrl-rules x))))
  (ifv (eqv (ckrl2clos:ckrl-rules x) nil)
     (format t " - ~a CANNOT fulfil this role because it contains no rules~%"
(file-namestring (value-of (ckrl2clos:ckrl-filename x)))))
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; A dish recipes role must contain instances of a task
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass dish-recipes-kb (role)
  ()
  )

(defmethod initialize-instance :after ((x dish-recipes-kb) &rest args)
  (let ((r-instances (ckrl2clos:ckrl-instances x))
        (r-rules (ckrl2clos:ckrl-rules x))
        )

    ;; The type is compared with "task" rather than 'task because of
    ;; the way that the translation to CKRL has been implemented.
    ;; It is easier to deal with strings than symbols because they do
    ;; not belong in a package.
    (assert! (somev #'(lambda(y) (equalv y "task"))
                    (mapcarv #'get-instance-type r-instances)))

    ;;; Assert that the KB must also contain a KB called schedule-genesis
    (assert! (memberv 'schedule-genesis r-rules))
    )
  )

;;; Returns the instance type of a CKRL instance after parsing by
;;; CKRL2CLOS
(defmethod get-instance-type ((i cons)) (secondv i))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This class describes a configuration
;;; A configuration consists of a problem solver, with some input roles,
;;; a problem solving goal, and some knowledge bases. One of the main tasks
;;; is to assign those knowledge bases to appropriate input-roles.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass configuration ()
  (
   (problem-solver :initarg :problem-solver :initform (make-variable))
   (input-roles :initarg :roles :initform (make-variable))
   ;; The directory containing the knowledge bases
   (kb-path :initarg :kb-path :initform *ckrl-dir*)
   (kbs :initarg :kbs :initform (make-variable))
   (plausible-output :initform (make-variable))
   (goal :initarg :goal :initform (make-variable))
   ;; This slot refers to the assignment of knowledge bases to roles
   ;; it is a set of pairs in which each kbs is labelled with a
   ;; set of plausible roles
   )
  )

(defmethod initialize-instance :after ((c configuration) &rest args)
  (let
      (kbs roles possible-roles)
    ;; The knowledge bases contained in the given directory are retrieved
    ;; and put into the kbs slot
    (setq kbs (directory (slot-value c 'kb-path) :wild t))

    ;; Note the roles with their possible knowledge bases
    (setf (slot-value c 'input-roles)
          (mapcar #'(lambda(x) (list x (a-member-ofv kbs)))
                  (slot-value (slot-value c 'problem-solver) 'input-roles)
                  )
          )

    (setq roles (mapcarv #'carv (slot-valuev c 'input-roles)))

    ;; Note the knowledge bases with their possible roles
```

```
    (setf (slot-value c 'kbs)
          (mapcar #'(lambda(x) (list x (a-member-ofv (cons nil roles)))) kbs)
          )
    )
  )

;;; This method sets the problem solver of a configuration
(defmethod set-problem-solver ((c configuration) (p problem-solver))
  (assert! (equalv (slot-valuev c 'problem-solver) p))
  )

;;; This method returns all the plausible configurations for the problem
;;; solver
(defmethod plausible-configurations ((c configuration))
  (all-values (solution (slot-value c 'input-roles) (static-ordering #'linear-
force)))
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; For modelling the designer
;;;
;;; This is a renaming, rather than an extension, of the dish class
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass plausible-design (knowledge::dish)
  ()
  )

(defmethod initialize-instance :after ((x plausible-design) &rest args)
   ;; Assert that the main feature of the dish is one of those listed
   (assert! (memberv (slot-valuev x 'main) *main*))
   ;; Assert that the carbohydrate is one of those listed
   (assert! (memberv (slot-valuev x 'carbohydrate) *carbo*))
   ;; Assert that vegetables is a list of length 2...
   (assert! (equalv (slot-valuev x 'vegetable) (make-listv 2)))
   ;; ...in which both members are each a vegetable
   (assert! (everyv #'(lambda(l) (memberv l *veg*)) (slot-valuev x 'vegeta-
ble)))
   ;; Assert that the vegetables are different
   (assert! (all-differentv (firstv (slot-valuev x 'vegetable))
                            (secondv (slot-valuev x 'vegetable))
                            )
            )
   ;; Assert that one of the sauces must be taken
   ;; Assert that the total cost is the sum of the costs of the parts
   (assert! (equalv (slot-value x 'cost)
              (+v
                (slot-valuev (slot-valuev x 'main) 'cost)
                (slot-valuev (slot-valuev x 'carbohydrate) 'cost)
                (slot-valuev (firstv (slot-valuev x 'vegetable)) 'cost)
                (slot-valuev (secondv (slot-valuev x 'vegetable)) 'cost)
                )
              )
     )
   ;; Assert that the points score is the sum of the points scores of the parts
   (assert! (equalv (slot-value x 'dietary-points)
              (+v
                (slot-valuev (slot-valuev x 'main) 'dietary-points)
                (slot-valuev (slot-valuev x 'carbohydrate) 'dietary-points)
                (slot-valuev (firstv (slot-valuev x 'vegetable)) 'dietary-
points)
                (slot-valuev (secondv (slot-valuev x 'vegetable)) 'dietary-
points)
                )
              )
            )
   )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; For modelling the scheduler
;;;
;;; A plausible schedule is an extension of a plausible design.
;;; The extension is such that plausible preparation times are added.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defclass plausible-schedule (plausible-design)
  (
   (preparation-time :initarg :preparation-time :initform (an-integer-abovev
0))
   )
  )

(defmethod initialize-instance :after ((x plausible-schedule) &rest args)
  ;; Assert the minimum preparation time
  (assert! (>=v (slot-value x 'preparation-time)
             (maxv
               (slot-valuev (slot-valuev x 'main) 'preparation-time)
               (slot-valuev (slot-valuev x 'carbohydrate) 'preparation-time)
               (slot-valuev (firstv (slot-valuev x 'vegetable)) 'preparation-
time)
               (slot-valuev (secondv (slot-valuev x 'vegetable)) 'prepara-
tion-time)
               )
              )
           )
  ;; Assert the maximum preparation time
   (assert! (<=v (slot-value x 'preparation-time)
              (+v
                (slot-valuev (slot-valuev x 'main) 'preparation-time)
                (slot-valuev (slot-valuev x 'carbohydrate) 'preparation-time)
                (slot-valuev (firstv (slot-valuev x 'vegetable)) 'prepara-
tion-time)
                (slot-valuev (secondv (slot-valuev x 'vegetable)) 'prepara-
tion-time)
                 )
                )
           )
   )


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Plausibility Testing
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; The meta-designer
;;; Uses object-based representation of roles and checks role satisfaction
;;; at object creation time.

(defun designer (&key (components "kbs/dish-components.ckrl")
                      (ontology "kbs/background.ckrl"))
  "Meta Problem Solver in domain of dish design; returns plausible dish"
  (let (dish-ontology component-instances ontology-concepts instance-types)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Role Satisfaction
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    ;;; The constraints on the roles are guaranteed by the object constructors
    (setq dish-ontology (make-instance 'dish-ontology :filename ontology))
    (setq component-instances
      (make-instance 'component-instances-kb :filename components)
      )


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Inter-role Consistency
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    ;;; Assert that every one of the instance types is for one of the known
    ;;; concepts
    (setq ontology-concepts (ckrl-concepts dish-ontology))
    (setq instances (ckrl-instances component-instances))
    (assert! (everyv #'(lambda(x) (memberv x ontology-concepts))
                     (mapcarv #'get-instance-type instances)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;; Generate Output
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

   ;; Extract partitions of the components
   ;; The following values are used for expressing the relationship between
   ;; the inputs and the outputs when creating the instance of the plausible
   ;; design
    (setq *main* (remove-if-not #'(lambda(x) (typep x 'knowledge::main)) *in-
stances*)
          *carbo* (remove-if-not #'(lambda(x) (typep x 'knowledge::carbohy-
drate)) *instances*)
         *veg* (remove-if-not #'(lambda(x) (typep x 'knowledge::vegetable))
*instances*))
    (make-instance 'plausible-design)
    )
  )


;;; The meta-scheduler
;;; Uses LISP-based roles
(defun scheduler (&key (components "kbs/dish-components.ckrl")
                       (recipes "kbs/recipes.ckrl")
                       (ontology "kbs/background.ckrl")
                       (resources "kbs/resources.ckrl"))
  "Meta Problem Solver in domain of dish design; returns plausible dish and
preparation times"
  (let (dish-ontology component-instances ontology-concepts instance-types
        dish-recipes uses resources-used known-resources
        precedes precedes-tasks known-tasks)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Role Satisfaction
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

   ;;; The constraints on the roles are guaranteed by the object constructors
    (setq dish-ontology (make-instance 'dish-ontology :filename ontology))
    (setq component-instances
      (make-instance 'component-instances-kb :filename components)
      )
    (setq dish-recipes (make-instance 'dish-recipes-kb :filename recipes))
    (setq dish-resources (make-instance 'role :filename resources))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Inter-role Consistency
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    ;;; Assert that every one of the instance types is for one of the known
    ;;; concepts
    (setq ontology-concepts (ckrl-concepts dish-ontology))
    (setq instances (ckrl-instances component-instances))
    (assert! (everyv #'(lambda(x) (memberv x ontology-concepts))
                     (mapcarv #'get-instance-type instances)))


    ;;; For inter-role consistency,
    ;;; - each uses relation may only use known resources,
    ;;; - each precedes relation may only refer to defined tasks, and
    ;;; - each subtask-of relation may only refer to other tasks defined
    ;;;   in the recipes knowledge base.

    ;;; - each uses relation may only use known resources,
    (setq uses (knowledge::match-facts '(knowledge::uses ? ? ?)))
    (setq resources-used (mapcar #'third (mapcar #'third uses)))
   (setq known-resources (mapcar #'(lambda(z) (find-symbol (string-upcase z)
:knowledge)) (mapcar #'first (ckrl-instances dish-resources))))
    (assert! (everyv #'(lambda(y) (memberv y known-resources)) resources-
used))

    ;;; - each precedes relation may only refer to defined tasks, and
    (setq precedes (knowledge::match-facts '(knowledge::precedes ? ?)))
    (setq precedes-tasks (append (mapcar #'third (mapcar #'second precedes))
                                 (mapcar #'third (mapcar #'third precedes))))
```

```
    (setq known-tasks (mapcar #'(lambda(x) (slot-value x 'knowledge::object-
name)) (remove-if-not #'(lambda(y) (typep y 'knowledge::task)) *instances*)))
    (assert! (everyv #'(lambda(y) (memberv y known-tasks)) precedes-tasks))

    ;;; - each subtask-of relation may only refer to other tasks defined
    ;;;   in the recipes knowledge base.
    (setq subtasks (knowledge::match-facts '(knowledge::subtask-of ? ?)))
    (setq subtask-tasks (append (mapcar #'third (mapcar #'second precedes))
                                (mapcar #'third (mapcar #'third precedes))))
    (assert! (everyv #'(lambda(y) (memberv y known-tasks)) subtask-tasks))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Generate Output
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

   ;; Extract partitions of the components
   ;; The following values are used for expressing the relationship between
   ;; the inputs and the outputs when creating the instance of the plausible
   ;; design

   ;; Extract partitions of the components
   (setq *main* (remove-if-not #'(lambda(x) (typep x 'knowledge::main)) *in-
stances*)
         *carbo* (remove-if-not #'(lambda(x) (typep x 'knowledge::carbohy-
drate)) *instances*)
         *veg* (remove-if-not #'(lambda(x) (typep x 'knowledge::vegetable))
*instances*))
    (make-instance 'plausible-schedule)
    )
  )
```

Appendix C: Monitoring Expensive Computations

The function `with-abort-check` was described in Chapter 6 (page 196) as one of the possible approaches to ensuring the computational attractiveness of a problem-solver meta-description. It uses the multi-processing capabilities of LISP to monitor the progress of the computation. More specifically, the function not only evaluates the given expression, it also spawns an independent "monitor thread" within LISP (see

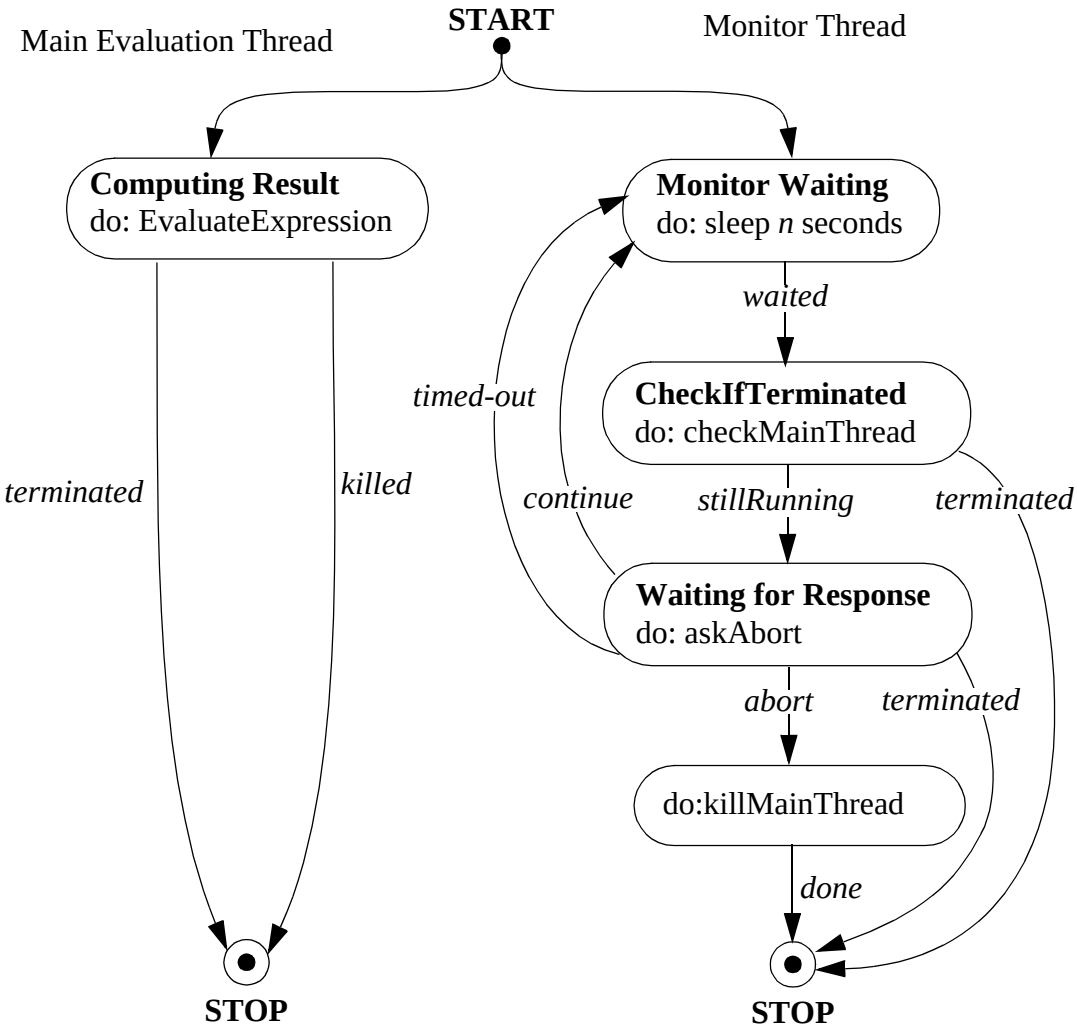


Figure 63: State Transition Diagram for `with-abort-check`

Figure 63). At first, this extra thread simply sleeps for a predetermined number of seconds. If the main computation has not terminated within that period, the user is asked whether the computation should continue. With the user's agreement the computation continues and the monitor-process again sleeps for a predetermined time interval before intervening again. This pattern is repeated until either the user aborts the computation, or the computation terminates. If the main thread of execution terminates, the monitor thread dies automatically. Furthermore, the regular request for user feedback is provided with a five second time-out. This means that the user is not obliged to answer the feedback request, because if no response is provided within those five seconds, the computation continues anyway.

The following is a code listing:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This set of functions implements with-abort-check in Allegro LISP.
;;; Uses Allegro LISP's "multiprocessing" (actually multithreading)
;;; abilities to execute a command which can take a long time to run, but
;;; periodically asks the user whether the computation should be aborted.
;;;
;;; By default, the process checks with the user every 60 seconds, so
;;; if a computation lasts less than 60 seconds, then no user requests
;;; would be generated; e.g.,
;;; > (with-abort-check (sleep 15))
;;; NIL
;;;
;;; However, the period can also be specified by supplying an argument:
;;; > (with-abort-check (sleep 15) :period 5)
;;; Abort? (y/n)n
;;; Abort? (y/n)n
;;; NIL
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; The top-level macro for with-abort-check
;;; The programmer supplies a command to execute (expression to evaluate)
;;; and an optional period after which the user is asked whether or not
;;; the computation should continue.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmacro with-abort-check (cmd &key (period 60))
  `(monitor-cmd (quote ,cmd) :period ,period)
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This function spawns two subprocesses (LISP threads) - the thread of
;;; main execution, and a monitor thread which periodically asks the user
;;; whether the main execution thread should continue.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun monitor-cmd (cmd &key (period 60))
  (declare (special *result*))
  (let (searchproc timeproc)
    (setq *result* '?)
    (setq searchproc (mp:process-run-function "Execution Thread" #'execute
cmd))
    (setq timeproc (mp:process-run-function "Monitor" #'monitor searchproc
period))
    ;; Waits until the result is bound
    (mp:process-wait "Search Wait"
```

```
                    #'(lambda() (not (mp:process-active-p searchproc)))
                    )
    *result*
    )
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This function runs as a thread which monitors the execution of another
;;; thread (LISP process). It waits a predefined number of seconds, then
;;; asks the user if (s)he wants to abort the monitored thread.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun monitor (timed-process seconds)
  (declare (special *result*))
  (loop
    (mp:process-sleep seconds)
    ;; If the process is still active
    (if (mp:process-active-p timed-process)
        ;; Ask if it should be continued
        (when (ask-abort timed-process)
          (mp:process-kill timed-process)
          (setq *result* 'aborted)
          (return)
          )
      ;; Otherwise quit this process
      (return)
      )
    )
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This function simply evaluates the expression it is given, and puts
;;; the result in the global variable *result*. It is the function named
;;; as the main execution thread from within monitor-cmd.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun execute (cmd)
  (declare (special *result*))
  (setq *result* (eval cmd))
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This function asks the user whether a given process should be aborted.
;;; The user is given 5 seconds to respond to the question. If no response
;;; is received in those 5 seconds then the computation would normally
;;; proceed. If, however, the computation terminates "naturally" within
;;; those 5 seconds, then a message to that effect is printed.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun ask-abort (process &aux cont)
  (format t "Abort? (y/n)")
  ;; Give the user 5 seconds to abort the process
  ;; If the request times out, return false and continue
  (mp:with-timeout (5
                    (if (mp:process-active-p process)
                        (format t "~%Continuing...~%")
                      (format t "~%[Computation completed.]~%")
                      )
                    nil)

                   (setq cont (read))
                   (if (or (equal cont 'y) (equal cont 'yes))
                       t
                     nil)
                   )
  )
```

Appendix D: The COCKATOO Application Programmer's Interface

D.1 Grammars

When dealing with non-trivial grammars, one must be able to manage the name space of the clauses such that clauses may be updated, and intentionally different clauses of the same name do not interfere with each other. COCKATOO solves this problem in a similar way to the *package* concept of Common LISP. That is, a grammar defines a "name space" within which only one clause of any given name may exist.

A grammar is introduced with the function `in-grammar`, supplied with a single argument name; e.g., `(in-grammar 'foo)`. If a grammar with the supplied name is not known, then a new grammar object is created and returned. Otherwise, the existing grammar is set to be the current name space, and the corresponding grammar object is returned as the value of the expression. Whenever `defclause` expressions are subsequently encountered by the LISP reader, those clauses are stored in the current grammar object. If a grammar is reloaded, the known definitions of clauses are overwritten by those from the loaded file.

In addition, the `:start-with` keyword argument sets a grammar's "top-level" clause. The method `acquire-grammar` can be used to acquire the values that satisfy the grammar's specification, starting from that top-level clause. COCKATOO Application Programmers' Interface

---

`defgrammar` [Macro]

Synopsis: (**`defgrammar`** *g* `&key` *(start-with clause)*)

Description: This macro defines the name space of a grammar (*c.f.*, the Common LISP macro `defpackage`). It is usual for `defgrammar` to be used once somewhere near the top of a grammar definition file. It is an error to define a clause without first defining at least one grammar.

Examples

```
(defgrammar breakfast
  :start-with 'choose-breakfast)
```

`acquire-grammar` [Method]

Synopsis: (**`acquire-grammar`** *g*)

Description: This method recursively expands the clauses of a grammar in a depth-first fashion, acquiring a value for each non-terminal symbol on its path. The argument *g* names the grammar whose top-level clause should be acquired. The method returns a cons structure (in effect a parse tree) of acquired values.

Examples

```
(acquire-grammar *current-grammar*)
```

`in-grammar` [Function]

Synopsis: (**`in-grammar`** *g*)

Description: This function states the current operative grammar. The supplied value *g* becomes the current grammar held in the variable `*grammar*` and any subsequent calls to `defclause` augment (or modify) the clause definitions maintained by that grammar. It is an error to attempt to switch to an undefined grammar but, if required, the unknown grammar can be created.

Examples

```
> (in-grammar ’breakfast)
#<GRAMMAR:BREAKFAST>

> (in-grammar 'english)
Error: The grammar ENGLISH is not known.
Restart actions (select using :continue):
 0: Create the grammar.
 1: Return to Top Level (an "abort" restart)
[1c] USER(59): :continue 0
#<GRAMMAR:ENGLISH>
>
```

`find-grammar` [Function]

Synopsis: (**`find-grammar`** *g*)

Description: This function searches for a grammar of the given (symbolic) name and, if found, returns it as the result of the search. If the grammar is not found then the function returns `nil`.

Examples

```
;;; The grammar 'breakfast is known
> (find-grammar 'breakfast)
#<GRAMMAR:BREAKFAST>

;;; The grammar 'baz is not known
> (find-grammar 'baz)
NIL
>
```

`*grammar*` [Variable]

Synopsis: **`*grammar*`**

Description: This variable holds the value of the current grammar in the same way that the Common LISP variable `*package*` holds the value of the current LISP package.

Examples

```
> *grammar*
#<GRAMMAR:BREAKFAST>
>
```

`cockatoo` [Function]

Synopsis: (**`cockatoo`**)

Description: A top-level function call for acquiring a grammar; currently defined as `(acquire-grammar *grammar*)`.

D.2 Clauses

`defclause` [Macro]

Synopsis: (**`defclause`** *name* `::=` *exp*) | (**`defclause`** *name* `->` *exp*)

Description: This macro defines a grammar clause by using combinations of the operators `seq`, `one-of`, `repeat+`, `repeat*` and `optional` as prefix (rather than infix) operators. A grammar must be known (i.e. `*grammar*` should hold an instance of a grammar object) before a clause is defined; otherwise an error is signalled. `defclause` allows the predominantly BNF form of clauses using '`->`' as the defining syntax, as well as the predominantly EBNF form using '`::=`'.

Examples

```
(defclause foo ::= (one-of 'yes 'no))

(defclause bar -> (one-of 'true 'false))

;;; This is what happens if no grammar has yet been defined
> (defclause no-grammar -> (seq 'foo))
Error: A grammar has not been defined
Restart actions (select using :continue):
 0: Define a default grammar
 1: Return to Top Level (an "abort" restart)
[1c] > :continue 0
Current grammar is now #<GRAMMAR:DEFAULT>
#<PRODUCTION:NO-GRAMMAR>
>
```

`seq` [Function]

Synopsis: (**seq** x_1 x_2 … x_n)

Description: This is the sequential composition operator. It returns the sequence $\langle x_1, x_2, \ldots, x_n \rangle$ where each x_i may be a terminal or non-terminal symbol. If x_i is a non-terminal symbol, then its exact value must be determined (by acquisition or otherwise) before the value of the sequence is returned. From a LISP programmer's point of view, `seq` is a synonym for the Common LISP function `list`.

Examples

```
> (seq 'foo 'bar)
(FOO BAR)
>
```

`one-of` [Function]

Synopsis: (**`one-of`** x_1 x_2 … x_n)

Description: Informally, this operator expresses an alternative; that is, any one of the symbols $x_1 \ldots x_n$ is an allowable value of the expression. Actually, it returns an expression which evaluates to a variable constrained to be one of the given alternatives. The additional level of evaluation is a necessary feature of the implementation and should be transparent to a COCKATOO user.

Examples

```
USER(5): (eval (one-of 'foo 'bar))
[1 nonnumber enumerated-domain:(FOO BAR)]
>
```

`repeat+` [Macro]

Synopsis: (**`repeat+`** *exp* `&key` *comment question*)

Description: This operator returns a repetition of one or more occurrences of the expression *exp*. It is equivalent to the EBNF syntax `{`*exp*`}+`, except that since COCKATOO repeatedly acquires values for the expression *exp*, `repeat+` also accepts keyword arguments to support the interaction with the user. As with a clause, a comment and/or a question can be supplied, which are displayed for each iteration of the `repeat+` cycle (see examples below).

Examples

```
> (setq r (repeat+ 'tea :question "Would you like a cuppa? "))
#<REPETITION:1>
> (acquire r)
Would you like a cuppa? y
Would you like a cuppa? n
(TEA TEA)
>
```

`repeat*` [Macro]

Synopsis: (**`repeat*`** *exp* `&key` *comment question*)

Description: This is very similar to the operator repeat+, described above, except that it returns a repetition of zero or more occurrences of the expression *exp*. It is equivalent to the EBNF syntax {*exp*}*, except that since COCKATOO repeatedly acquires values for the expression *exp*, `repeat*` also accepts keyword arguments to support the interaction with the user. As with a clause, a comment and/or a question can be supplied, which are displayed for each iteration of the `repeat*` cycle (see examples below).

Examples

```
> (setq r (repeat* 'tea :question "Would you like a cuppa? "))
#<REPETITION:2>
> (acquire r)
Would you like a cuppa? y
Would you like a cuppa? n
(TEA)
>
```

`optional` [Macro]

Synopsis: (**`optional`** *exp* `&key` *comment question*)

Description: This macro returns an optional expression. If the user affirms the option at acquisition time, then the expression *exp* is acquired and returned. Otherwise no values are returned. As with `repeat+` and `repeat*`, `optional` accepts the keyword arguments for generating a comment and/or a question at acquisition time.

Examples

```
> (setq o (optional 'drink :question "Would you like a drink? "))
#<OPTION:3>
> (acquire o)
Would you like a drink? n
> (acquire o)
Would you like a drink? y
DRINK
```

```
>
```

`acquire` [Method]

Synopsis: (**`acquire`** *exp*)

Description: This method applies to clauses, sequences, alternatives, repetitions, and options. It acquires an instantiation of the expression *exp*, usually by interacting with the user. The user supplies makes any decisions which are necessary, and supplies any values which are needed.

Examples

Given the clause definition

```
(defclause foobar -> (one-of 'foo 'bar))
```

a value can be acquired as follows:

```
> (acquire (find-clause 'foobar))
The possible values are:
  A.  FOO
  B.  BAR
Which value? : b
BAR
>
```

D.3 Constraining Clauses

`make-clause` [Function]

Synopsis: (**`make-clause`** *x* `&key` *name question comment postproc*)

Description: This function has been provided so that clauses can create other clauses. The mandatory argument *x* is a clausal expression using the keywords `seq`, `one-of`, `optional`, `repeat+` or `repeat*`. A clause is returned that, when acquired, returns a value satisfying the keyword specification.

Examples

```
> (setq a (make-clause (one-of 'red 'green)
                       :name 'colour
                       :question "Which colour?"))
#<CLAUSE:COLOUR>

> (acquire a)

Which colour?

The possible values are:
  A.  RED
  B.  GREEN
Which value? : b
GREEN
>
```

`find-clause` [Function]

Synopsis: (**`find-clause`** *x* `&optional` (*g* `*grammar*`))

Description: This function searches for a clause with the given (symbolic) name. If found, it returns the clause and the grammar containing the clause. When supplied, the grammar *g* is always searched first (*g* defaults to the value of `*grammar*`). If the clause is not found in *g* then the search is widened to the other known grammars. When the search is widened, the task is to find out whether a clause of the given name exists in any grammar (rather than searching for all known clauses of that name in all the known grammars). The search is therefore terminated as soon as a clause with the required name is found.

Examples

```
> (find-clause 'brekky)
#<CLAUSE:BREKKY>
#<GRAMMAR:BREAKFAST>

> (find-clause 'baz)
NIL
>
```

`acquired-valuev` [Method]

Synopsis: (**`acquired-valuev`** *clause*)

Description: This method returns a variable which is constrained to be the result of acquiring the value of the given clause.

Examples

```
(defclause dessert ::=
  (let (
        (a (make-clause (one-of 'chocolate-cake 'fruit-salad)))
        )

    (assert! (not-equalv (acquired-valuev a) 'chocolate-cake))

    a)
  )
```

The above-given clause creates a sub-clause in the local variable `a`, then asserts its acquired value not to be `'chocolate-cake`. Since this leaves `'fruit-salad` as the only choice, acquiring `dessert` automatically returns `'fruit-salad`.

```
> (acquire (find-clause 'dessert))
FRUIT-SALAD
```

D.4 File Naming Convention

During the development and subsequent use of COCKATOO, I adopted the convention of naming grammar source files with the suffix "`.gmr`". This makes them easily distinguishable from conventional LISP source files, which are typically tagged with one of the suffixes "`.cl`", "`.lisp`" or "`.lsp`".

One disadvantage is that when loading a grammar file named in this way, one would normally have to type in the *whole* of the file name, and could not omit the filename extension as with conventional LISP sources. This problem can be overcome in Franz Allegro Common LISP by redefining the `system::*load-search-list*` variable, so that `.gmr` becomes a recognised filename extension.

For example, after placing the following assignment in a LISP initialisation file, grammar source files can be loaded without the need for the filename extension.

```
(setq sys:*load-search-list*
  '(:first
    #.(make-pathname)
    #.(make-pathname :type "sfasl")
    #.(make-pathname :type "fasl")
    #.(make-pathname :type "cl")
    #.(make-pathname :type "lisp")
    #.(make-pathname :type "gmr")
    )
  )
```

viz.

```
> (load "dish")
; Loading C:\Allegro\Cockatoo\dish.gmr
Warning: Updating clause HEADER in grammar DISH
Warning: Updating clause SUBTASK-REL in grammar DISH
Warning: Updating clause TASKNAMES in grammar DISH
Warning: Updating clause MAIN-COMPONENT in grammar DISH
Warning: Updating clause CARBOHYDRATE in grammar DISH
Warning: Updating clause VEGETABLE in grammar DISH
T
>
```

Subject Index

A

ACKnowledge 87
a-consv 269
acquire 337
acquired-valuev 233, 339
acquire-grammar 330, 331
AC-1 51
AC-4 51
a-listv 269
all-differentv 142, 282
all-values 110
appendv 280
APT 80
AQUINAS 20, 21
Arc Consistency 50
arefv 125, 287
assert! 108
assumptions 91
a-stringv 269
a-symbolv 269
at-leastv 151, 153, 297
at-mostv 151, 153, 297
a-typed-varv 270

B

Backjumping 57
Backpropagation 31
Backtracking 48
Backus-Nauer Form (BNF) 205
Backward-chaining 11
bag-equalv 285
best-value 110
Binary Decision 197
Binding Propagation 113
BJ-FC 58
Boolean Constraint Propagation . . . 114
Bounds Propagation 115
Bull 63

C

Car Sequencing Problem 147
car sequencing problem 147
carv 120, 278
cdrv 129, 279
Certainty Factor 195
CHARME 63
CHEF 183
CHIP 62
CIGOL 80
CKRL 41, 79, 184
CLAIRE 69, 70
class-namev 291
class-ofv 291
classpv 290
CLIPS 34, 231
CLOS 207
CLP 62
CML 37, 157
CN2 80
cockatoo 333
Common LISP 102
Completeness 198
Computational Attractiveness 196
conditional constraints 126
Configuration 164
Consolidated Systems 62
conspv 268
Constraint 45
  Binary 45
  Programming 44
  Satisfaction Problem 44, 46
  Technology 44
Constraint Network 51
constraint primitives 108
Constraint Programming 60
Constraint-Augmented Grammar . . 211
constraint-fn 122, 151, 298
Consultant 84

consv 120, 277
Context-Free Grammar . . 99, 202, 204
context-free grammars 204
Context-Sensitive Grammar 204
context-sensitive grammars 204
COSYTEC 63
Cover-and-Differentiate 39
cross-breeding 27
Crossnumber Puzzle 142
crossover point 27
Cryptarithmetic 46

D

daemon 125
Dash 231
Data Mining 42
Decision Tree 23
DecisionPower 63
defclause 207, 333
defgrammar 331
degree of a node 96
DENDRAL 10
Designer 184
DFKI 72
DMP 80
Domain 10
domain 45
Domain Expert 12
domain theory 25
Duality 197
Dynamic Ordering 53
Dynamic Properties 199

E

EBG 25
ECLAIR 69, 71
ECLIPSE 63, 67
ECRC 62
EKAW 41
Elicitation 12
EMYCIN 12, 38
EPSRC . . . . . iv, 42
equalv 118
ETS 20
everyv 296
exactlyv 142, 151, 297
Expert Systems 10
Extended Backus-Nauer Form 206

F

Fail-First Strategy 54
find-clause 208, 233, 338
find-grammar 332
firstv 273
Fitness for Purpose 93, 156, 160
fitness function 27
Fitness-for-Purpose 201
Flexibility 197
Formalisation
Encoding 12
Formative Systems 61
formatv 125, 300
Forward Checking 54, 55
Generalised 56
Forward-chaining 11
fourthv 273
funcallinv 299
funcallv 119

G

Gantt Charts 193
Generalised Directive Models 87
Generalised Forward Checking 56, 114
Generate-and-Test 53, 167, 185
Genetic Algorithms 22, 27
goal-driven knowledge acquisition . . 77
Gödel 74
Grammar 202
Formal 202
Grammar Development Lifecycle . 228
Grammar-Driven Knowledge Elicitation
206
ground-level 166

H

Halting Problem 94
Heuristic Classification 39
Holy Grail 60

I

IBROW 41
IBROW3 91
ICL 63
IC-Parc 63
ID3 24
`ifv` 126, 271
ILOG Solver 65
`impliesv` 127, 270
Improvability 196
`in-grammar` 332
Inter-KB propagation 233
Inter-role Consistency 172
`intersectionv` 142, 283
Interview 18
  Focussed 18
  Structured 18
  Unstructured 18
Intra-KB propagation 233
Iterative Sampling 58

J

JCL 74
JULIA 183

K

KADS 36, 37
KARL 37, 157
KBG 80
KEW 87
KIF 41, 79
Knowledge
  Concensual 21
  Definition 35
  Level 35
  Level Model 36
  Management 41, 42
  Role 163
  Tacit 13
Knowledge Acquisition 10
  Bottleneck 12, 13, 22
  Definition 12
  Distributed 22
  Emergent 42
  Reactive 216
  Shell 229, 231
knowledge base 162
Knowledge Base Refinement 13, 14, 31, 32, ... 34
Knowledge Elicitation . . 13, 15, 16, 34
  Automated 13
  Contrived 16
  Uncontrived 16
Knowledge Engineer 12
Knowledge-Based Systems 10
KRUST 33

L

Laddering 19
LASH 80
LAURE 71
Learning 22, 23
  Analytical 22, 25
  Connectionist 22, 29
  Explanation-Based 25
  Inductive 22, 23
  Machine 23
  Rate of 30
  Supervised 24
  Unsupervised 24
`lengthv` 142, 276
Limited Discrepancy Search 58
LISP 100
`listpv` 268

M

Machine Learning 13, 14, 23, 34
maintaining arc consistency 57
`make-arrayv` 125, 286
`make-clause` 337
`make-equal` 272
`make-instancev` 287
`make-listv` 151, 281
`make-variable` 107, 110
MAKEY 80
Map Colouring 51
`mapcarv` 294
`maplistv` 295
Mastermind Game 140
Maximum Degree Ordering 54
`memberv` 119

Meta-Designer 187
meta-knowledge 33
meta–problem-solver 166
Metaproblem-Solvers 187
Meta-Scheduler 191
Minimal Bandwidth Ordering 53
Minimum Width Ordering 54
Minimum-Conflicts 59
MLT 77
MLT Consultant 84, 87
MOBAL 80
modus ponens 127
modus tollens 127
MUSKRAT 77, 86, 88, 156
Scope 87
MUSKRAT Advisor 89
mutation 28
MYCIN 10, 11, 12, 32, 33

N

negative instances 23
Neural Nets 29
NewID 80
Node-Consistency 49
Non-Terminal Symbol 203
`notanyv` 296
`noteveryv` 296
noticer 111
`nthv` 274

O

`one-of` 335
`one-value` 110
Ontology 40, 185
operationality criterion 25
`optional` 208, 336
Oz 69, 72

P

Palindrome 135
parsing problem 207
Path Consistency 52
PC-Pack 21
PECOS 65
Perceptron 29
Personal Construct Theory 19
Plausibility 94
Approximation 160
positive instances 23
Principle of Rationality 35
Problem Reduction 49
problem solver 162
Problem Solver Configuration 163
Problem Solving Method 38
problem-solving role 162
Production Clause 202
Progressive Systems 69
Propagation of Bounds 115
Propose-and-Revise 39, 59
Protégé 231
Protocol 17
ProtoKEW 21
PSM 39, 91

R

`reconcile` 293
Refinement 12
Regular-Expression Grammar 204
regular-expression grammars 204
`repeat*` 208, 336
`repeat+` 208, 335
Repertory Grid 16, 21
role 162
Role Configuration 163
ROSA 159
Rubik’s Cube 18
R1 11

S

SALT 60
Scheduler 184, 186
scope 45
SCREAMER 69, 70, 104
SCREAMER+ 202
Search 48
Constructive 52, 58
Non-Systematic 58
Repair-Driven 59
Systematic 58
`secondv` 273

Self Referential Quiz 145
Semantic Well-Foundedness 195
`seq` 334
sequential composition 205
`set-equalv` 282
`setq-domains` 273
SICLA 80
SICStus PROLOG 74
Siemens 63
`slot-exists-pv` 292
`slot-valuev` 288
`solution` 109
`somev` 121, 296
Sorting 18
Soundness 198
STALKER 34
Static Ordering 53
Steele's Constraint System 61
`stringpv` 268
Strong Methods 39
`subseqv` 275
`subsetpv` 285
surrogate constraints 151
suspension 68
`symbolpv` 268

T

Task 38
task instance 38
TDIDT 23
TEIRESIAS 33
Terminal Symbol 203
`thirdv` 273
training set 23
transfer approach 12
`typepv` 269

U

Unification 113
`unionv` 284
UPML 37

V

Variable Ordering Strategy . . . 53, 109
VDM 157

W

Weak Methods 39
WebOnto 22
`with-abort-check` 196, 327

X

XCON 11
XML 41, 79

Z

Z 157

Symbols

$(ML)^2$ 37
`*enumeration-limit*` 301
`*grammar*` 333
`*load-search-list*` 340