Pragmatics of the Knowledge Level

A. Th. (Guus) Schreiber

University of Amsterdam, Department of Social Science Informatics
Roetersstraat 15, NL-1018 WB Amsterdam, The Netherlands
Tel: +31 20 525 6792; Fax: +31 20 525 6896
E-mail: schreiber@swi.psy.uva.nl

Pragmatics of the Knowledge Level

(Gebruiksaspecten van het Kennisnivo)

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. P. W. M. de Meijer
in het openbaar te verdedigen in de Aula van de Universiteit
(Oude Lutherse Kerk, ingang Singel 411, hoek Spui),
op woensdag 28 oktober 1992 te 13.00 uur.

door

August Theodoor Schreiber

geboren te Heerlen

Promotor: Prof. dr. B. J. Wielinga
Faculteit Psychologie

Cover design: A. H. Schreiber

To Wilma, Niels & Judith

Contents

Preface

During the past years I have received help and support from many people. At Leiden University, Koos Mars taught me the first principles of science and raised my interest for the field of artificial intelligence. In Amsterdam I learned that research is in fact to a large extent team work. Many parts of this thesis have been influenced by work I have done together with colleagues at SWI. The work described in Ch. 3 is based on research of the whole KADS team anno 1986, in particular Joost Breuker and Bob Wielinga. I learned many things from the development of the StatCons system together with Paul de Greef and Jan Wielemaker. Part of the work on design described in Ch. 6 is based on research in the KADS project in which Bert Bredeweg, Massoud Davoodi, Peter Terpstra and Bob Wielinga participated. Ch. 2 arose from discussions about the knowledge level with Hans Akkermans and Bob Wielinga. Also, my work together with the colleagues of the REFLECT project has served as an important source of inspiration. The formal specification language used in Ch. 9 is the result of work of a group of people, in particular Hans Akkermans, John Balder, Frank van Harmelen and Bob Wielinga.

I am grateful to Ton de Jong and Jacobijn Sandberg, with whom I shared an office during five years, for the pleasant and stimulating working atmosphere. I have profited from many discussions with Hans Akkermans, Frank van Harmelen, Werner Karbach, Marc Linster and Angi Voß. My supervisor Bob Wielinga has coached me all along the way and his prying questions (and answers) provided the basis for this thesis. I also like to thank Anjo Anjewierden, Manfred Aben, Robert de Hoog and Jan Treur for their comments on earlier versions. of this thesis. All figures in this book are created with the PCEDRAW program developed by Jan Wielemaker.

The research reported in this thesis has been supported by a number of projects partially funded by the ESPRIT Programme of the Commission of the European Communities, notably projects P1098, P3178 and P5248. The partners in the P1098 (KADS) project were STC Technology Ltd. (UK), SD-scicon plc. (UK), Polytechnic of the South Bank, (UK), Touche Ross MC (UK), SCS GmbH (Germany), NTE NeuTech (Germany), Cap Gemini Innovation (France), and the University of Amsterdam (The Netherlands). The partners in the P3178 (REFLECT) project were the University of Amsterdam (The Netherlands), the German National Research Institute for Computer-Science GMD (Germany), the Netherlands Energy Research Foundation ECN (The Netherlands), and BSR-Consulting (Germany). The partners in the P5248 (KADS-II) project are Cap Gemini Innovation (France), Cap Gemini Logic (Sweden), Netherlands Energy Research Foundation ECN (The Netherlands), ENTEL SA (Spain), IBM France (France), Lloyd's Register (UK), Swedish Institute of Computer Science (Sweden), Siemens AG (Germany), Touche Ross MC (UK), University of Amsterdam (The Netherlands) and Free University of Brussels (Belgium). This thesis reflects the opinions of the author and not necessarily those of the consortia.

Chapter 1

Introduction

1.1 Symbolic AI and the Field of Knowledge Engineering

In this age of technological advances, the question whether one can build a "thinking machine" has become a popular topic, both in scientific circles and in social conversations. Leibniz was probably one of the first to come up with the idea of mechanising human thought through a calculus [Russell, 1961; Mars, 1987]. The goal of his concept of a *Characteristica Universalis* was to define a generalised version of mathematics which would enable the resolvement of philosophical debates through computations. More than two hundred years later, Turing defined the concept of a universal, programmable machine, and the "computer" (at least in theory) was born. When, after the second world-war, the first programmable computers became available, researchers started to work on computer programs that could do something "intelligent", e.g. translate sentences from one language into another. This research field has become known as *artificial intelligence* (AI).

A critical assumption behind much AI research is the hypothesis that it is possible to simulate intelligent behaviour through formal manipulations of symbols in a computer program. Smith has summarised this assumption elegantly in his *knowledge-representation hypothesis* [Smith, 1985; p. 33]:

> "Any mechanically embodied intelligent process will be comprised of structural ingredients that a) we as external observers naturally take to represent a propositional account of the knowledge that the overall process exhibits, and b) independent of such external semantical attribution, play a formal but causal and essential role in engendering the behaviour that manifests that knowledge."

This hypothesis, especially in its strong form ("*all* intelligent behaviour can be simulated"), has given rise to many debates within and outside the field of AI. We agree with Smith that, although we are not even in a position to commit ourselves fully to the weak version of the knowledge representation hypothesis, "it deserves our attention".

One of the areas in AI which builds heavily on the knowledge representation hypothesis is the field of knowledge-based systems. A knowledge-based system (KBS) is a system that is capable of carrying out problem-solving tasks, such as diagnosing diseases or configuring

device components.[1]. A KBS employs a symbolic representation of domain-specific knowledge in carrying out its task (hence the name *knowledge-based*). Knowledge-based systems were first developed in the sixties as a reaction to the so-called "general-purpose" or "weak" problem solving programs such as GPS [Newell & Simon, 1963]. Weak systems employed one general method such as means-end analysis for solving problems. Such general methods were however considered by many as inadequate for solving non-toy problems.[2] In contrast, a KBS was said to rely on large amounts of domain-specific knowledge and domain-specific strategies, which would enable it to achieve problem-solving capability. Given the focus in current KBS research on explicating the general problem solving strategy behind a system, the truth probably lies somewhere in the middle.

The first generation knowledge-based systems employed one relatively simple *inference engine* working on a knowledge base in a particular representational format, usually production rules. [Clancey, 1983] showed in his analysis of the prototypical system of this generation, MYCIN, that such a knowledge base hides various important properties of the reasoning process and of the structure of the knowledge in the application domain. Certain rules, or parts of rules, fulfill particular roles in the reasoning process which remain implicit in such a KBS organisation. This implicitness of underlying structures impairs the acquisition and refinement of knowledge for the KBS as well as the reuse of the system, its explanatory power and the assessment of its relation with other systems.

It is fair to say, that this problem was not specific for the field of *knowledge engineering* (KE, the more or less standard term for the process of KBS development). Similar problems were being identified in the broader area of knowledge representation. The clearest evidence of this was brought forward by Brachman and Smith through the results of their SIGART questionnaire [Brachman & Smith, 1980]. The aim of this questionnaire was to get data on various knowledge representation approaches in order to perform a comparative study. About the results of their analysis of the huge amount of data received, they remark:

> "Perhaps more than anything else, it has emerged as a testament to an astounding range and variety of opinions held by many different people in many different places." [Brachman & Smith, 1980; p. 1]

Everyone seemed to be speaking a different language: a true Babel.

In response to this confusion, Newell coined, in his presidential address to AAAI-80, the 'knowledge-level hypothesis". The key point underlying his argument was that the confusion arose because AI research was too much focused on detailed representational issues. What was missing was a description of the rationality behind the use of AI techniques. He pleaded for a shift of emphasis in AI research from the "how" questions to the "why" questions. The knowledge-level was his proposal for realising a description of an AI system in terms of its rational behaviour: why does the system (the "agent") perform this "action", independent of its symbolic representation in rules, frames or logic (the "symbol level").

[1] The term "expert systems" is also sometimes used to denote such systems, as problem-solving tasks are often carried out by experts in particular field, e.g. a doctor, an engineer.

[2] The recent successes of chess systems has made clear that at least for one type of application of weak methods this objection does not hold. The statement of Boden that "there is no prospect of a chess master being beaten by a program in the near future" [Boden, 1977; p. 353] has certainly turned out to be incorrect.

During the eighties, this idea of introducing a knowledge-level description was taken on in knowledge engineering research to solve the problems mentioned earlier. The purpose of a knowledge-level model of a KBS is to make the organisation of knowledge in the system explicit. It should provide an implementation-independent description of the role that various knowledge elements play during the problem-solving process of the system. A knowledge-level model should be able to explain the rationale behind the way in which the system carries out a task in a vocabulary understandable for humans. This makes such a model an important vehicle for communicating about the system both during development and during system execution.

The central topic of this thesis is to study how Newell's idea of a knowledge-level can be put into use in a principled way to support knowledge engineering.

1.2 Context and Theme of this Thesis

The research described in this thesis was carried out in the context of the development of the KADS[3] approach to KBS development. KADS has been and is being developed in a series of ESPRIT projects. The major actors in the development of the original ideas behind KADS were Bob Wielinga and Joost Breuker [Wielinga & Breuker, 1984; Breuker & Wielinga, 1984; Wielinga & Breuker, 1986].

Fundamental to the KADS approach is the use of "models of expertise" to analyse and specify the required problem-solving behaviour in a KBS application domain. KADS models of expertise have a cognitive flavour in the sense that they are aimed at human interpreters. The vocabulary in which they are expressed is that of experts and/or potential users of the system. KADS models of expertise are not cognitive models in the true sense of the word: their purpose lies in supporting a structured engineering process of knowledge-based systems.

The early KADS-related work in the first half of the eighties has developed into a rather broad field of research issues concerning a methodology for KBS development, including topics such as organisational embedding, life-cycle models, user interaction, formal specification and executable languages. The core of the approach however still has remained to be the nature and role of conceptual models of expertise in the knowledge engineering process. Although this was not an articulate assumption behind the KADS models of expertise as presented in [Wielinga & Breuker, 1986], these models can be seen as a *reification of the knowledge-level hypothesis for practical use in knowledge engineering.* Their aim is similar: providing an implementation-independent specification of the required problem-solving behaviour in the target KBS, which is intelligible for humans.

The *general theme* of this thesis is a *theoretical and practical investigation of* KADS *models of expertise as knowledge-level descriptions.* We study the nature of the relation between KADS models and Newell's knowledge hypothesis. We investigate a number of important questions, notably:

- What is an adequate language for describing the structure of domain-specific knowledge?

[3]The interpretation of this acronym has evolved over the years: from "Knowledge Acquisition Documentation System" via "Knowledge Acquisition Documentation Structuring" to "Knowledge Analysis and Design Structuring".

- What are principles underlying the model construction process?
- How should knowledge-level descriptions be transformed into symbol-level descriptions?

We describe a sample application of KADS. We also compare the KADS approach with two leading software engineering approaches and describe the use of KADS in a comparative study between two knowledge-level models developed outside the scope of KADS.

1.3 Structure of this Thesis

The body of this thesis consists of eight chapters. Each chapter represents a self-contained piece of work. Some chapters constitute papers that have been published elsewhere. Every chapter addresses particular research topics within the general theme of this thesis. This section gives an overview of these research topics.

KADS and the knowledge-level hypothesis Ch. 2 contains an investigation of a number of general issues related to Newell's knowledge level. This paper was triggered by Sticklen's criticism of the knowledge-level hypothesis [Sticklen, 1989]. We relate KADS models of expertise to Newell's original hypothesis and to other interpretations of the knowledge-level. We discuss criticisms such as the predictive power of knowledge-level models and the inability to represent control. In the process, we try to explicate a number of assumptions behind KADS such as the distinction between analysis and design and the computational adequacy of models of expertise.

Fundamentals of KADS models of expertise The purpose of Ch. 3 is to give a comprehensive and consistent description of the fundamentals of KADS. Over many years, people have complained about the lack of such a description. We present these fundamentals through three cornerstones underlying the KADS approach: (i) the introduction of various models as a means of coping with the complexity of the knowledge engineering process, (ii) the KADS framework for modelling the required expertise, and (ii) the reusability of generic model components as templates supporting top-down knowledge acquisition. We also relate KADS to other knowledge engineering approaches.

Of course, much of the material discussed in this chapter is the product of research of a group of people, notably the co-authors of this chapter, Wielinga and Breuker. However, it is probably fair to say that one of the reasons that an overview article such as this had not been written earlier was that some issues were not worked out in sufficient detail. In this chapter we describe:

- The introduction of the notion of a domain schema as a structural description of the domain-specific knowledge.
- An exploration of one of the key parts of the modelling framework: the relation between domain and inference knowledge.
- A refinement of the formulation of task knowledge.
- A (brief) overview of knowledge modelling activities.
- The description of a non-toy "running example".

The paper from which this chapter is derived was the basis for the special issue on KADS of the *Knowledge Acquisition Journal* [Schreiber, 1992].

Models of expertise: related research topics Ch. 4, Ch. 5 and Ch. 6 address three research topics that have proven to be of vital importance when using models of expertise in knowledge engineering:

Domain-knowledge modelling Traditionally, KADS was very functionally oriented: the emphasis was on notions like inference structures, which specify the basic functions (from a knowledge-level point of view) of the target system. The descriptive vocabulary for the domain-specific data manipulated by inferences consisted of "concepts and relations". The goal of Ch. 4 is to develop a more expressive data modelling framework that meets the specific requirements posed by knowledge engineering. The idea of the proposed language is to provide the knowledge engineer with a notation that allows generalisations over various knowledge representation techniques. This is in line with the idea of a knowledge-level perspective, where one does not want to commit oneself to a particular, "symbol-level", representation. The use of the proposed data modelling language, which is partly based on ideas proposed in semantic database modelling research, is illustrated through its application to a sample domain.

Model construction An often-heard criticism of the KADS approach is that it only provides a descriptive framework. Apart from the set of interpretation models, there is little support for the actual model-construction process. In Ch. 5 we discuss the construction of inference structures: a crucial ingredient of KADS models of expertise. We show how an initial inference structure can be gradually refined into an inference structure that meets the requirements of the application domain at hand. Model construction can be supported by generic model components of a smaller grain size than interpretation models. We illustrate the approach with an example showing how a KADS version of Clancey's heuristic classification model can be constructed. We also compare this inference structure with Clancey's latest observations about heuristic classification [Clancey, 1992].

In addition, we discuss a number of ambiguities in the graphical representation of inference structures and propose some extensions to cope with these problems.

From knowledge-level to symbol-level In Ch. 6 the *operationalisation problem* is discussed: how does one transform a knowledge-level description into a symbol-level description that can be implemented on a machine? We discuss the various types of decisions that have to be made in this design and implementation process. The notion of *structure-preserving design* is discussed as the leading principle that should underly this process. A skeletal architecture for "KADS" systems is presented, which supports a structure-preserving design. We also discuss various existing support environments for operationalising knowledge-level models and present an example environment that we developed.

A sample application Ch. 7 contains a sample application of KADS in a domain of allocating offices to employees. This application was built in the context of the Sisyphus'91 project "Models of Problem Solving". This project was initiated at the European Knowledge Acquisition Workshop 1990. The aim of the enterprise was to apply various

knowledge modelling approaches to the same application domain in order to get a common understanding of these approaches. This was felt to be a necessary first step in arriving at generally-shared theories about the nature an role of knowledge-level models in knowledge engineering.

KADS and conventional software engineering A question that frequently arises, especially in discussions with people from the software-engineering community, is how KADS relates to developments in software engineering. The aim of Ch. 8 is to compare the KADS approach with two leading software-engineering methodologies: Modern Structured Analysis [Yourdon, 1989b] and Object-Oriented Analysis [Rumbaugh *et al.*, 1991; Coad & Yourdon, 1991]. We discuss both similarities and differences with respect to three perspectives from which a system can be viewed: the *data* perspective, the *functional* perspective and the *control* perspective. A major common theme that arises, also for future research, is the issue of reusability.

The future – a first comparative study Currently, all knowledge-level models are described either in an informal way or through the systems that implement these models. This type of description often leads to ambiguities and/or misunderstandings. In Ch. 9 an attempt is made to describe the cover-and-differentiate method for diagnosis in a more formal way and to compare this method to heuristic classification. We identify considerable differences between the two methods, although in the original literature cover-and-differentiate was said to be a specialised form of heuristic classification.

We are not claiming that the model presented is the only correct one. However, the account given in this chapter can be a starting point for a precise definition of what methods like cover-and-differentiate actually do. We think this type of comparison can be a prelude for much further work which should ultimately lead to a shared library of generic models and model components.

In the last chapter we summarise the conclusions of the research described in this thesis. We also discuss open questions with respect to KADS as well as in the broader context of knowledge enginnering research. Some possible future developments are briefly addressed.

Chapter 2

On Problems with the Knowledge-Level Perspective

In this chapter some points of criticism on Newell's Knowledge-level Hypothesis are investigated. Among those are: the inability to represent control, the potential computational inadequacy, the lack of predictive power and the non-operational character (the problem of 'how to build it'). We discuss Sticklen's Knowledge-level Architecture Hypothesis in which he tries to overcome these problems. On the basis of general arguments as well as specific insights from our KADS knowledge level modelling approach we reject the points of criticism. We also argue that the extension Sticklen proposes is not necessary and partly also unwanted.

This chapter was first presented as a paper at the Banff'90 Knowledge Acquisition Workshop and later invited for presentation at the AISB'91 conference on Simulation of Behaviour. It is co-authored by Hans Akkermans and Bob Wielinga. Reference: G. Schreiber, H. Akkermans, and B. Wielinga. On problems with the knowledge-level perspective. In L. Steels and B. Smith, editors, *AISB91: Artificial Intelligence and Simulation of Behaviour*, pages 208–221, London, 1991. Springer Verlag. Also in: *Proceedings Banff'90 Knowledge Acquisition Workshop*, J.H. Boose and B.R. Gaines (editors), SRDG Publications, University of Calgary, pages 30-1 – 30-14.

2.1 Introduction

The introduction of the *knowledge-level hypothesis* by Newell has attracted considerable attention and stimulated new lines of research. It claims that there exists a distinct computer systems level that lies immediately above the symbol or program level. This knowledge level characterises the behaviour of problem solving agents in terms of their goals and actions, with knowledge serving as the medium, using a simple principle of rationality saying that an agent will carry out a certain action if it has knowledge that one of its goals can be achieved by that action [Newell, 1982].

It was Clancey who first showed the importance of this idea for the theory of knowledge-based reasoning [Clancey, 1983; Clancey, 1985b]. Since then, various authors have elaborated on the viewpoint that the knowledge level is the right level of abstraction for knowledge acquisition and engineering [Wielinga *et al.*, 1989; Bylander & Chandrasekaran, 1988; McDermott, 1988; Alexander *et al.*, 1988; Musen *et al.*, 1987; Steels, 1990]. This research has focused on the conceptual, implementation-independent aspects of knowledge and,

on this basis, has contributed to generic models of problem solving and to more solid methodologies for KBS development.

Evidently, the knowledge-level hypothesis has been a very fruitful one. On the other hand, it has also attracted strong criticism. This is well exemplified in a recent paper by Sticklen and the associated commentaries [Sticklen, 1989]. In brief, from discussions like these we single out as basic problems related to the knowledge-level hypothesis:

1. *Computational inadequacy*: Due to the high level conceptual bias of knowledge-level models, a real danger is their potential computational inadequacy.

2. *Non-operational character*: knowledge-level models do describe knowledge that is necessary for problem solving, but give no clue as to how to build computational systems embodying and exploiting that knowledge.

3. *Inability to represent control*: knowledge-level models capture knowledge used in problem solving actions, but they do not provide ways to express the control of problem solving.

4. *Lack of predictive power*: knowledge-level models are useful to explain —in retrospect the behaviour of certain problem solving agents (such as AI programs), but are unable to generate empirically testable predictions about that behaviour.

Interestingly, most of this criticism on the knowledge-level hypothesis appears to be inspired by what is seen by some as its greatest advantage: moving away from implementational issues in favour of the conceptual aspects of knowledge. In the present work, which is basically a *position paper*, the knowledge-level hypothesis and its problems will be investigated in some detail. On the basis of general arguments as well as specific insights from our KADS knowledge-level approach to KBS development [Wielinga *et al.*, 1989], we will consider —and reject— the points of criticism raised above.

2.2 The Knowledge-level Hypothesis Debate

2.2.1 The knowledge-level hypothesis The "knowledge-level hypothesis" (KLH) was put forward by Newell in his presidential address to AAAI-80 [Newell, 1982]. Newell discusses in this address the notions of *knowledge* and *representation* which in his view are central to AI. Newell signals that most of the work in AI is centred on representation. Representation refers here to the actual data structures and processes in an (AI) program. He suggests that the confusions in AI research on representation may (partly) be due to the limited attention the research community has given to the study of the nature of knowledge. The Knowledge-level Hypothesis is aimed at providing a platform for studying knowledge independent of its representation in a programming language. Newell phrases the KLH as follows:

> "There exists a distinct computer systems level, lying immediately above the symbol level, which is characterised by knowledge as the medium and the principle of rationality as the law of behaviour."

Representations (data structures and processes) are part of the the symbol level, whereas knowledge is the prime ingredient (the medium) at the knowledge level. Newell sketches the structure of the knowledge level as an agent that has a physical body (consisting of a set of actions), a body of knowledge, and a set of goals (goals are bodies of knowledge about the state of the environment). The principle of rationality that governs the behaviour of the agent is formulated as follows:

> If an agent has knowledge that one of its actions will lead to one of its goals, then the agent will select this action.

The central issue now becomes: what is the nature of the knowledge the agent has? Newell characterises knowledge as a"competence-like notion, being a potential for generating action" and as "entirely *functionally* in terms of what it does, not *structurally* in terms of physical objects". A representation is defined as a "symbol system that encodes a body of knowledge".

As an example of the usefulness of a distinction between knowledge level and symbol level, Newell points to the work of Schank on conceptual dependency structures [Schank, 1975]. He argues that the main contribution of this work is at the knowledge level – namely by providing a quite general way of describing knowledge of the world. Although AI researchers felt that this work was incomplete without an implementation, the actual program added little to the theoretical work: it was just a large AI program with the usual ad hoc constructions.

2.2.2 Knowledge-level modelling in knowledge acquisition One of the areas where the knowledge-level hypothesis has received considerable attention is the field of knowledge acquisition. Experience with the first generation of knowledge-based systems (MYCIN and its derivatives) showed that the *transfer approach* to knowledge acquisition was simply inadequate. In the transfer approach the knowledge engineer tries to extract knowledge from a domain expert in the form of the representation in the system (e.g. as production rules). The problems with this approach are manifold: the mapping from elicited expertise-data onto the required representation is difficult and often not possible; systems with a large knowledge base become difficult to maintain; explanation facilities are poor; etc. The main reason for this is that the gap between the observed problem solving behaviour and the target application is just too wide. What is needed is an intermediate description of the expertise in a task domain at a more abstract level. The knowledge level provides precisely this intermediate level of description. The introduction of this intermediate model implies a different approach to knowledge acquisition. In contrast to the transfer approach, expert data are not transferred directly into machine symbols, but serve as input for a modelling process.

Broadly speaking, two approaches can be identified to the use of the idea of a knowledge-level description in knowledge acquisition, In the first approach the starting point is an implementation of (parts of) a problem solver. Here, knowledge-level notions are introduced by providing abstract, implementation-free, descriptions of the knowledge elements required by the problem solver. Examples of this approach are the Generic Task approach [Bylander & Chandrasekaran, 1988] and the work of McDermott et al. on MOLE, SALT and other systems [Marcus, 1988]. In the second approach, exemplified in the KADS

methodology [Wielinga & Breuker, 1986], the knowledge-level descriptions are part of a conceptual model of a task domain. The conceptual model serves as a specification of the (knowledge) requirements for a particular knowledge-based application. The conceptual model is not directly linked with the actual implementation.

2.2.3 Criticism of the knowledge-level hypothesis The introduction of the KLH has led to criticism both of the KLH itself and also of the application of knowledge-level descriptions in the development of knowledge-based systems. In a recent article, Sticklen [Sticklen, 1989] phrases these points of criticism in the form of an extension of the Knowledge-level Hypothesis: the "Knowledge-level Architecture Hypothesis". We will go through his argument in some detail, as his discussion is in a sense typical for critics of the KLH.

First, Sticklen acknowledges that the notion of a knowledge-level description is a useful one. He points to the work of Clancey on Heuristic Classification [Clancey, 1985b] as a prototypical example of a knowledge-level description of a problem solving agent (process). In his article, Sticklen apparently views the "horse shoe" inference structure of heuristic classification as a complete knowledge-level description. He then concludes that this type of description is unable to yield verifiable predictions of the problem solving behaviour of an agent and that thus the knowledge-level hypothesis is incomplete. This conclusion is based on the assumption that a scientific theory has two necessary components: (i) the theory must account for known phenomena and (ii) the theory must make (verifiable) predictions about phenomena that will be observed in the future. He argues that the lack of predictive power of knowledge-level descriptions is due the fact that there is no way in Newell's knowledge level to specify problem solving control. In his Knowledge-level Architecture Hypothesis he proposes to extend the KLH with the possibility of decomposing an agent (task) into sub-agents (sub-tasks) and allow specification of ordering among sub-agents. This extension seems harmless enough. Newell himself is not very clear whether decomposition of agents is allowed. A requirement for the decomposition is that the resulting sub-agents are 'knowledgeable', i.e. that they can be described in knowledge-level terms. Sticklen is however not very clear on how this can be ensured.

Given a decomposition of an agent into sub-agents, Sticklen defines the corresponding knowledge-level architecture through two ingredients: (i) a specification of the communication paths between sub-agents, and (ii) the specification of the message protocols for inter-agent communication.

Sticklen claims that a knowledge-level architecture description provides a

> "*blueprint* for how to build a problem solver that may be used as a simulator."[1]

The simulator provides the required predictive part of the theory, The role of the simulator can be compared with the role of numeric simulations in physics.

In his article, Sticklen tries to cope with some –highly interrelated– points of criticism of the knowledge-level approach,. He mentions two points explicitly: the *inability to represent control* and the *lack of predictive power*. Two other points of criticism are:

- *Computational inadequacy* When a knowledge-level description is transformed into

[1] Italicisation by the authors

a symbol level description, what kind of guarantees does one have that the resulting system is computationally adequate?

- *Non-operational character* A knowledge-level model gives no clues about how to build an operational system, i.e. it does not solve the 'design problem'.

These last two points are also covered by the knowledge-level architecture hypothesis through the blueprint for building a simulator (i.e. a knowledge-based application).

In the rest of this chapter we discuss the various points of criticism. We will argue that the extension that Sticklen proposes is not necessary and partly also unwanted.

2.3 Representing Control in Knowledge-level Models

Sticklen's Knowledge-level Architecture Hypothesis encompasses an extension of the knowledge level with respect to the description of problem solving control. In our view knowledge about task (agent) decompositions and dependencies is indeed just another type of knowledge with its own specific characteristics that should —and can– be described in a knowledge-level model. The description in [Clancey, 1985a] of the diagnostic strategy developed for NEOMYCIN is a good example of what we would call a knowledge-level description of problem solving control. Clancey calls this a "competence model" of diagnostic strategy, which already indicates that the description has the knowledge-level flavour Newell is aiming for in his KLH.

In our opinion, the question whether this type of (competence-like) control knowledge is or is not present in the Knowledge-level Hypothesis, as originally stated by Newell, is not very important or interesting. From our experience it is clear that the above-mentioned type of control knowledge is a necessary and important ingredient of a practical knowledge-level theory of problem solving. In addition to the task decompositions and inter-dependencies (the task knowledge in the KADS model, see Ch. 3) we would also add meta-knowledge about a problem solving agent as a separate kind of control knowledge (somewhat confusingly termed "strategic knowledge" in KADS). This strategic or tactical knowledge is an important ingredient for building more flexible knowledge-based systems.

The problem of ensuring that the decomposition results in 'knowledgeable' sub-agents can be handled by providing a knowledge-level typology of canonical inferences (the lowest level of sub-agents), thereby ensuring that such agents can indeed be described in knowledge level terms. Examples of canonical inferences are the three steps in the Heuristic Classification inference structure (abstraction, association and refinement) [Clancey, 1985a] and the set of knowledge sources defined in KADS (see Table 3.2). Clearly, much work still needs to be done in this area to arrive at a coherent and more or less complete typology.

What worries us in the Knowledge-level Architecture idea is that it should provide a blueprint for building a simulator. Whereas we think that it is very well possible to define structured ways of building a knowledge-based system from a knowledge-level specification (see Sec. 2.6), we do not believe, that a knowledge-level model should –and can– contain all information necessary for building the implementation. In fact, by nature it should not! In the process of implementing a system it will always be necessary to add specific information. The design process is constrained by the knowledge level specification, but

a large number of design decisions concern issues that are not relevant at the knowledge level.

We think that the confusion arises from the way in which the word "control" is used. There is a difference between what we would call respectively *knowledge-level control* and *symbol level control*. The description of the diagnostic strategy of NEOMYCIN in [Clancey, 1985a] is a clear example of knowledge-level control. This type of knowledge concerns task decompositions of and orderings between knowledgeable agents and possibly also meta-knowledge about agents. Symbol level control is concerned with the control issues that arise when a particular representation or AI technique is selected to realise a problem solving agent. A similar distinction between these two types of control is made by Gruber [Gruber, 1989; p. 5].

Although symbol level control is not an issue at the knowledge level, it does pose several problems that have to have solved in the implementation of a particular application. One only has to look at the vast amount of "symbol level" AI research to conclude that these problems are by no means trivial. Thus, specification of knowledge-level control does *not* provide a blueprint for building a simulator.

2.4 Epistemological and Computational Adequacy

Knowledge-level models give a high level description of the knowledge as it is utilised in problem-solving reasoning. Although terminology is different, a common view appears to be emerging in the literature based on the idea that the knowledge level is constituted by different types and components of knowledge, and that these forms of knowledge play different roles in the reasoning process and have inherently different structuring principles. In addition, the knowledge-level approach attempts to demonstrate that many of these knowledge types and components have a generic character, *i.e.*, they are applicable to a broad class of tasks and/or domains. In this sense, generic knowledge components can be viewed as intermediate in the continuum from weak to strong methods.

The advantage of knowledge-level models lies in yielding a high level and intuitive *explanation of reasoning behaviour*. Moreover, generic models at the knowledge level are important vehicles for knowledge acquisition, because they provide the knowledge engineer with *reusable interpretation 'templates'* that guide the analysis and organisation of elicitation data.

As a consequence of the high-level conceptual nature, a major objection to the use of knowledge-level models in the KBS development process is their potential computational inadequacy. Since knowledge-level models do not specify the operational control regime in full detail they are apt to potential combinatorial explosive behaviour. The generic task approach by Chandrasekaran *et al.* takes the view that the knowledge-level description and the operational problem-solving method as employed in the computational system cannot be separated. In contrast, the choice of computational techniques to realise a certain knowledge-level function is seen in KADS as part of the design activity in knowledge engineering (further discussed in Sec. 2.6). An important criticism of the KADS-type of conceptual models is therefore that —although they are very useful from a practical epistemological viewpoint— they do not guarantee computational tractability.

2.4.1 The knowledge level: role limitations In this very broad and general way of speaking, the criticism above is correct: knowledge-level models do not make statements about computational adequacy as such. However, we claim that the *structure* of knowledge-level models as outlined previously provides important safeguards against the computational inadequacy. The combinatorial complexity at the computational level is caused in our view by the unrestricted applicability of and access to knowledge as present in the knowledge base. The underlying principle of knowledge-level modelling is *imposing structure through knowledge differentiation*. This is achieved by distinguishing within the body of knowledge involved different types and components that play specialised roles in the totality of the problem-solving process.

An example is the cover-and-differentiate method [Eshelman *et al.*, 1988] for diagnosis which takes the following steps:

1. Determine events (hypotheses) that potentially explain symptoms.

2. Identify information that can differentiate between candidate explanations by ruling out, providing support for candidates, providing preferences.

3. Get this differentiating information and apply it.

4. If new symptoms become available, go to step 1.

In this example, several types of knowledge are specified in an informal way. First, the domain should provide concepts like: event, symptom, explanation link, preference, rule-out relations etc. Second, a number of basic inference types (similar to KADS knowledge sources, see Ch. 3) are defined: generating a hypothesis, matching a hypothesis to the available data, selection or ordering, compute preferences. In addition there is control knowledge that indicates that all possible candidates are generated given a set of symptoms, and that differentiating information is obtained in a backward manner. These aspects of knowledge would be categorised as task knowledge in KADS.

The differentiating requirements and constraints as laid down in the knowledge model express the specific role that the considered knowledge plays in the problem-solving process. This limits the use that can be made of that knowledge. For instance, a logical implication sentence of a certain type can be specified to be usable only for matching inferences, and not for other types of inference steps (like, say, a selection). KADS notions such as knowledge sources, metaclasses and task structures (see Ch. 3) generally yield the possibility of selecting specific rules or theories needed to produce a certain inference. In this way the knowledge-level model provides *role-limiting constraints* [McDermott, 1988] to the use of knowledge.

2.4.2 The computational level: access limitations Thus, an essential epistemological feature of knowledge-level models is that they specify role limitations of knowledge. The corresponding notion at the computational level is that of access limitations[2]. The knowledge that is specified in a KADS conceptual model cannot be used in arbitrary ways: it has to fulfill certain typing requirements and can only be applied in accordance with the

[2]Cf. Newell's slogan: $representation = knowledge + access$

constraints specified by the model. Given that knowledge components can only be selectively used due to specified role-limiting constraints, the consequence will be a restricted access to the computational representation of those components in a knowledge-based system. Assuming that (as said) the computational complexity results from unrestricted applicability of and access to knowledge, the structure of a knowledge model has a profound effect on the computational adequacy. If the knowledge model sufficiently refines the various knowledge types and explicitly indicates what inferences access what domain structures, the computational complexity will be greatly reduced.

Our position is supported by experiences from AI. Here, a common method to achieve computational tractability is to introduce structural differentiations that at the knowledge level can be characterised as adding new role specialisations and limitations of knowledge. As a matter of fact, this attitude is quite clearly exemplified in the work on generic tasks. Also heuristic classification is a good specimen: if direct association between data and solutions results in a computationally inadequate model, a possible method to obtain a more tractable model is the introduction of additional inference steps, *viz.*, abstraction and refinement. The associated knowledge can now be specified to be accessible only in a restricted part of the 'horse-shoe' inference structure, whereas this is impossible in a simple direct-association model. Yet another interesting example is provided by the work of Patil on medical diagnosis [Patil, 1988], showing that complicated forms of diagnosis can be gradually built up by starting from a simple generate-and-test method and subsequently introducing new elements of knowledge differentiation, so as to preserve the computational adequacy with increasing task complexity.

In conclusion, we suggest that *epistemological role limitations as described by a knowledge-level model are connected to computational access limitations*. Computational adequacy cannot be strictly guaranteed, but the knowledge-level approach does provide significant handles on the computational tractability by means of the role and access limitations ensuing from knowledge differentiation.

2.5 Do Knowledge-level Models Yield Predictions?

Sticklen puts forward as a central objection to the knowledge-level hypothesis in Newell's form its lack of predictive power. In his view, knowledge-level models are capable of explanatory analysis of the reasoning behaviour of intelligent systems in retrospect, but the do not generate empirically verifiable predictions. He discusses this in the context of the broad question whether AI can be considered to be a science when it does not contain a predictive component.

First of all, Sticklen's equation of science with predictive power needs some qualification. On this score, we basically agree with the critical commentaries on his position. Restricting ourselves to physics —Sticklen's favourite example of an "established science"— it is clear that explanatory and predictive power constitute much more subtle ingredients of science than he suggests. Especially in branches of physics that are close to engineering we encounter phenomenological models that have strong predictive power but no explanatory status. This is the case if such models numerically express important regularities of experimental data without referring to more fundamental laws of physics (this may be achieved by, simply speaking, a linear regression analysis of a certain large amount of data that happens to be successful in the general case). Conversely, there are models

that are based upon basic physical principles (and so are felt to be relevant in providing a physically intuitive picture of a process) but refer to phenomena that are far beyond any experimental test (such as processes on time scales that are orders-of-magnitude shorter than can be measured)[3]. In general, physical theory is an iceberg of which only a tiny part is visible for empirical verification. An important aspect is that physical theories only yield predictions at a certain level and within a certain regime. For instance, macro-physical theories such as hydrodynamics do make predictions about, say, fluid pressure and waves, but not about microphysical entities like atomic interaction potentials within water molecules. The latter notions do not even exist at the macro-physical level, whereas in the microphysical theory there is no place for concepts like fluid, waves and pressure. Thus, each level of physical description generates its own type of predictions.

This parallel carries over to modelling for knowledge-based systems. A knowledge-level model describes what types of reasoning steps an agent that is being modelled is expected to take in performing a certain task. In addition it makes certain claims about the structure of the domain-specific knowledge involved, and it specifies the strategic elements of the reasoning. For example, the cover-and-differentiate knowledge model outlined earlier tells us that certain hypotheses will be discarded in a *rule-out* inference step. How such an inference will operationally manifest itself depends on the system under investigation. An AI program may print out the removal of a hypothesis from the list of current candidates, while a human expert may utter a natural language sentence implying that the hypothesis is no longer considered. Although the operational form may differ, in both cases the ruling-out of a hypothesis is in principle empirically verifiable. Similarly, the heuristic-classification model would predict that much of the reasoning effort would be concentrated in data reduction and not in hypothesis handling, as is the case in the cover-and-differentiate method.

Thus, a knowledge-level model may be applied to both human and artificial problem solvers. In either case, however, it is hard to see why knowledge models as we have sketched them possess no predictive power, as seems to be Sticklen's complaint. But it has to be acknowledged that the corresponding predictions are of a certain kind only, namely, on the level of the reasoning steps —which type, under what knowledge conditions— that we expect an agent to perform. No predictions are made concerning the detailed, operational or symbol-level of observation. As corroborated by much of the literature concerning the philosophy of science, testing a theory requires an interpretation of the observations that we make concerning the real world. Evidently, the regime of prediction (and, thus, of validity) of knowledge-level models is limited — in this case to the type of and the conditions for inference steps to be carried out by an intelligent agent. Nevertheless, as pointed out, a limited regime or level at which predictions can be made is standard not only in AI but also in physics. Consequently, we disagree with Sticklen's criticism on the predictive power of knowledge level models.

[3]Our examples are based upon personal experience in mainstream nuclear physics and engineering. If we accept as a simple operational definition of what may count as science: the publication of work in refereed international journals on a regular basis (the latter to rule out incidental mistakes of reviewers), all these examples must be accepted as science. We can provide the interested reader with pertinent references to the literature.

2.6 System Building on the Basis of Knowledge-level Models

Knowledge-level models do not contain all information necessary for the implementation of a system. In the KADS approach to developing knowledge-based systems, a separate *design model* is introduced [Schreiber *et al.*, 1988] (see Ch. 3). In this design model appropriate AI techniques and representations are selected to realise the problem solving behaviour specified in the knowledge-level model. The design model is thus a specification of the symbol level notions such as data structures and processes. The design model is also the place where additional, *symbol level*, control (cf. Sec. 2.3) is specified.

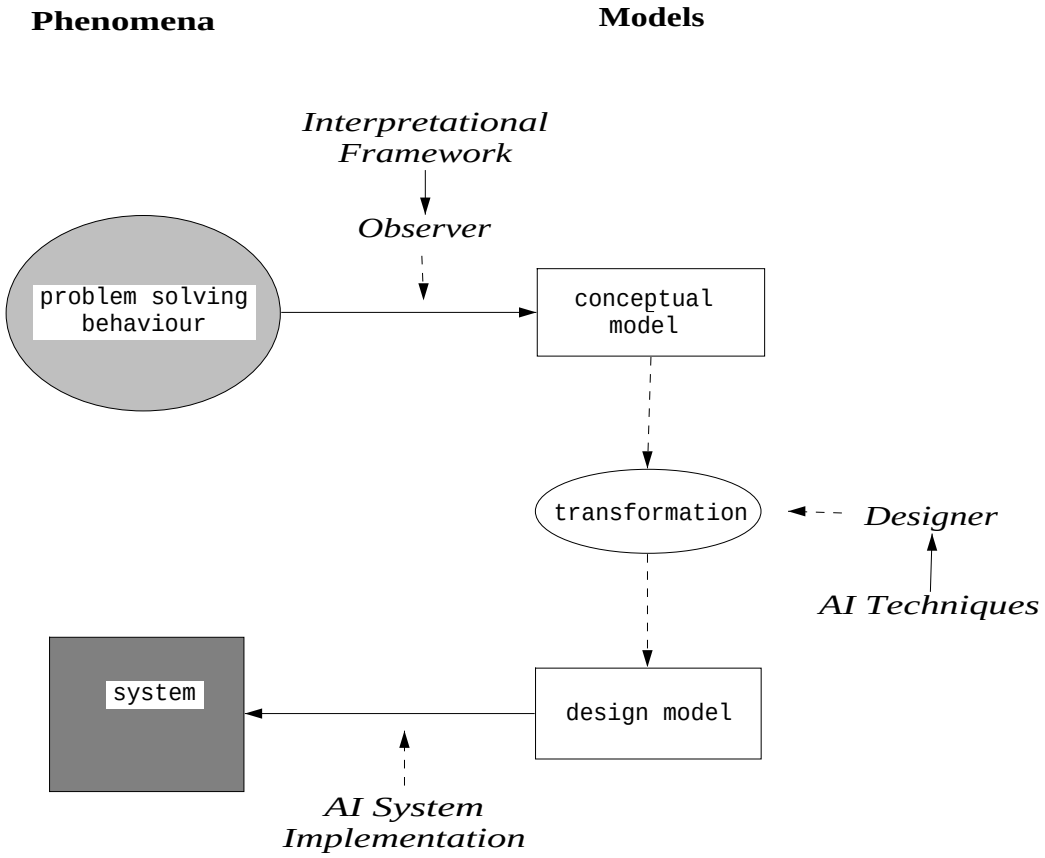

FIGURE 2.1: Role of the conceptual (knowledge-level) model and of the design (symbol level) model in the development of knowledge-based systems

Fig. 2.1 provides a graphical representation of the different roles that the knowledge-level model and the design model in our view play in the development process of a knowledge-based system. The knowledge-level model is constructed by an observer of problem solving phenomena (e.g. human expertise). The observer (knowledge analyst) is aided in this task by an interpretational framework, that should consist of two parts: (i) a vocabulary for describing various knowledge types, such as the categorisations in the KADS conceptual model, and (ii) generic knowledge components. These generic components are partial instantiations of knowledge reoccurring in a class of tasks and/or domains. Heuristic classification, cover-and-differentiate and the interpretation models in KADS [Breuker

et al., 1987] are examples of such generic components.

The design model is constructed by transforming the knowledge-level descriptions into symbol-level descriptions through the selection of appropriate AI techniques and representations, that realise the specified problem solving behaviour. The design model provides the basis for implementing the physical system. Although the designer is in principle free to develop any design model that meets the requirements of the knowledge-level model, we are strongly in favour of a *structure-preserving* design. With this we mean that there is a structural correspondence between knowledge-level elements and symbol level elements. Structural correspondence paves also the way for explaining the computation of a program at various levels of abstraction. Thus, design should basically be a process of *adding* symbol level information, such as operational control, to the knowledge-level model (see Ch. 6).

2.7 Conclusions

In this chapter we have investigated some points of criticism on Newell's Knowledge-level Hypothesis: the inability to represent control, the potential computational inadequacy, the lack of predictive power and the non-operational character (the problem of 'how to build it').

We have pointed at a possible confusion between two types of control. We have indicated that in our view there is a need to model in a practical knowledge theory of problem solving a particular type of control knowledge with a strong knowledge-level flavour as for example decompositions of knowledgeable agents and meta-knowledge about agents. If this type of knowledge-level control was not part of Newell's original hypothesis, we feel that it should be extended in this respect. We disagree with an extension such as Sticklen's Knowledge-level Architecture Hypothesis, as it requires a type of control specification that inherently belongs to the symbol level. We have argued that the role-limitations in knowledge-level descriptions give rise to access limitations at the symbol level. Together they provide important safe-guards against potential computational inadequacy. We have also explained that knowledge-level models as we propose them allow limited forms of prediction – namely at the level of the type of and the conditions for inference steps carried out by an intelligent agent. We have shown that this is not different from the role of formal theories in predicting physical phenomena. Finally, we have argued that in the process of system building there is a place for a separate design model. We view design as a process of adding symbol level information to a knowledge-level model. Notational devices, catalogues, and shellifications can support the design and implementation, but by nature these activities are open-ended, i.e. the solution space is large.

Although we feel that the original Knowledge-level Hypothesis still stands, much work still needs to be done to "make it work": to arrive at a practical knowledge-level theory of problem solving. Looking at the various approaches to knowledge-level modelling it is clear that the terminology used to describe the knowledge level is confusing and ambiguous. In our opinion there is a clear need for a *formal* framework for describing knowledge-level models. We fully agree with Newell that the major role of logic in AI should be to support the analysis of the knowledge level [Newell, 1982; pp. 121-122]. In [Akkermans *et al.*, 1992] a first effort is made to devise a logical framework for knowledge-level analysis. Although this work can only be seen as a first start, we feel that this is one important research

direction for improving the state of the art in knowledge acquisition.

Acknowledgement Frank van Harmelen provided useful comments on an earlier version of this chapter.

Chapter 3

KADS: A Modelling Approach to KBS Development

This chapter discusses the KADS approach to knowledge engineering. In KADS, the development of a knowledge-based system (KBS) is viewed as a modelling activity. A KBS is not a container filled with knowledge extracted from an expert, but an operational model that exhibits some desired behaviour that can be observed in terms of real-world phenomena. Three basic principles underlying the KADS approach are discussed, namely (i) the introduction of partial models as a means to cope with the complexity of the knowledge engineering process, (ii) the KADS four-layer framework for modelling the required expertise, (iii) the reusability of generic model components as templates supporting top-down knowledge acquisition, The actual activities that a knowledge engineer has to undertake are briefly discussed. We compare the KADS approach to related approaches and discuss experiences and future developments. The approach is illustrated throughout the chapter with examples in the domain of troubleshooting audio equipment.

This chapter represents a shortened and slightly revised version of an article published in the Knowledge Acquisition Journal. It is co-authored by Bob Wielinga and Joost Breuker. Reference: B. J. Wielinga, A. Th. Schreiber, and J. A. Breuker. KADS: A modelling approach to knowledge engineering. *Knowledge Acquisition*, 4(1):5-53, 1992. Special issue "The KADS approach to knowledge engineering".

3.1 Introduction

This chapter gives an overview of results of a European research project, in Europe commonly known as the KADS project (ESPRIT-I P1098). This project aimed at the development of a comprehensive, commercially viable methodology for knowledge-based system (KBS) construction. When the KADS project was conceived, sometime in 1983, little interest in methodological issues existed in the AI community. The prevailing paradigm for building knowledge-based systems was rapid prototyping using special purpose hard- and software, such as LISP machines, expert system shells etc. Since then, many organisations have become aware of the fact that KBS development from an organisational point of view does not differ much from the development of other types of information systems. Aspects of KBS development such as information analysis, application selection, project management, user requirement capture, modular design, reusability etc, are similar to those encountered in conventional system development. Problems that frequently occur in conventional information system development projects are amplified in the case of KBS

development. The wider capabilities of KBS technology allow more complex applications, which have a stronger impact on organisational structure than most conventional systems and often require a more sophisticated user-system interaction than is the case with conventional systems. Additionally, KBS development poses a number of problems of its own.

An often cited problem in KBS construction is the *knowledge acquisition bottleneck*. It turns out to be very difficult to extract the knowledge that an expert has about how to perform a certain task efficiently in such a way that the knowledge can be formalised in a computer system. The actual realisation of a KBS system often poses problems as well. The reasoning methods that are used in KBS's are not always fully understood. Although the AI literature abounds in methods and techniques for modelling reasoning processes, their description is not uniform and unambiguous. So, the need for a sound methodology for KBS development has become recognised over the last few years.

In this chapter we will discuss some principles that comprise the framework on which the KADS methodology is founded and describe its main ingredients.

3.2 Views on Knowledge Acquisition

During the knowledge acquisition process the knowledge that a knowledge-based system (KBS) needs in order to perform a task, is defined in such a way that a computer program can represent and adequately use that knowledge. Knowledge acquisition involves in our view at least the following activities: *eliciting* the knowledge in an informal — usually verbal — form, *interpreting* the elicited data using some conceptual framework, and *formalising* the conceptualisations in such way that the program can use the knowledge. In this chapter we will mainly focus on the interpretation and formalisation activities in knowledge acquisition. Elicitation techniques have been the subject of a number of recent papers and their role in the knowledge acquisition process is now reasonably well understood [Neale, 1988; Breuker *et al.*, 1987; Diaper, 1989; Meyer & Booker, 1991].

Traditionally the knowledge acquisition process was viewed as a process of extracting knowledge from a human expert and transferring the extracted knowledge into the KBS. In practice this often means that the expert is asked what rules are applicable in a certain problem situation and the knowledge engineer translates the natural language formulation of these rules into the appropriate format. Several authors [Hayward *et al.*, 1987; Morik, 1989] have pointed out that this transfer-view of knowledge acquisition is only applicable in very few cases. The expert, the knowledge engineer and the KBS should share a common view on the problem solving process and a common vocabulary in order to make knowledge transfer a viable way of knowledge acquisition. If the expert looks at the problem or the domain in a different way than the knowledge engineer, asking for rules or similar knowledge structures and translating them into the knowledge representation formalism of the system, does not work.

A different view on knowledge acquisition is that of a modelling activity. 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. The use of models is a means of coping with the complexity of the development process.

Constructing a KBS is seen as building a computational model of desired behaviour. 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 behaviour of the KBS. However, a KBS is hardly ever the functional and behavioural equivalent of an expert. There are a number of reasons for this. Firstly, the introduction of information technology often involves new distributions of functions and roles of agents. The KBS may perform functions which are not part of the experts repertory. Secondly, the underlying reasoning process of the expert can often not be made fully explicit. Knowledge, principles and methods may be documented in a domain, but these are aimed at a human interpreter and are not descriptions of how to solve problems in a mechanical way. Thirdly, there is an inherent difference between the capabilities of machines and humans. For example, in an experiment in a domain of configuring moulds [Barthèlemy *et al.*, 1988] a decision was made to generate all possible solutions instead of the small set generated by experts. The decision was guided by the fact that for a machine it presents no problem to store a large number of hypotheses in short-term memory, whereas for humans this is impossible.

So, in the modelling view knowledge acquisition essentially is a constructive process in which the knowledge engineer can use all sorts of data about the behaviour of the expert, but in which the ultimate modelling decisions have to be made by the knowledge engineer in a constructive way. In this sense knowledge engineering is similar to other design tasks: the real world only provides certain constraints on what the artefact should provide in terms of functionality, the designer will have to aggregate the bits and pieces into a coherent system.

In KADS we have adopted the modelling perspective on knowledge acquisition. The KADS approach can be characterised through a number of principles that underlie the process to building knowledge-based systems, namely:

- The introduction of multiple models as a means to cope with the complexity of the knowledge engineering process.
- The KADS four-layer framework for modelling the required expertise.
- The reusability of generic model components as templates supporting top-down knowledge acquisition.
- The process of differentiating simple models into more complex ones.
- The importance of structure-preserving transformation of models of expertise into design and implementation.

The first three principles are discussed in this chapter, respectively in Sec. 3.3, Sec. 3.4 and Sec. 3.5. The fourth principle is discussed in more detail in Ch. 5; the fifth principle in Ch. 6.

Although a description of the use of KADS on practical KBS projects is outside the scope of this chapter, we look briefly at the actual knowledge engineering process (Sec. 3.6). We also compare the KADS approach to other approaches (Sec. 3.7). Finally we discuss experiences and future developments (Sec. 3.8 and Sec. 3.9).

The approach is illustrated throughout this chapter with examples, most of them in the domain of diagnosing and correcting malfunctions of an audio system.

3.3 Principle 1: Multiple Models

The construction of a knowledge-based system is a complex process. It can be viewed as a search through a large space of knowledge-engineering methods, techniques and tools. Numerous choices have to be made with regard to elicitation, conceptualisation and formalisation. Knowledge engineers are thus faced with a jungle of possibilities and find it difficult to navigate through this space.

The idea behind the first principle of KADS is that the knowledge-engineering space of choices and tools can to some extent be controlled by the introduction of a number of *models*. A model reflects, through abstraction of detail, selected characteristics of the empirical system in the real world that it stands for [DeMarco, 1982]. Each model emphasises certain aspects of the system to be built and abstracts from others. Models provide a decomposition of knowledge-engineering tasks: while building one model, the knowledge engineer can temporarily neglect certain other aspects. The complexity of the knowledge-engineering process is thus reduced through a divide-and-conquer strategy.

In this section we discuss a number of models, namely (i) the organisational model, (ii) the application model, (iii) the task model, (iv) the model of cooperation, (v) the model of expertise, (vi) the conceptual model, and (vii) the design model.

We use the term *knowledge engineering* in a broad sense to refer to the overall process of KBS construction (i.e. the construction of all these models and the artefact) and the term *knowledge acquisition* in a more restricted sense to refer to those parts of this construction process that are concerned with the information about the actual problem solving process. The scope of the present chapter is limited to the knowledge acquisition aspects. Other knowledge engineering aspects are only briefly addressed.

3.3.1 Organisational model, application model and task model

In KADS we distinguish three separate steps in defining the goals of KBS construction, namely:

1. Defining the *problem* that the KBS should solve in the organisation.
2. Defining the *function* of the system with respect to future users (which can be either humans or possibly other systems).
3. Defining the actual *tasks* that the KBS will have to perform.

In this section we discuss three models that address parts of this three-step process. The first two are discussed briefly as these are outside the scope of this chapter.

Organisational model An organisational model provides an analysis of the socio-organisational environment in which the KBS will have to function. It includes a description of the functions, tasks and bottlenecks *in the organisation*. In addition, it describes (predicts) how the introduction of a KBS will influence the organisation and the people working in it. This last activity can be viewed as a type of *technology assessment* [de Hoog *et al.*, 1990]. We have experienced [de Hoog, 1989; van der Molen & Kruizinga, 1990] that it is dangerous to ignore the impact of the interaction between the construction of a KBS and the resulting changes in the organisation. Neglecting this aspect may lead to a system that is not accepted by its prospective users. It is also important to realise that the process of KBS construction itself can, by its nature (for instance, through extensive interviewing), change the organisation in such a way that it becomes a moving target [van der Molen &

Kruizinga, 1990]. The result may be that the final system is aimed at solving a problem that does not exist any more in the organisation. We are convinced that the organisational viewpoint is important throughout the KBS construction process.

Application model An application model defines what problem the system should solve in the organisation and what the function of the system will be in this organisation. For example, the daily operation and fault handling of an audio system can pose serious problems for people who are not familiar with or just not interested in more than the superficial ins-and-outs of such a system. A potential solution to this problem could be the development of a knowledge-based system. The function of this system would be to ensure that the owner of the audio system is supported in the process of correcting operational malfunctions of the audio system.

In addition to the *function* of the KBS and the *problem* that it is supposed to solve, the application model specifies the *external constraints* that are relevant for the development of the application. Examples of such constraints are the required speed and/or efficiency of the KBS and the use of particular hardware or software.

Task model A task model specifies how the function of the system (as specified in the application model) is achieved through a number of tasks that the system will perform. Establishing this relation between function and task is not always as straightforward as it may seem. For example, consider a problem such as the medical care of patients with acute infections of the bloodstream. One approach to solve this problem is to perform the following tasks: (i) determine the identity of the organism that causes the infection and (ii) select on the basis of that diagnosis the optimal combination of drugs to administer to the patient. In real life hospital practice however, the recovery of the patient is the primary concern. So, if identification of the organism proves difficult, e.g. because no laboratory data are available, a therapy will be selected on other grounds. In fact, some doctors show little interest in the precise identity of the organism causing an infection as long as the therapy works. Stated in more general terms: given a goal that a system should achieve, there may be several alternative ways in which that goal can be achieved. Which alternative is appropriate in a given application depends on characteristics of that application, on availability of knowledge and data, and on requirements imposed by the user or by external factors.

With respect to the content of the task model, we distinguish three facets: (i) task decomposition, (ii) task distribution, and (iii) task environment:

Task decomposition A task is identified that would achieve the required functionality. This task is decomposed in sub-tasks. A technique such as *rational task analysis* is often used to achieve such a decomposition. We call the composite top-task a "real-life task", as it often represents the actual task that an expert solves in the application domain. The sub-tasks are the starting point for further exploration, such as the modelling of expertise and cooperation. A simple decomposition of a real-life task in the audio domain is shown in Fig. 3.1.

Each separate task is described through an input/output specification, where the output represents the goal that is achieved with the task and the input is the information that is used in achieving this goal. What constitutes the goal of a task

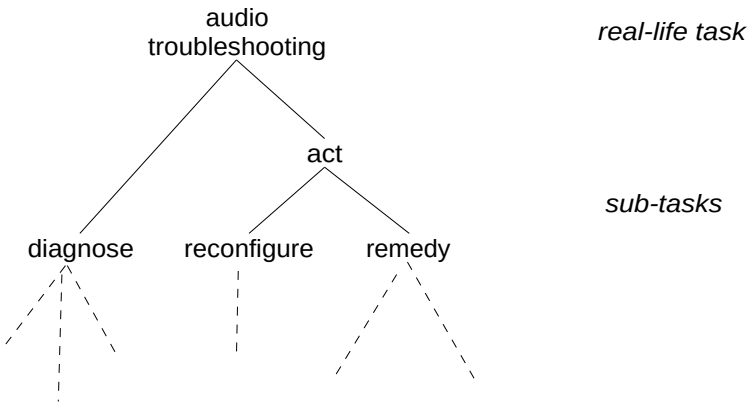

FIGURE 3.1: Task decomposition for the audio example

is not always self-evident. Even for a seemingly well understood task such as diagnosis, it is not always clear what a diagnosis of a faulty system means. A diagnosis could be the identification of a subsystem (a component of an audio system) that malfunctions, or it could be a full causal model of how a malfunction came about. Similarly the result of a design task could be a detailed description of the structure of a system (e.g. a device for monitoring patients in an intensive care unit) or it could be a description of the functionality, structure and use of the device.

Task distribution The task distribution is the *assignment* of tasks to *agents*. Example agents are the KBS, the user or some other system. The last two agents are called *external* agents. Given the task decomposition the knowledge engineer has to decide what subtasks to assign to the system and what tasks to the user. These decisions constitute essentially cognitive engineering problems [Roth & Woods, 1989]: they should be made on the basis of an analysis of the user requirements and expectations, the knowledge and skills that the user has, and the potential capabilities and limitations of the system.

Task environment The nature of the task-domain itself usually enforces a number of constraints on how the task can be performed. We call these constraints the *task environment*. For example, the task environment of a support system for handling malfunctions in an audio system could consist of the following constraints:

- The KBS is not a physical part of the audio system.
- It has no sensors to make observations (and thus depends on the user to do this).
- It has no robot arm to perform reconfigurations and/or repairs (and thus again depends on the user to do this),
- The KBS users will be novices, who are not expected to be able to understand technical terms or to examine the interiors of the audio system.

The constraints posed by the task environment influence both the scope and the nature of the models of expertise and cooperation (see further).

The task model and its role in specifying system-user interaction is discussed in more detail in [de Greef & Breuker, 1992]

3.3.2 Model of cooperation The task model consists of a decomposition of the real-life task into a number of primitive tasks and a distribution of tasks over agents. The model of cooperation contains a specification of the functionality of those sub-tasks in the task model that require a cooperative effort. These tasks can for instance be data acquisition tasks activated during problem solving or various types of explanation tasks. Such tasks are called *transfer* tasks, as they involve transferring a piece of information from the system to an external agent or vice versa.

There is thus a clear dependency between the model of cooperation and the model of expertise. Some of the sub-tasks will be achieved by the system, others may be realised by the user. For example, in a diagnostic task in the audio example, the system may suggest certain tests to be performed by the user, while the user will actually perform the tests and will report the observed results back to the system. Alternatively, the user may want to volunteer a solution to the diagnostic problem while the system will criticise that solution by comparing it with its own solutions.

The result is a model of cooperative problem solving in which the user and the system together achieve a goal in a way that satisfies the various constraints posed by the task environment, the user and the state of the art of KBS technology. The modelling of cooperation is outside the scope of this chapter, but is discussed in more detail in [de Greef & Breuker, 1992; de Greef *et al.*, 1988a; de Greef & Breuker, 1989].

3.3.3 Model of expertise Building a model of expertise is a central activity in the process of KBS construction. It distinguishes KBS development from conventional system development. Its goal is to specify the problem solving expertise required to perform the problems solving tasks assigned to the system.

One can take two different perspectives on modelling the expertise required from a system. A first perspective – one that is often taken in AI – is to focus on the computational techniques and the representational structures (e.g. rules, frames) that will provide the basis of the implemented system. A second perspective focuses on the behaviour that the system should display and on the types of knowledge that are involved in generating such behaviour, abstracting from the details of how the reasoning is actually realised in the implementation. These two perspectives correspond to the distinction Newell makes between respectively the *symbol level* and the *knowledge level* [Newell, 1982].

We take the second perspective and view the model of expertise as being a knowledge-level model. The model of expertise specifies the desired problem solving behaviour for a target KBS through an extensive categorisation of the knowledge required to generate this behaviour. The model thus fulfills the role of a *functional specification* of the problem solving part of the artefact. As stated previously, it is not a cognitive model of the human expert. Although the construction of the model of expertise is usually guided by an analysis of expert behaviour, it is biased to what the target system should and can do.

In modelling expertise we abstract from those sub-tasks that specify some form of cooperation with the user. For example, in the audio domain we could identify two tasks that require such interactions: *performing a test* and *carrying out a reconfiguration*. In the model of expertise, such interaction or *transfer* tasks are specified more or less as a

black box (see Sec. 3.4.3). The detailed study of the nature of these transfer tasks is the subject of the modelling of cooperation.

As the model of expertise plays a central role in KBS development, its details are discussed extensively in Sec. 3.4.

3.3.4 Conceptual model = model of expertise + model of cooperation Together, the model of expertise and the model of cooperation provide a specification of the behaviour of the artefact to be built. The model that results from merging these two models is similar to what is called a *conceptual model* in database development. Conceptual models are abstract descriptions of the objects and operations that a system should know about, formulated in such a way that they capture the intuitions that humans have of this behaviour. The language in which conceptual models are expressed is not the formal language of computational constructs and techniques, but is the language that relates real world phenomena to the cognitive framework of the observer. In this sense conceptual models are subjective, they are relative to the cognitive vocabulary and framework of the human observer. Within KADS we have adopted the term "conceptual model" to denote a combined, implementation-independent, model of both expertise and cooperation.

3.3.5 Design model The description of the computational and representational techniques that the artefact should use to realise the specified behaviour is not part of the conceptual model. These techniques are specified as separate *design decisions* in a design model. In building a design model, the knowledge engineer takes external requirements such as speed, hardware and software into account. Although there are dependencies between conceptual model specifications on the one hand and design decisions on the other hand, we have experienced that building a conceptual model model without having to worry about system requirements makes life easier for the knowledge engineer.

The separation between conceptual modelling on the one hand and a separate design step on the other hand has been identified as both the strength and the weakness of the KADS approach [Karbach *et al.*, 1990].

The main advantage lies in the fact that the knowledge engineer is not biased during conceptual modelling by the restrictions of a computational framework. KADS provides a more or less universal framework for modelling expertise (see the next section) and although computational constraints play a role in the construction of such models (cf. Ch. 5) experience[1] has shown that this separation enables knowledge engineers to come up with more comprehensive specifications of the desired behaviour of the artefact. The disadvantage lies in the fact that the knowledge engineer, after having built a conceptual model, is still faced with the problem of how to implement this specification. In Ch. 6 we discuss some principles that can guide the knowledge engineer in this design process.

Fig. 2.1 (see previous chapter) summarises the different roles which the conceptual model and the design model play in the knowledge engineering process. An observer (knowledge engineer) constructs a conceptual, knowledge-level, model of the artefact by abstracting from the behaviour of experts. This abstraction process is aided by the use of an interpretational framework, such as generic models of classes of tasks or task-domains. The conceptual model is real-world oriented in the sense that it is phrased in real-world

[1] See Sec. 3.8 for an overview of applications developed with the KADS approach

terminology and can thus be used as a communication vehicle between knowledge engineer and expert. The conceptual model does not take detailed constraints with regard to the artefact into account. The design model on the other hand is a model that is phrased in the terminology of the artefact: it describes how the conceptual model is realised with particular computational and representational techniques.

Fig. 3.2 shows the dependencies between the models discussed in this section. Connections indicate that information from one model is used in the construction of another model. The actual activities in the construction process do not necessarily have to follow the direction from organisation model to system. In fact, several life-cycle models have been developed, each defining various phases and activities in building these models. The first life-cycle model developed in KADS [Barthèlemy *et al.*, 1987] was of the water-fall type. At the end of the KADS project, a new life-cycle was defined [Taylor *et al.*, 1989] based on the concept of a spiral model [Boehm, 1988].

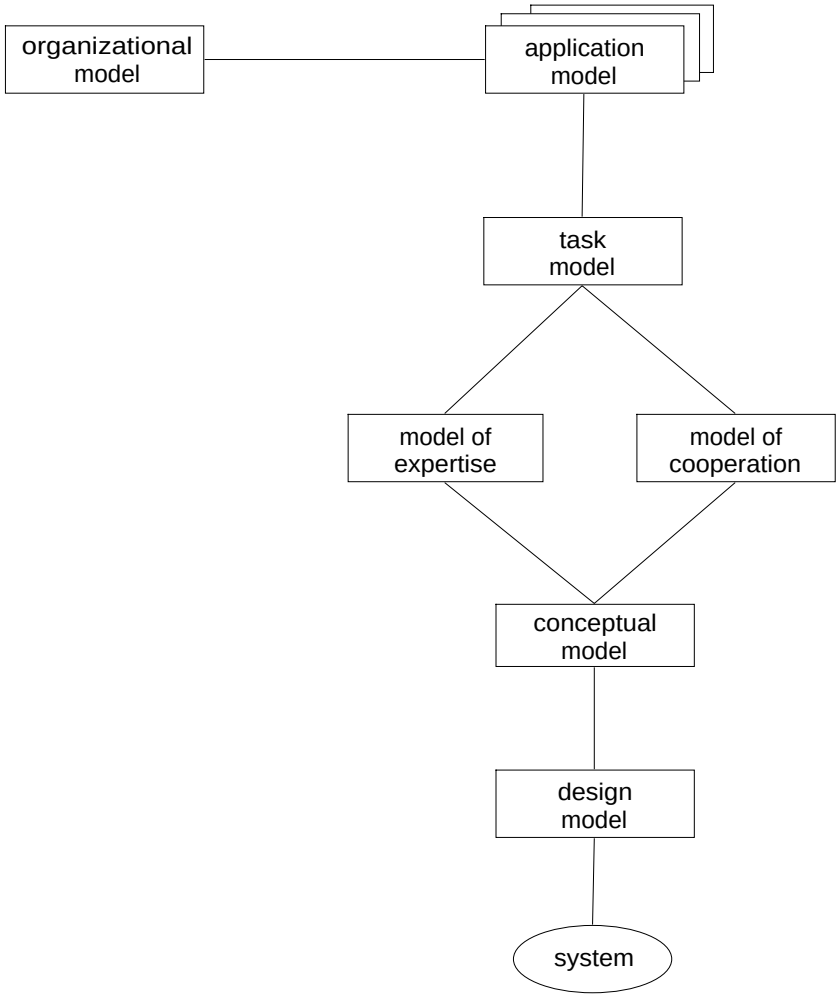

FIGURE 3.2: Principle 1: Models provide a decomposition of the knowledge-engineering task

The nature of knowledge engineering thus becomes a process that bridges the gap between required behaviour and a system that exhibits that behaviour through the creation

of a set of models. Summarising, we can say that the KADS modelling view of knowledge acquisition gives rise to a methodology that involves the construction of a variety of models in the course of the knowledge engineering process. Each model represents a particular view on the KBS. They allow the knowledge engineer to cope with the complexity of the knowledge engineering process through a "divide & conquer" strategy.

The remainder of this chapter focuses mainly on the model of expertise, as it plays such a central role in KBS development.

3.4 Principle 2: Modelling Expertise

The major challenge for any modelling approach to KBS construction is to find an adequate answer to the question of how to model expertise. It is this aspect of the system that distinguishes KBS development from the development of conventional systems. As discussed previously, we require of the resulting model of expertise that it is independent of a particular implementation. In this section a framework for modelling expertise is outlined. Slightly different versions of this KADS approach to modelling expertise (usually called the "four-layer model") have been presented in [Wielinga & Breuker, 1986; Hayward *et al.*, 1987; Schreiber *et al.*, 1988; Breuker & Wielinga, 1989]

Two basic premises underly the ideas presented here. First, we assume that it is possible and useful to distinguish between several generic types of knowledge according to different *roles* that knowledge can play in reasoning processes. Second, we assume that these types of knowledge can be organised in several *layers*, which have only limited interaction. A first distinction that is often made is the one between *domain knowledge* and *control knowledge*. Here we will take such a separation of knowledge in two layers one step further, and will argue for a refined distinction of different types of control knowledge at three levels.

The categories in which the expertise knowledge can be analysed and described are based on epistemological distinctions: they contain different types of knowledge. We distinguish between:

1. Static knowledge describing a declarative *theory* of the application domain (domain knowledge).
2. Knowledge of different *types of inferences* that can be made in this theory (first type of control knowledge).
3. Knowledge representing *elementary tasks* (second type of control knowledge).
4. *Strategic knowledge* (third type of control knowledge)

Each of these categories of knowledge is described at a separate level. The separation reflects different ways in which the knowledge can be viewed and used. In the following sections each of the four categories of knowledge distinguished in KADS is discussed in more detail.

The distinction between different types of knowledge is not new. Several authors have reported ideas which pertain to the separation of domain and control knowledge, and have proposed ways to increase the flexibility of control in expert systems. The work of Davis [Davis, 1980] introduced explicit control knowledge as a means to control inference processes in a flexible way. In the NEOMYCIN system [Clancey, 1985a] different functions of knowledge are explicated by separating domain knowledge and control knowledge and

by introducing an explicit description of the strategies that the system uses. Pople (1982) [Pople, 1982] has stressed the problem of the right task formulation. He considers it to be a fundamental challenge for AI research to model the control aspects of the reasoning process of expert diagnosticians which determines the optimal configuration of tasks to perform in order to solve a problem.

3.4.1 Domain knowledge The domain knowledge embodies the *conceptualisation* of a domain for a particular application in the form of a *domain theory*. The primitives that we use to describe a domain theory are based on the epistemological primitives proposed by [Brachman & Schmolze, 1985]: concepts, properties, two types of relations, and structures:

Concept *Concepts* are the central objects in the domain knowledge. A concept is identified through its name (e.g. `amplifier`).

Property/Value Concepts can have *properties*. Properties are defined through their name and a description of the values that the property can take. For example, `amplifier` has a property `power` with as possible values `on/off`.

Relation between concepts A first type of relation is the relation between concepts, for example `amplifier is-a component`. The most common relations of this type are the sub-class relation and the part-of relation. Several variants of these two relations exist, each with its own semantics.

Relation between property expressions A second type of relation is the relation between expressions about property values. An expression is a statement about the value(s) of a property of a concept, e.g. `amplifier:power = on`.[2] Examples of this type of relation are causal relations and time relations. An example of a tuple of a causal relation in the audio domain could be:

```
amplifier:power-button = pressed CAUSES amplifier:power = on
```

Structure A structure is used to represent a complex object: an object consisting of a number of objects/concepts and relations. For example, the audio system as a whole can be viewed as a structure, consisting of several components and relations (part-of, wire connections) between these components.[3]

The choice of this set of primitives is in a sense arbitrary and probably somewhat biased by the types of problems that have been tackled with KADS. The problem is to find a subset that provides the knowledge-engineer with sufficient expressive power. One could consider including additional special-purpose primitives such as mathematical formulae. There is clearly a link here with research in the field of semantic database modelling (see for an overview [Hull & King, 1987]).

The primitives are used to specify what we call a *domain schema* for a particular application. A domain schema is a description of the *structure* of the statements in the

[2] We use the shorthand <concept>:<property> for "the <property> of <concept>"'.

[3] The term "structure" as used here should not be confused with the "structural descriptions" in KL-ONE.

domain theory. It is roughly comparable to the notion of a signature in logic.[4] For example, in a domain schema we could specify that the domain theory contains `part-of` relations between `component` concepts without worrying about the actual tuples of this relation. We prefer to use the term "schema" rather than "ontology" to stress the fact that the domain theory is the product of knowledge engineering and thus not necessarily describes an inherent structure in the domain (as the word "ontology" would suggest).

The domain schema specifies the main decisions that the knowledge engineer makes with respect to the *conceptualisation* [Genesereth & Nilsson, 1987; Nilsson, 1991] of the domain. For example, when a domain schema for a diagnostic domain is constructed, a decision has to be made whether "correct" or "fault" models (or both) are part of the domain theory. Parts of a domain schema often reappear in similar domains and could be reused (see Sec. 3.5 for a more detailed discussion of reusability). The domain schema also provides convenient handles for describing the way in which inference knowledge uses the domain theory. Issues related to the interaction between domain knowledge and inference knowledge are discussed in the next section. In Ch. 4 a language for describing domain schemata is presented.

An example domain schema of a simple domain theory for diagnosing faults in an audio system is shown in Table 3.1.[5] Two types of concepts appear in this theory: *components* and *tests*. Both components and tests can have properties: respectively a *state-value* and a *value*. Two relations are defined between concepts of type "component": *is-a* and *sub-component-of*. In addition, two relations between property expressions are defined: (i) a *causal* relation between state values of components, and (ii) an *indicates* relation between test values and state values.

Fig. 3.3 shows some domain knowledge in the audio domain. The domain knowledge description follows the structure defined in the domain schema of Table 3.1.

Domain knowledge can be viewed as a declarative theory of the domain. In fact, adding a simple deductive capability would enable a system in theory (but, given the limitations of theorem-proving techniques, not in practice) to solve all problems solvable by the theory. The domain knowledge is considered to be relatively task neutral, i.e. represented in a form that is independent of its use by particular problem solving actions. There is ample evidence [Wielinga & Bredeweg, 1988] that experts are able to use their domain knowledge in a variety of ways, e.g. for problem solving, explanation, teaching etc. Separating domain knowledge embodying the theory of the domain from its use in a problem solving process, is a first step towards flexible use and reusability of domain knowledge.

3.4.2 Inference knowledge At the first layer of control knowledge we abstract from the domain theory and describe the inferences that we want to make in this theory. We call this layer the *inference layer*. An inference specified at the inference level is assumed to be *primitive* in the sense that it is fully defined through its name, an input/output

[4] The relation between property expressions corresponds to an axiom schema; structures correspond to a sub-theory.

[5] The description of the domain schema given here is rather informal. For example, nothing is said about cardinality (e.g. can a property have one or more values at some point in time). Techniques exist for describing these schemata in a more precise and formal way, e.g. [Davis & Bonnel, 1990; Hull & King, 1987].

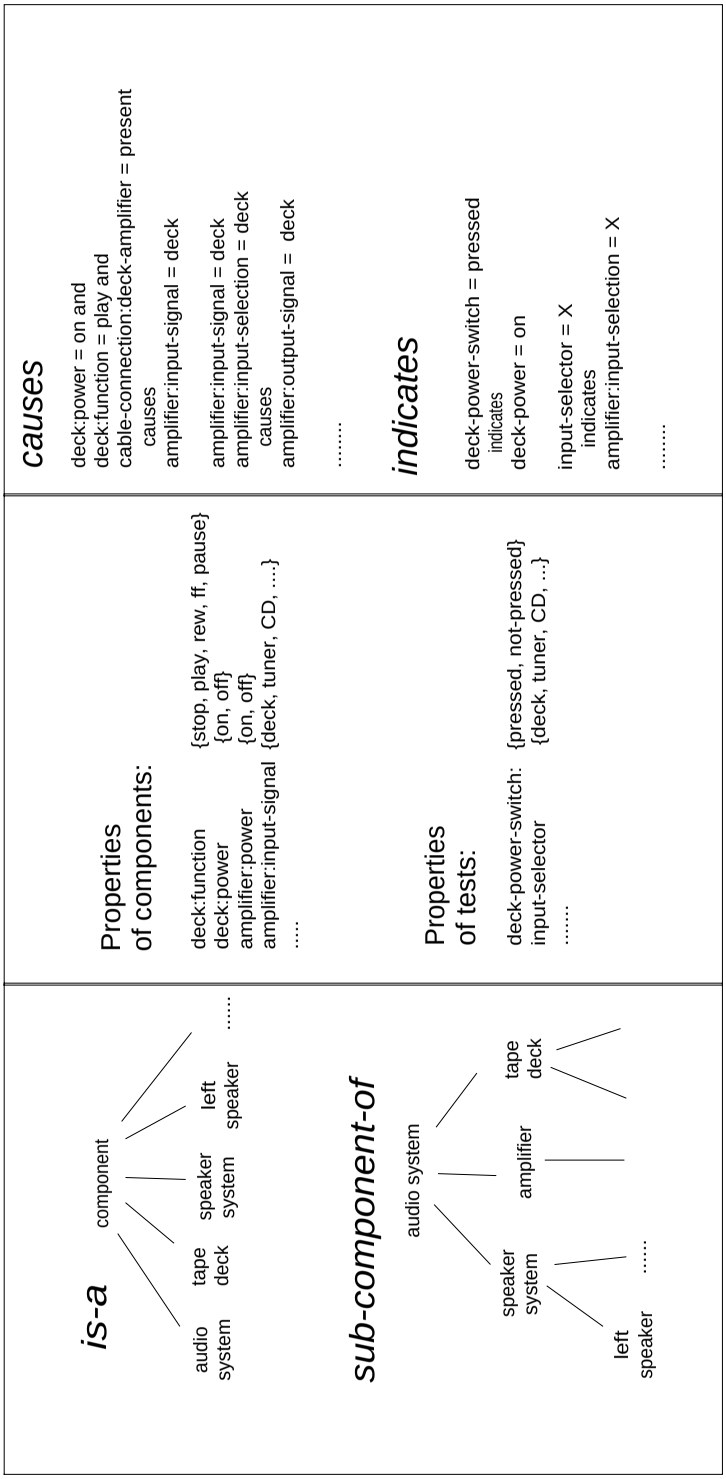

FIGURE 3.3: Domain knowledge of the audio system using the schema described in Table 3.1

Primitive	Name	Description
Concept	component	The elements of the audio system
Relation between concepts	component IS-A component	Sub-type hierarchy of components of the audio system
Relation between concepts	component SUB-COMPONENT-OF component	Part-of hierarchy of components of the audio system
Property	component:state-value	Components have properties describing the state that components are in at some moment in time.
Relation between expressions	component:state-value CAUSES component:state-value	Causal relations that specify how normal state-values of components are causally related to each other.
Concept	test	Test that can be performed to establish a state of an audio system.
Property	test:value	Possible outcomes of a test.
Relation between expressions	test:value INDICATES component:state-value	A relation describing which internal state is indicated by a particular test outcome.

TABLE 3.1: A domain schema for diagnosing faults in an audio-system

specification and a reference to the domain knowledge that it uses. The actual way in which the inference is carried out is assumed to be irrelevant for the purposes of modelling expertise. From the viewpoint of the model of expertise no control can be exercised on the internal behaviour of the inference. One could look upon the inference as applying a simple theorem prover.

Note that the inference is only assumed to be primitive with respect to the model of expertise. It is very well possible that such a primitive inference is realised in the actual system through a complex computational technique.

In the KADS model of expertise we use the following terms to denote the various aspects of a primitive inference:

Knowledge source The entity that carries out an action in a primitive inference step is called a *knowledge source*[6]. A knowledge source performs an action that operates on some input data and has the capability of producing a new piece of information ("knowledge") as its output. During this process it uses domain knowledge. The name of the knowledge source is supposed to be indicative of the type of action that it carries out.

Meta-class A knowledge source operates on data elements and produces a new data element. We describe those data elements as *meta-classes*. A meta-class description serves a dual purpose:

(i) it acts as a *placeholder* for domain objects, describing the *role* that these objects play in the problem solving process, and
(ii) it points to the *type(s)* of the domain objects that can play this role.

[6]The term "knowledge source" was inspired by Clancey's [Clancey, 1983] use of this term as a process that generates an elementary piece of information. Its intended meaning corresponds only roughly to the meaning of the term in blackboard architectures.

Domain objects can be linked to more that one meta-class. For example, a particular component of an audio system could play the role of a *hypothesis* at one point in time and the role of *solution* at some other instant. The name "meta-class" is inspired by the fact that it provides a "meta" description of objects in a domain "class".[7] An input data element of a knowledge source is referred to as an *input* meta-class; the output as an *output* meta-class. Each meta-class can be input and/or output of more than one knowledge source.

Domain view The *domain view* specifies how particular parts of the domain theory can be used as a "body of knowledge" by the knowledge source.

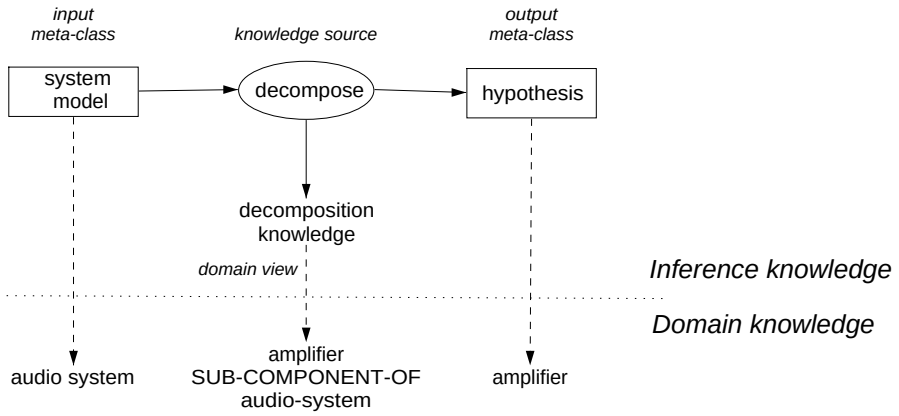

FIGURE 3.4: A primitive inference performing an decomposition action.

Fig. 3.4 describes a primitive inference in the audio domain with example references to domain knowledge. At the inference level a *decomposition* inference is specified. The action that is performed in this inference is the decomposition of a composite model of the audio system into sub-models. *System model* and *hypothesis* are examples of meta-classes. They describe the role that domain objects like `audio-system` and `amplifier` can play in the problem solving process. The *decompose* knowledge source achieves its goal, the generation of a new hypothesis, through the application of decomposition knowledge. The domain view of this inference specifies that tuples of the `SUB-COMPONENT-OF` relation in the domain theory can be used as decomposition knowledge. Fig. 3.4 shows one applicable tuple of this relation.

A somewhat more formal specification of the *decompose* inference is given below. The arrow specifies how inference knowledge maps onto domain knowledge.

knowledge-source *decompose*
  input-meta-class:
    system-model → component
  output-meta-class:
    hypothesis → component
  domain-view:
    decomposition(system-model, hypothesis) → sub-component-of(component, component)

[7] It should not be confused with the meaning of this term in object-oriented systems.

Note that this specification only refers to elements of the schema of the domain theory. Both *system model* and *hypothesis* are place holders of objects of type "component" and describe the role these objects play in the inference process. In this particular example the domain view refers to just one type of knowledge in the domain theory, namely the `SUB-COMPONENT-OF` relation. In principle however, there could be several of these mappings.[8]

There are distinct advantages of separating the domain theory from the way it is viewed and used by the inferences:

- The separation allows multiple use of essentially the same domain knowledge. Imagine for example a knowledge source *aggregate*, that takes as input a set of components and aggregates them into one composite component. This knowledge source could use the same `SUB-COMPONENT-OF` relation, but *view* it differently, namely as aggregation knowledge. Such an inference could very well occur in a system that performs configurations of audio systems.

- Domain knowledge that is used in more that one inference is specified only once. In this way, knowledge redundancy is prevented.

- It provides a dual way to *name*[9] domain knowledge: both use-independent and use-specific. Knowledge engineers tend to give domain knowledge elements names that already reflect their intended use in inferencing and keep changing the names when their usage changes. We would argue that both types of names can be useful and should be known to the system, for example for explanation purposes.

- The scope of the domain theory is often broader than what is required for problem solving. For example, explanatory tasks (in KADS defined in the model of cooperation) often require deeper knowledge than is used during the reasoning process itself.

This is not to say that we claim that a domain theory can in general be defined completely independent of its use in the problem solving process. The scope and the structure of the domain knowledge has to meet the requirements posed by the total set of inferences. In many applications there are interactions between the process of conceptualising a domain and specifying the problem solving process. We are convinced however, that it is useful to *document* them at least separately.

As stated previously, the primitive inference steps form the building blocks for an application problem solver. They define the basic inference actions that the system can perform and the roles the domain objects can play. The combined set of primitive inferences specifies the basic inference capability of the target system. The set of inference steps can be represented graphically in an *inference structure*. The inference structure thus specifies the problem solving *competence* of the target system.

Fig. 3.5 presents such an inference structure for the audio domain. The inferences specify a top-down and systematic approach to find a sub-model of the audio system that behaves inconsistently. The following inferences appear in the inference structure:

[8] We omit here the details of specifying the mapping between a domain view and a domain theory. See for a more detailed discussion [Schreiber *et al.*, 1989b].

[9] We would argue that the whole activity of knowledge acquisition is in fact for a large part a matter of giving (meaningful) names.

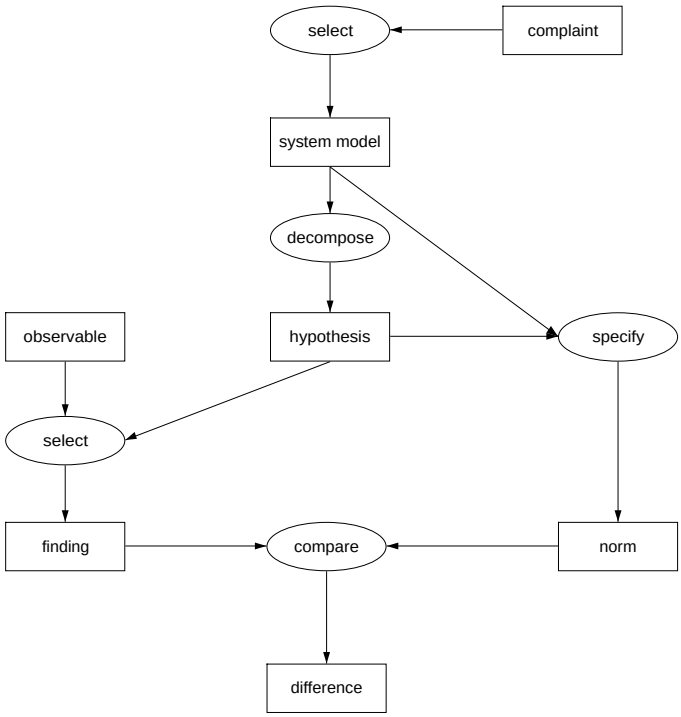

FIGURE 3.5: An inference structure for diagnosing faults in an audio system. Rectangles represent meta-classes; ovals represent knowledge sources. Arrows are used to indicate input-output dependencies.

- A selection of a a (sub-part) of the audio system (*system model*) on the basis of a *complaint*.
- A decomposition of some part of the system into a number of sub-components that play the role of *hypothesis*.
- A prediction of a *norm*-value for a hypothesis. The norm is a value of a test, that is consistent with the normal state of the hypothesis.
- A specification of an observable, for which a value is to be obtained (the *finding*).
- A comparison of the observed *finding* and the predicted *norm*.

The inference structure defines the vocabulary and dependencies for control[10], but *not* the control itself. This latter type of knowledge is specified as task knowledge.

3.4.3 Task knowledge The third category contains knowledge about how elementary inferences can be combined to achieve a certain goal. The prime knowledge type in this category is the *task*. Tasks can achieve a particular *goal*. The relations between tasks and goals are in principle many-to-many. Task knowledge is usually characterised by a vocabulary of control terms, for instance indicating that a finding has been processed or a hypothesis has been verified.

[10] We use the term *control* here to refer to the process of controlling the execution of knowledge sources. We are not referring to more detailed, symbol-level forms of control such as search control in the application of a computational technique. See [Schreiber *et al.*, 1991a] for a more elaborate discussion on these different types of control.

Tasks represent fixed strategies for achieving problem solving goals. Several researchers [Clancey, 1985a; Gruber, 1989] have pointed out that task knowledge is an important element of expertise. The competence model of the diagnostic strategy of NEOMYCIN [Clancey, 1985a] is an example of what we call task knowledge. Clancey describes the sub-tasks of this strategy via meta-rules. The main difference approach between his approach and our approach is that he refers directly in these meta-rules to the domain knowledge. In KADS, tasks only refer to inferences and not explicitly to domain knowledge.

We use the following constructs to describe task knowledge:

Task A task is a composite problem solving action. It implies a decomposition into sub-tasks. The application of the task to a particular (sub-)problem results in the achievement of a goal.

Control terms The control vocabulary used. A control term is nothing more than a convenient label for a set of meta-class elements. The label represents a term used in the control of problem solving, e.g. "differential" or "focus". Each control term is defined through the specification of a mapping of this term onto sets of meta-class elements (e.g. the differential is the set of all active hypotheses).

Task structure A decomposition into sub-tasks and a specification of the control dependencies between these sub-tasks.[11] The decomposition can involve three types of sub-tasks:

1. Primitive problem solving tasks: inferences specified in the inference layer.
2. Composite problem solving tasks: a task specified in the task layer. In principle, this could be a recursive invocation of the same task.
3. Transfer tasks: tasks that require interaction with an external agent, usually the user.

The dependencies between the sub-tasks are described as a structured-English procedure such as used in conventional software engineering [DeMarco, 1978], with selection and iteration operators.

The conditions in these procedures always refer to control terms and/or meta-class elements, e.g. "**if** *the differential is not empty* **then** ...".

There is interaction between the task knowledge in the model of expertise on the one hand and the model of cooperation on the other hand with respect to the specification of the transfer tasks. Transfer tasks are more or less specified as a black box in the model of expertise. We distinguish four types of transfer tasks (for more details, see [de Greef & Breuker, 1992]):

1. *Obtain*: the system requests a piece of information from an external agent. The system has the initiative.
2. *Present*: the system presents a piece of information to an external agent. The system has the initiative.

[11] We agree with [Steels, 1990] that "control structure" is a more appropriate term for this type of structure. We stick here to the term "task structure" mainly for historical reasons.

3. *Receive*: the system gets a piece of information from an external agent. The external agent has the initiative.

4. *Provide*: the system provides an external agent with a piece of information. The external agent has the initiative.

An example task-knowledge specification for our audio domain is shown below. It consists of three tasks. The first task is *systematic-diagnosis*. The goal of this task is to find a sub-system with inconsistent behaviour at the lowest level of aggregation. The task works under the single-fault assumption. On the basis of a complaint, an applicable system model is selected. This selection task corresponds to the knowledge source *select* specified in the inference layer. Subsequently, hypotheses in the differential are generated through the *generate-hypotheses* sub-task. In the sub-task *test-hypotheses* these hypotheses are then tested to find an inconsistent sub-system. This hypothesis then becomes the focus for further exploration. The generate-and-test process is repeated, until no new hypotheses are generated (i.e. the differential is empty).

task *systematic-diagnosis*
  goal:
    find the smallest component with inconsistent behaviour, if one.
  input:
    complaint
  output:
    inconsistent-sub-system: sub-part of the system with
      inconsistent behaviour
  control-terms:
    differential: set of currently active hypotheses
  task-structure:
    systematic-diagnosis(complaint → inconsistent-sub-system) =
      select(complaint → system-model)
      generate-hypotheses(system-model → differential)
      REPEAT
        test-hypotheses(differential → inconsistent-sub-system)
        generate-hypotheses(inconsistent-sub-system → differential)
      UNTIL differential = ∅

For readability purposes, the names of knowledge sources are italicised in the task structure. The arrows in the task structure describe the relation between input and output of the sub-task. Note that all arguments of tasks and conditions are either explicitly declared (*differential*) or are meta-class names.

The task *generate-hypotheses* is a very simple task. It just executes the *decompose* knowledge source until it produces no more solutions.

task *generate-hypotheses*
  goal:
    generate new set of hypotheses through decomposition
  input:
    system model
  output:
    hypothesis-set: set of newly generated hypotheses
  control-terms:
    hypothesis: device component

task-structure:
generate(system-model → hypothesis-set) =
REPEAT
decompose(system-model → hypothesis)
hypothesis-set := hypothesis ∪ hypothesis-set
UNTIL no more solutions of *decompose*

The task *test-hypotheses* tests the hypotheses in the differential sequentially until an inconsistency is found (*difference* = *true*). Testing is done through a kind of experimental validation: a norm value is predicted and this value is compared with what is actually observed. *Obtain(observable, finding)* is an example of a transfer task, that starts an interaction with the user to obtain a test value. How the transfer task is carried out, should be specified in the model of cooperation.

task *test-hypotheses*
goal:
test whether a hypothesis in the differential behaves inconsistently
input:
differential
output:
hypothesis: element of the differential with inconsistent behaviour
control-terms: -
task-structure:
test(differential → hypothesis) =
DO FOR EACH hypothesis ∈ differential
specify(hypothesis → norm)
specify(hypothesis → observable)
obtain(observable → finding)
compare(norm + finding → difference)
UNTIL difference = true

If one abstracts from the control relations between sub-tasks and assumes a fixed task decomposition, the set of task structures can be represented graphically as a tree. The tree for systematic diagnosis is shown in Fig. 3.6. Such a decomposition of a task assigned to the system is in fact a further refinement of the decomposition specified in the task model (see Sec. 3.3).

3.4.4 Strategic knowledge The fourth category of knowledge is the *strategic knowledge*.[12] Strategic knowledge determines what goals are relevant to solve a particular problem. How each goal is achieved is determined by the task knowledge. Strategic knowledge will also have to deal with situations where the afore-mentioned knowledge categories fail to produce a partial solution. For example, the problem-solving process may reach an impasse because information is not available or because contradictory information arises. In such cases the strategic reasoning should suggest new lines of approach or attempt to introduce new information *e.g.*, through assumptions (cf. [Jansweijer, 1988; Jansweijer *et al.*, 1989]).

Strategic knowledge concerns, among other things, the dynamic planning of task execution. However, most systems developed with the KADS approach used only fixed task

[12] Gruber uses the term "strategic knowledge" in a different way [Gruber, 1989]. His strategic knowledge is in many aspects similar to the task knowledge in KADS.

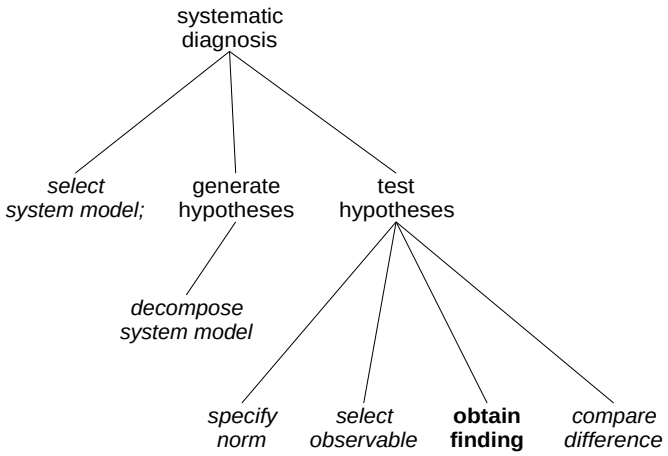

FIGURE 3.6: Task tree of systematic diagnosis. The leaves of such a tree are either knowledge sources or transfer tasks.

decompositions and had little or no strategic knowledge. In our opinion, this does not mean that strategic knowledge is unimportant or superfluous. When knowledge engineers have to construct more complex and flexible knowledge-based systems than presently is usually the case, we think a much more detailed exploration of strategic knowledge will be necessary. We have recently started to work on an ESPRIT project named REFLECT where the central topic is the exploration of strategic knowledge. Apart from dynamic planning, strategic knowledge can also enable a system to answer questions such as "Can I solve this problem?" [Voß *et al.*, 1990]. For the moment however, the study of the nature of strategic knowledge remains mainly a research topic.

3.4.5 Synopsis of the model of expertise The four knowledge categories (domain, inference, task and strategic knowledge) can be viewed as four levels with meta-like relations in the sense that each successive level interprets the description at the lower level. In Fig. 3.7 these four levels and their interrelations are summarised.

The four-layer framework is a structured but informal framework. This means that the specifications are sometimes not as precise as one might want them to be and thus may be interpreted in more than one way. This has led to research aimed at defining a formal framework for representing models of expertise [van Harmelen & Balder, 1992; Wetter, 1990] The price paid for a greater amount of precision in formal specifications is however a reduction in conceptual clarity. In our view, there is a place for both informal and formal representations in the knowledge engineering process. The use of both informal and formal model representations is a major topic of research in the KADS-II project.

The four-layer framework for knowledge modelling has been successfully used as a basis for structured acquisition and description of knowledge at an intermediate level between the expertise data obtained from experts, text books, etcetera and the knowledge representation in an implemented system [de Greef & Breuker, 1985]. From a knowledge-level viewpoint, the present four-layer model captures knowledge categories that are quite similar to those encountered in other models in the literature. However, differences in opinion exist about where to situate particular types of knowledge. This point will be

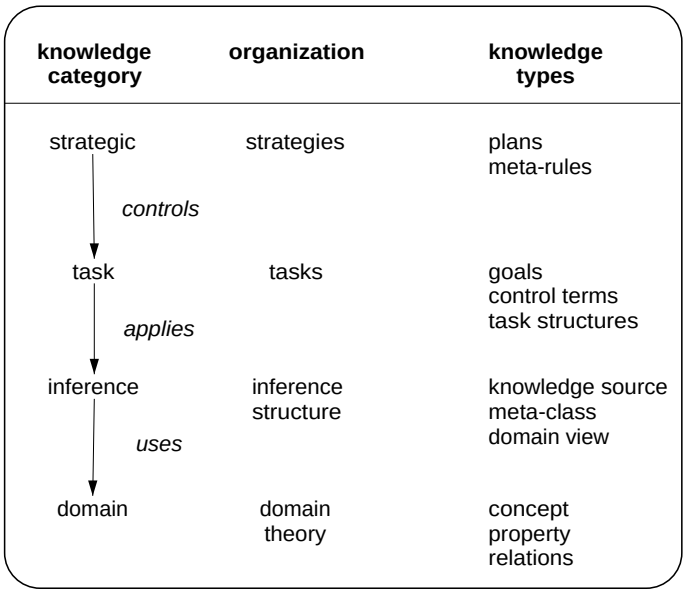

FIGURE 3.7: Synopsis of the KADS Four-Layer Model

discussed in more detail in Sec. 3.7.

3.5 Principle 3: Reusable Model Elements

There are several ways in which models of expertise can be used to support the knowledge acquisition process. A potentially powerful approach is to *reuse* (structures of) model elements. When one models a particular application, it is usually already intuitively clear that large parts of the model are not specific for this application, but reoccur in other domains and/or tasks. KADS (as do most other approaches to knowledge modelling) makes use of this observation by providing a knowledge engineer with predefined sets of model elements. These libraries can be of great help to the knowledge engineer. They provide her with ready-made building blocks and prevent her from "re-inventing the wheel" each time a new system has to be built. In fact, we believe that these libraries are a *conditio sine qua non* for improving the state of the art in knowledge engineering.

In this section, two ways of reusing elements of the model of expertise are discussed: (i) *typologies* of primitive inference actions (knowledge sources) and (ii) *interpretation models*. In principle however, the reusability principle holds for all models in the KBS construction process.

3.5.1 Typologies of knowledge sources In [Breuker *et al.*, 1987] we have defined a tentative typology of primitive problem solving actions (knowledge sources) which has been the basis of a considerable amount of models. The typology is based on the possible operations one can perform on the epistemological primitives defined in KL-ONE [Brachman & Schmolze, 1985]. This set of primitives consists of:

- concept
- attribute (of concept)

- value (of attribute)
- instance (of concept)
- set (of concepts)
- structure (of concepts)

In the typology of inferences we view these primitives not as data-structures but as epistemological categories. Their actual representation in a system may be quite different (e.g. in terms of logical predicates rather than KL-ONE like constructs).

Operation type	*Knowledge source*	*Arguments*
Generate concept/instance	instantiate	concept → instance
	classify	instance → concept
	generalise	set of instances → concept
	abstract	concept → concept
	specify	concept → concept
	select	set → concept
Change concept	assign-value	attribute → attribute-value
	compute	structure → attribute-value
Differentiating values/structures	compare	value + value → value
	match	structure + structure → structure
Structure manipulation	assemble	set of instances → structure
	decompose	structure → set of instances
	transform	structure → structure

TABLE 3.2: A Typology of Knowledge Sources

Table 3.2 gives an overview of the typology of knowledge sources used in KADS. The inferences are grouped on the basis of the type of operation that is carried out by the knowledge source: *generate concept/instance*, *change concept*, *differentiate values/structures* and *manipulate structures*. A detailed description of the inferences mentioned in Table 3.2 is given in [Breuker & Wielinga, 1989].

Although this typology has been a useful aid in many analyses of expertise, it has a number of important limitations:

- The selected set in Table 3.2 is in a sense arbitrary. For example, we could have added other operations on sets such as *join*, *union*, or *merge*.
- The ontology on which the typology is based is of a very general nature and hence weak. The operations are defined more or less independent of tasks and/or domains. Often, it is difficult for the knowledge engineer to identify how an inference in a particular application task must be interpreted.
- A more serious limitation is that some inferences cannot be adequately classified because they require another ontological framework. For example, operations on causal relations such as abduction and differentiation cannot be represented in a natural way.

We consider the study of more adequate taxonomies of inferences to be a major research issue. Potentially, taxonomies are very powerful aids for the knowledge engineer. In a

new research project (KADS-II) we are exploring the possibility of describing taxonomies that are specific for classes of application domains such as technical diagnosis. These taxonomies will be based on a much more task-specific ontology.

It is interesting to see that from a different angle the "Firefighter" project [Klinker *et al.*, 1991] is aiming at similar results. An important goal of this project is to look at what they call *mechanisms* that are used in various applications, detect commonalities between these mechanisms, and construct a library of mechanisms that can be reused in other applications. These mechanisms appear to have the same grain size as the knowledge sources in KADS. The main difference is that mechanisms have a computational flavour.

3.5.2 Interpretation models Typologies of elements of a model of expertise, such as a typology of knowledge sources, represent a first step into the direction of reusability. A further step would be to supply *partial models* of expertise such as models without all the detailed domain knowledge filled in. Such partial models can be used by the knowledge engineer as a template for a new domain and thus support top-down knowledge acquisition. In KADS such models are called *interpretation models*, because they guide the interpretation of verbal data obtained from the expert.

The KADS interpretation models are models of expertise *with an empty domain layer*. Interpretation models describe typical *inference* knowledge and *task* knowledge for a particular task. As these descriptions are phrased in domain-independent terminology, they are prime candidates for reuse in other domains. For example, the inference and task description of the audio domain could very well be applied to another domain where some device is being diagnosed. In [Breuker *et al.*, 1987] interpretation models for a large number of tasks are presented. One of these is the model for systematic diagnosis as presented here.

Example interpretation model Another model in this library is that of the *monitoring* task. This model has been used in applications ranging from process control [Schrijnen & Wagenaar, 1988] to software project management [de Jong *et al.*, 1988] It is also interesting because it illustrates how different tasks can apply the same set of inferences in different ways.

The inference structure of the interpretation model for the monitoring task (shown in Fig. 3.8) depicts the following inferences:

- The selection of a system parameter.
- The instantiation of the normal value of the parameter (the norm).
- The selection of a corresponding observable.
- A comparison of observed and expected values leading to a difference description.
- A classification of the difference into a discrepancy class, e.g. *minor* or *major* disturbance. Often, data from previous monitoring cycles are used in this inference.

Two typical tasks (fixed strategies) were identified for monitoring. One could view them as two different ways of "going through" the inference structure of Fig. 3.8.

The first task, *model driven monitoring*, describes a monitoring approach where the system has the initiative. This type of task is usually executed at regular points in time. The system actively acquires new data for some selected set of parameters and then checks whether the observed values differ from the expected ones.

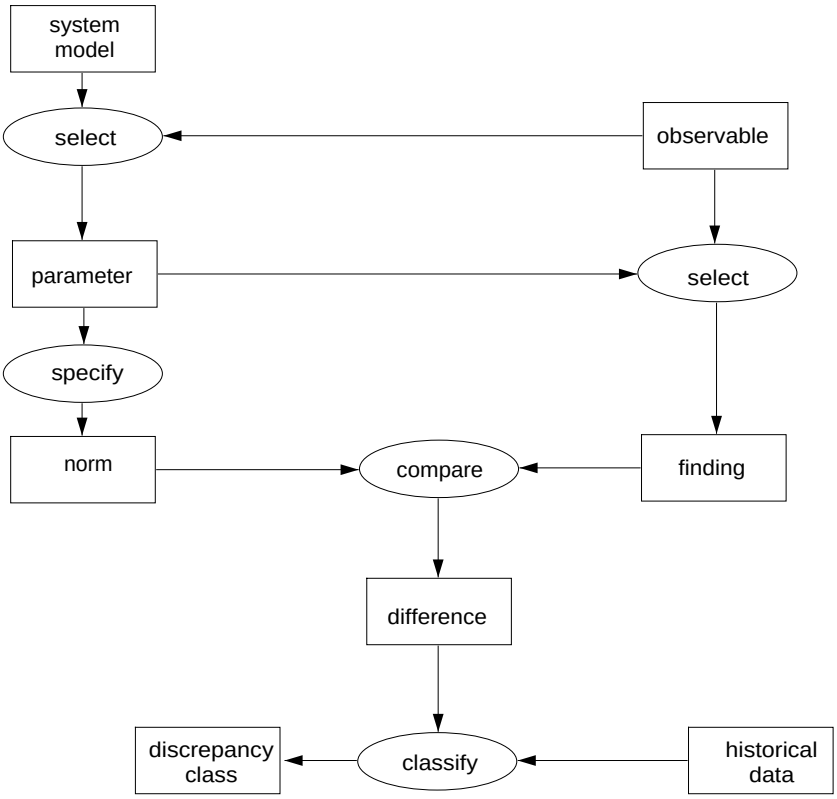

FIGURE 3.8: Inference structure of the interpretation model for *monitoring*

task *model-driven monitoring*
goal:
execute a monitoring cycle in which the system actively acquires new data
input: -
output:
discrepancy
control-terms:
active-parameters:set of parameters
task-structure:
monitor(discrepancy) =
select(system-model, active-parameters)
DO FOR EACH parameter ∈ active-parameters
specify(parameter → norm)
select(parameter → observable)
obtain(observable → finding)
compare(norm + finding → difference)
classify(difference + historical-data → discrepancy)

The second task, *data-driven monitoring*, is initiated through incoming data. It contains a *receive* statement representing a transfer task in which an external agent (a human user or another system) has the initiative(see Sec. 3.4.3). The values received are checked against expected values for the observables concerned. Resulting differences are subsequently classified in discrepancy classes.

task *data-driven-monitoring*

goal:
execute a monitoring cycle when a new value of an observable
is received by the system
input: -
output:
discrepancy
task-structure:
monitor(discrepancy) =
receive(observable-set → finding)
DO FOR EACH observable ∈ observable-set
select(observable + system-model → parameter)
specify(parameter → norm)
compare(norm + finding → difference)
classify(difference + historical-data → discrepancy)

Selecting an interpretation model from the library The library of interpretation models consists of a number of models that can be used to describe the reasoning process in various applications. The knowledge engineer is guided in deciding which interpretation model to choose for a particular application through a decision tree. Part of this tree is shown in Fig. 3.9. indexinterpretation model library, model selection

The decision tree is based on a taxonomy of task types. This taxonomy is a modified and extended version of the Clancey's description of problem types [Clancey, 1985b] which in turn was derived from [Hayes-Roth *et al.*, 1983; p. 14]. The decision points in this tree concern features of the solution space, the problem space and the required domain knowledge types.

The first decision point concerns the availability of information about the structure of the system involved in a task. The term "system" refers here to the central entity in the application domain, e.g. the audio system in the audio domain, the patient in a medical domain, the device in a technical domain, etc. Other decision points concern for example the type of solution (state, category, types of categories, etc.) and the nature of the domain knowledge (fault-model or correct-model of the system). The leaves of the decision tree are associated with one or more interpretation models that specify typical inference and task knowledge for modelling this task. For example, the interpretation model for monitoring presented earlier is associated with the *monitoring* task in Fig. 3.9. This model is chosen if (i) the structure of the system is given, (ii) the solution is a category and (iii) this solution category is not a fault category nor a decision class, but a simple discrepancy between observed and expected behaviour.

It should be noted that in many real-life applications the task is a compound one: it consists of several basic tasks. For example, in the model of expertise for the audio domain, we focused only on the diagnostic sub-task. In actual practice the repair /remedy task also needs to be addressed. This may result in a combination of (parts of) two or more interpretation models. An example of this process of combining is described in [Hayward, 1987].

A number of researchers have developed knowledge acquisition tools that are based on the notion of a generic model of the problem solving task. For example, ROGET [Bennet, 1985], MOLE [Eshelman *et al.*, 1988] and BURN [Klinker *et al.*, 1991] are all systems that drive the knowledge acquisition dialogue with an expert through a strong model of the

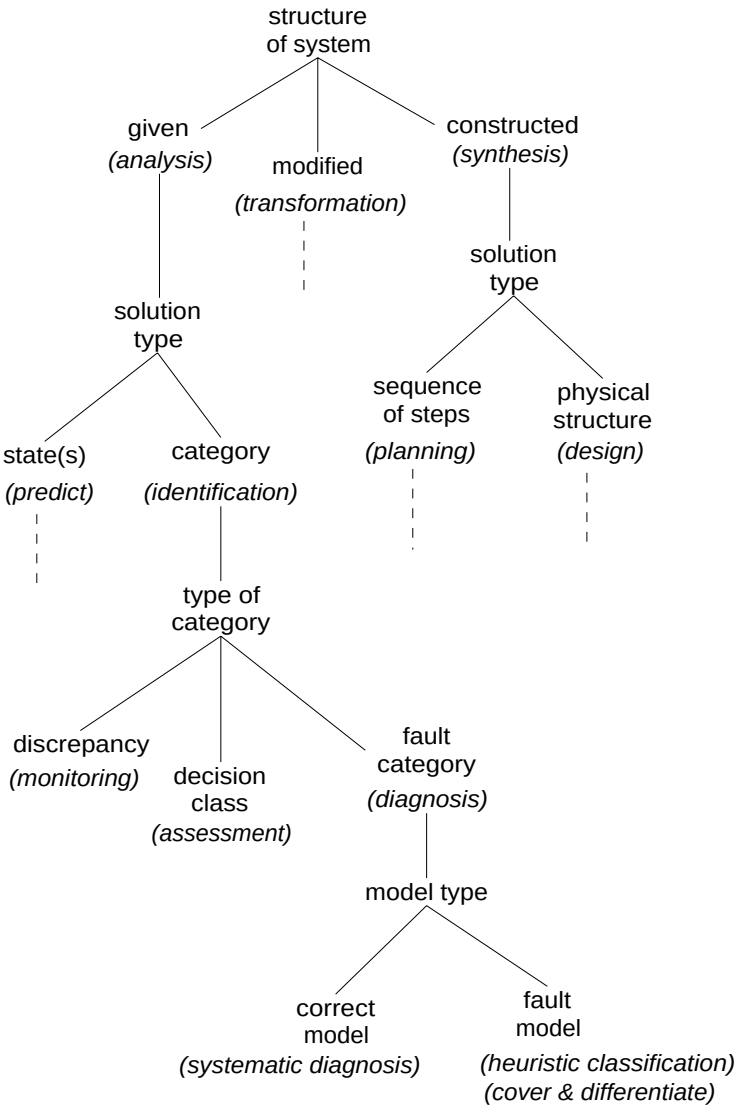

FIGURE 3.9: Partial decision tree of interpretation models

problem solving process. This model prescribes what domain knowledge is needed to build an actual expert system. In OPAL [Musen *et al.*, 1987] this approach is taken one step further. The conceptual model in OPAL is not just a model of the problem solving process (i.e. the upper three layers in the KADS framework) but also contains templates of the domain knowledge needed. As a consequence OPAL can present the expert with detailed forms that he or she can fill in with the details of an application domain. Although this approach is very powerful indeed, it has limitations in scope and applicability.

3.6 The Knowledge Acquisition Process

The description of the various models can be seen as the *product* of KBS construction, With respect to the *process* of KBS construction, KADS provides two ways of support:

(i) a description of phases, activities, and techniques for knowledge engineering, and (ii) computerised support tools. Both are briefly discussed in this section.

3.6.1 Phases, activities and techniques A *phase* represents a typical stage in the knowledge engineering process. A phase is related to a number of *activities*, that are usually carried out in this phase. One particular activity can occur in more than one phase. For example, "data collection" can occur in many different phases. The activities are the central entities in the process view on knowledge engineering. An activity is a piece of work that has to be carried out by the knowledge engineer. An activity produces a result. This result constitutes either directly a part of one or more models or it represents some intermediate product, that is used by other activities. An activity applies one or more *techniques*. For example, a "time estimation" activity can be carried out with an extrapolation technique. Life cycle models predefine particular phases, activities, techniques and products and also their interrelations. Life cycle models for KADS have been described in [Barthèlemy *et al.*, 1987; Taylor *et al.*, 1989] We limit the discussion here to those activities that are related to building a first model of expertise. We distinguish two phases in building such a first model of expertise: *knowledge identification* and *knowledge modelling*.

Knowledge identification is more or less a preparation phase before the actual construction of the model of expertise can begin. Relevant activities for this phase are shown in Fig. 3.10 together with applicable products and techniques. The results include a task model and also intermediate products that are used by activities in other phases, especially the knowledge modelling phase. Example activities are glossary and lexicon construction. A glossary and a lexicon provide a way of documenting the application domain without committing to any formal conceptualisation.

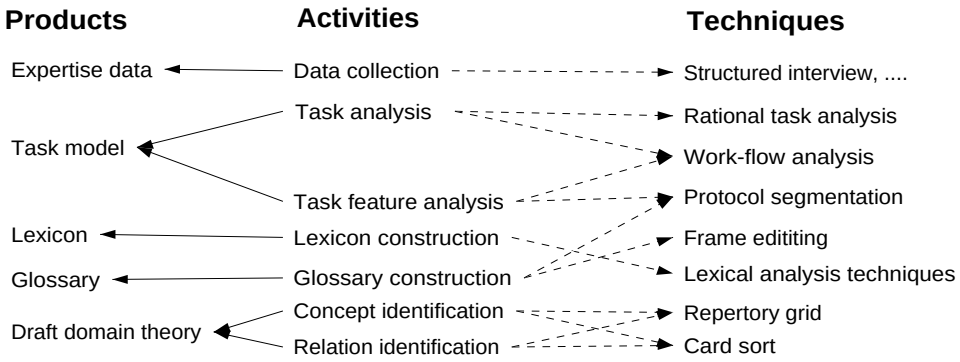

FIGURE 3.10: Knowledge identification activities and related products and techniques

In the knowledge modelling phase the knowledge engineer constructs a model of expertise. Fig. 3.11 summarises the main activities relevant for knowledge modelling. A crucial one is the selection of an interpretation model. This activity is supported through the decision tree discussed in Sec. 3.5. The model validation and model differentiation activities often make use of protocol analysis techniques. Model validation can also be supported by transformation of the model into a functional prototype. This prototype can be seen as a simulator of the problem solving aspects of the artefact. The KADS-II project is currently working on a tool to support this type of prototyping. Other activities

deal with the definition of the domain conceptualisation. In KADS we usually assume that in the resulting model the domain theory can be a partial one , but with a fully defined domain schema. Refinement and debugging of the domain theory is performed in a later phase, possibly with the use of automised techniques.

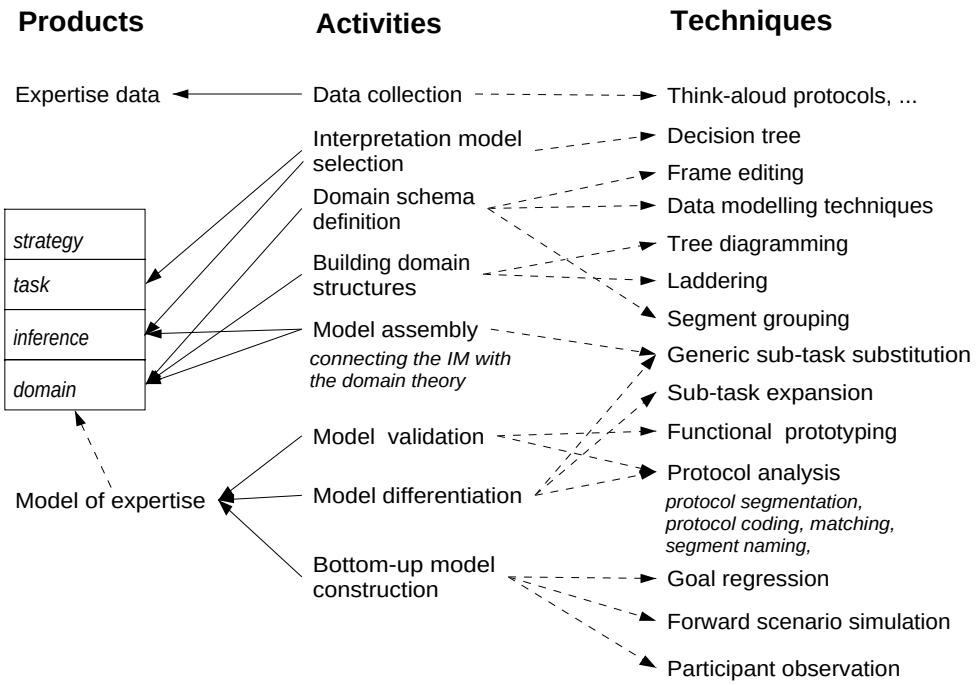

FIGURE 3.11: Knowledge modelling activities and related products and techniques

3.6.2 Tools Within the KADS project the Shelley workbench was developed to support activities in the KBS life cycle. Shelley contains an integrated set of computerised support tools. The user of the workbench is the knowledge engineer. Example support tools in Shelley for the knowledge modelling phase are:

- A *domain text editor*: a tool that allows management and analysis of protocols or other texts, for example through the creation of text fragments of a particular type. These fragments can subsequently be linked to other objects, such as elements of the model of expertise.
- A *concept editor* to create concepts and corresponding attributes.
- An *interpretation model library* from which models can be selected.
- An *inference structure editor* that supports the construction of the inference layer of the model of expertise.

Fig. 3.12 shows an example of the use of Shelley in the audio domain. The knowledge engineer has selected the interpretation model of systematic diagnosis from the library and inserted it into the inference structure editor. A think-aloud protocol is being analysed.

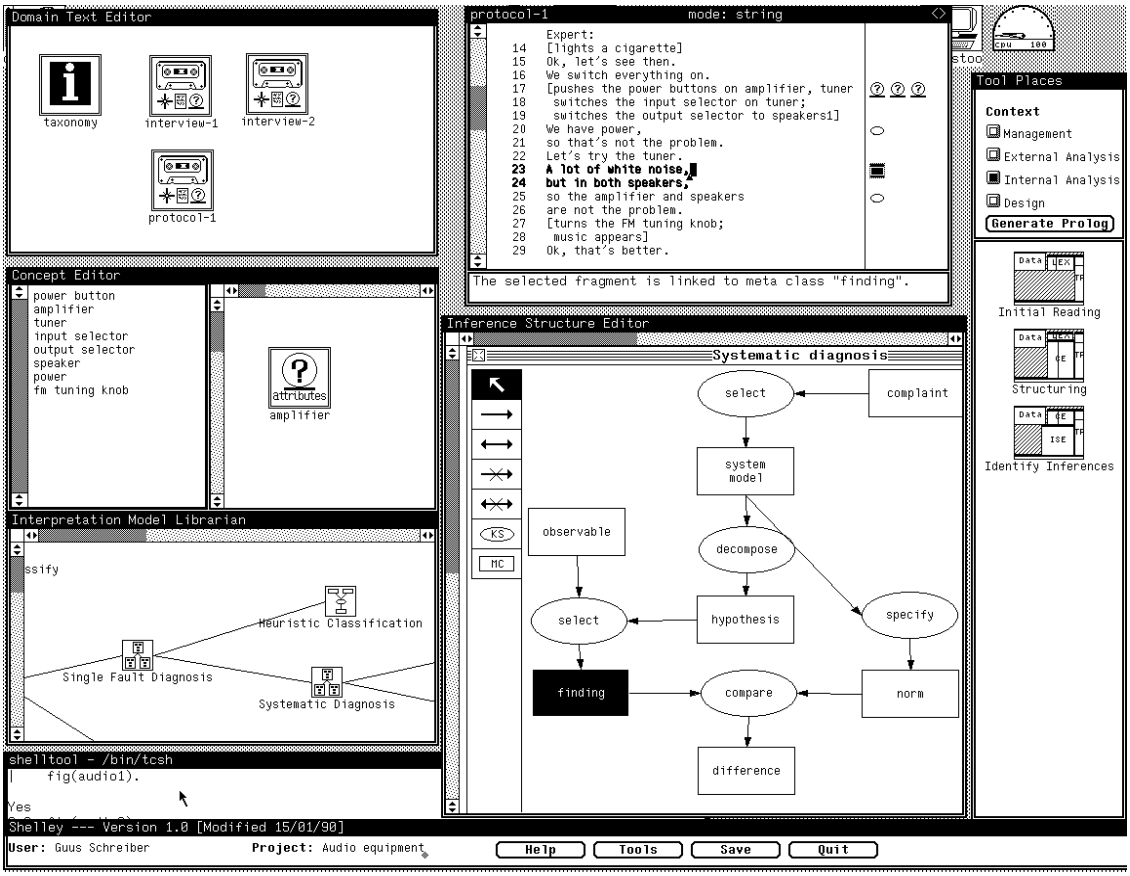

FIGURE 3.12: Example session with the Shelley workbench

The link of a particular fragment of the protocol to a meta-class in the inference structure editor is shown.

The Shelley workbench is described in more detail in [Anjewierden *et al.*, 1992]

3.7 Relation to Other Approaches

We make no claim that all ideas underlying KADS are new. On the contrary, work of other researchers has heavily influenced the work on KADS. In this section we discuss a number of these approaches and relate them to the KADS approach.

Brachman proposed five levels for describing knowledge [Brachman, 1979]: the linguistic, the conceptual, the epistemological, the logical and the implementational level. Brachman and also Clancey [Clancey, 1983] showed that the epistemological level of Brachman is the "missing" level in the description of knowledge-based systems. We interpret Newell's knowledge level as a combined description of Brachman's conceptual and epistemological level. In the KADS model of expertise the domain knowledge roughly corresponds to the conceptual level and the three other categories to the epistemological level. The KADS design description (Newell's symbol level) corresponds to the logical level.

The work of Clancey has had a large impact on KADS. [Clancey, 1985b] introduced the

notion of an inference structure in the description of the model of heuristic classification (HC). In the work on NEOMYCIN [Clancey, 1985a] a similar type of task decomposition is found as is used in the task layer in KADS. The main difference is that there is no explicit relation between the tasks in NEOMYCIN and the inferences in the HC model. These tasks refer directly to the domain knowledge, whereas the tasks in KADS reference the domain knowledge only indirectly via primitive inferences.

In the approach taken at Ohio State University [Bylander & Chandrasekaran, 1988; Chandrasekaran, 1988] the implementation environment consists of so called "generic tasks". A generic task (GT) is a combination of a problem (e.g. classification) and a problem solving method (e.g. hierarchical classification) with particular knowledge and inference requirements. GT's can perform quite general information-processing tasks. The assumption is that by combining a relatively small set of GT's one can solve a large number of problems. The problem solving methods in the GT approach have a somewhat smaller grain size than the interpretation models in KADS.

In the approach taken at DEC [McDermott, 1988; Marcus & McDermott, 1989; Eshelman *et al.*, 1988] a number of systems were built that provide an operationalisation of a particular problem solving method, such as ' 'propose & revise" and "cover & differentiate". The terminology used to describe these methods is such that during knowledge acquisition the expert can be prompted for domain knowledge in a high-level, method-specific, language, e.g. "`What are symptoms that the system should be able to explain?`". The problem solving methods have a similar grain size as the KADS interpretation models. More recently [Klinker *et al.*, 1991], the emphasis in this approach has shifted to the construction of an integrated environment in which the knowledge engineer can *configure* such single-task knowledge acquisition systems from a set of predefined mechanisms. As remarked in Sec. 3.5.1, the research on a typology of mechanisms is very close to aims in KADS.

In the PROTEGE [Musen, 1989] approach the problem is addressed that experts find it difficult to enter knowledge in a method-specific format. In this approach two steps are distinguished in building ONCOCIN-like systems: the knowledge engineer uses PROTEGE to specify the required domain knowledge in method-specific terms; PROTEGE then generates a knowledge acquisition tool called p-OPAL that enables the expert to enter knowledge in domain-specific terms. This dual way of naming domain knowledge is similar to the approach advocated in KADS. The PROTEGE system presupposes a single-task model, based on the skeletal planning method of ONCOCIN [Shortliffe *et al.*, 1981]

All models used in these last three approaches are hard-wired to particular computational constructs. As stated earlier, compared to the KADS approach this is both an advantage and a disadvantage.

The "Components of Expertise" (CoE) approach [Steels, 1990] is in many aspects similar to KADS. The main differences with KADS are the dynamic view on task decomposition based on task features and the absence of an explicit description of inference knowledge such as meta-classes. A dedicated computational framework has been developed for CoE models [Vanwelkenhuysen & Rademakers, 1990]. Research aiming at a synthesis of KADS and CoE is in progress within the KADS-II project.

The "Ontological Analysis" approach [Alexander *et al.*, 1988] describes knowledge in three categories: (i) the static ontology describing the primitive objects, properties and relations, (ii) the dynamic ontology describing the state space of the problem solver

and the actions that can make transitions in this space, and (iii) the epistemic ontology describing methods that control the use of knowledge of the first two categories. These three categories resemble closely the domain, inference and task knowledge in KADS. The formalisms used in Ontological Analysis are based on algebraic specification languages.

Although terminology is different, a common view appears to emerge based on the idea that different types of knowledge constitute the knowledge level and that these different types of knowledge play different roles in the reasoning process and have inherently different structuring principles. One salient characteristic is that all approaches distinguish between structural domain knowledge and control knowledge. In addition, various kinds of control knowledge are distinguished, like global control of how to go about the task as a whole, and local control knowledge specifying how and/or when to carry out certain individual actions.

There are also relations between KADS and conventional software engineering approaches. The introduction of multiple models was inspired by work of [DeMarco, 1982] As pointed out in Sec. 3.4.1, issues concerning modelling of domain knowledge are quite closely related to research in semantic database modelling. Software engineering techniques are used in KADS, e.g. a form of data-flow diagrams (for inference structures) and structured English (for task structures). Life-cycle models using a water-fall approach [Barthèlemy *et al.*, 1987] and a spiral model approach [Taylor *et al.*, 1989] have been defined in KADS. The relations with conventional software engineering are discussed in more detail in Ch. 8.

3.8 Experiences

The KADS approach has been (and is being) used in some 40 to 50 KBS projects. Not all these projects used "pure" KADS. The core activities of Bolesian Systems, a Dutch company, are teaching and applying an earlier version of KADS under the name SKE (Structured Knowledge Engineering). Other companies, such as Arthur Andersen Consulting and commercial partners in the KADS-I project, have incorporated KADS into their own methodology.

Within the KADS-I project the approach has been tested in a number of experiments in domains such as commercial wine making [Wielinga & Breuker, 1984], statistical consultancy [de Greef & Breuker, 1985; de Greef *et al.*, 1988b], the integration qualitative reasoning approaches [Bredeweg & Wielinga, 1988], network management [Krickhahn *et al.*, 1988; Readdie & Innes, 1987], mould configuration [Barthèlemy *et al.*, 1988], mixer configuration [Wielemaker & Billault, 1988], technical diagnosis [Wright *et al.*, 1988], insurance [Brunet & Toussaint, 1990], and credit card fraud detection [Porter, 1992; Killin, 1992], Other applications include re-engineering of ONCOCIN [Linster & Musen, 1992], process control [Schrijnen & Wagenaar, 1988], chemical equipment [Schachter & Wermser, 1988], room planning [Karbach *et al.*, 1989], social security [de Hoog, 1989], software project management [de Jong *et al.*, 1988], diagnosis of movement disorders [Winkels *et al.*, 1989], and paint selection [van der Spek *et al.*, 1990]. The last two systems and the credit card system have been in operational use for some time.

A recent publication for the commercial AI community [Harmon, 1991] commented that "before KADS, most of the methodologies were vague prescriptions rather than systematic

step-by-step models for large scale systems development efforts". On the basis of the success of KADS-I, the CEC has decided to fund a second ESPRIT project (KADS-II) with the aim to arrive at a *de facto* European standard for KBS development.

This is not to say that we think that the KADS approach has no deficiencies: on the contrary. It is clear that a group of KADS users finds certain aspects of KADS attractive, but it is also recognised that there are many weaknesses in current KADS. The first KADS user meeting [Ueberreiter & Voß, 1991] in which some forty, mainly German, KADS users participated, provided a good overview of the strong and weak points of KADS. Among the strong points are:

- The distinction between various models, especially the distinction between the model of expertise and the design.
- The framework for modelling expertise. Especially the inference structures are mentioned by many people as an intuitively appealing way of describing the reasoning process and as a communication vehicle with domain experts.
- The library of interpretation models. Although this library is far from complete, it has still provided useful starting points for many applications.

The list of weaknesses is considerably longer . A selection:

- The vocabulary in the four-layer framework for describing domain knowledge and task knowledge is not expressive enough.
- The typology of knowledge sources is too general. The precise meaning of the knowledge sources is ambiguous.
- The library of interpretation models is incomplete and needs serious revision. For example, coverage of synthetic tasks is marginal.
- KADS does not provide enough support for operationalising conceptual models.
- KADS gives you a vocabulary, but it provides little support for the modelling process.

In short, the experiences show that the KADS approach has some interesting and attractive features, but that it still needs a lot of work before it can really be considered a "comprehensive methodology". In addition, controlled validation studies are necessary to show that KADS actually provides advantages compared to other approaches. The work of [Linster & Musen, 1992] can be seen as a step in this direction.

3.9 Future Developments & Conclusions

In this chapter, we have taken the position that knowledge acquisition is to a large extent a constructive activity: models of several aspects of the task and domain have to be build before implementing a knowledge based system.

Looking at the future of knowledge acquisition from this point of view raises the obvious question of how AI and knowledge based systems themselves can support the various modelling processes. Recent developments in the area of knowledge acquisition tools provide some directions in how this could be done.

Given the modelling approach to knowledge acquisition it is of vital importance that a knowledge engineer has some language in which the various models can be formulated. Such a language is not only important for the knowledge acquisition process itself, but

also for communicating models and comparing models for different tasks. A comparative analysis of the problem-solving methods embodied in KBS's will advance the knowledge acquisition activity from an art to a proper engineering discipline. Although there is currently little consensus on what the ingredients and vocabulary of such a modelling language should be, the various ideas appear to converge. The result of the synthesis of the KADS and the CoE approach, which is currently being pursued and in which ideas from other approaches are also taken into account, may be a starting point for such a language. In our view, it is worthwhile to investigate the different types of knowledge and their relationships also from a more formal point of view. Attempts are being made in this direction (see [van Harmelen & Balder, 1992]). Such a formal account of knowledge models clarifies at least some of the notions that have been used in a rather informal way so far.

If a common language for defining conceptual models of problem solving processes became accepted, it would be of great interest to study the large collection of problem solving models that currently exist. A consolidation and integration of the models in the KADS interpretation model library [Breuker *et al.*, 1987], the generic problem solving models of Chandrasekaran and co-workers [Chandrasekaran, 1988], the models underlying the various model-driven knowledge acquisition tools [McDermott, 1988], and various other models in the literature, could provide the knowledge engineering community with a invaluable tool for knowledge acquisition. Also, such a collection of generic models could be the basis of powerful knowledge acquisition tools that communicate both with experts and with knowledge engineers.

Looking beyond the traditional knowledge engineering paradigm where the knowledge engineer does most of the work, we envisage an important role for knowledge about models in knowledge acquisition tools that integrate traditional knowledge acquisition techniques and automated learning techniques. One of the major problems in this area is that of integrating knowledge of various sources. A system that has knowledge about the kinds of knowledge that it needs to acquire can exercise much more focused control on the acquisition process and hence solve at least part of the integration problems.

Chapter 4

A KADS Domain Description Language

KADS has been critisised for the fact that it does not provide adequate support for modelling the structure of domain knowledge. Existing approaches to data modelling are only partly able to fill this gap. KBS development appears to impose a number of additional requirements on data modelling. Two important ones are (i) the need for *expressions* as an explicit data-modelling notion, and (ii) the need for a range of relation types. In this chapter we present a domain description language (DDL) based on ER modelling and KL-ONE, that provides a number of extensions making the language better suited for KBS data modelling. The language is defined in BNF grammar rules and illustrated with examples. In addition, a graphical representation for the data model is proposed.

This chapter will be published in a collection of articles on KADS. Reference: Schreiber, A. T. (1993). A KADS domain description language. In Schreiber, A. T., Wielinga, B. J., & Breuker, J. A., editors, *KADS: A Principled Approach to Knowledge-Based System Development*. Academic Press, London.

4.1 Introduction

It has been argued that one of the weak points of the KADS methodology is that it provides little to no support for the modelling of domain knowledge. Generally speaking, the KADS documents state that the domain layer consists of "concepts and relations". In most experiments within the KADS-I project P1098 the domain layer description consisted of a glossary of concepts with attribute descriptions and a number of *is-a* and/or *part-of* hierarchies. Originally, KADS advocated the use of KL-ONE as the sole formalism for domain layer modelling. Experiences in several P1098 experiments [de Greef *et al.*, 1987; Wielemaker & Billault, 1988] have however indicated that the KL-ONE primitives are not powerful enough for modelling all relevant structural properties of the domain knowledge.

The KADS domain layer can be viewed as a domain knowledge base containing the data/knowledge types in the domain. The activity of building a domain layer model is in many ways similar to data modelling in conventional software engineering. Efforts have been made to apply information analysis techniques to domain-knowledge modelling. In the next section we summarise three important existing approaches to data modelling and analyse their restrictions. As will be pointed out in Sec. 4.3, the primitives of these techniques are not powerful enough to describe all relevant structural properties which one wants to describe in an application domain. The nature of domain knowledge encountered in knowledge-based applications seems to pose additional requirements on the expressivity

of data modelling techniques. In Sec. 4.3 we formulate a number of these KBS-specific requirements for "data modelling".[1] Based on this analysis we propose in Sec. 4.4 a Domain Description Language (DDL) for describing the KADS domain layer. This DDL contains a number of extensions of the known data modelling languages. We illustrate the use of the DDL with examples in various domains. A full DDL specification of the domain knowledge in the Sisyphus domain (see Ch. 7) can be found in Appendix A.

The purpose of the DDL is to provide a generalisation of various available knowledge representation formalisms. This is in line with the knowledge-level idea: when building a knowledge-level model of an application one should not commit oneself already to a particular representation. In the design stage, the knowledge engineer needs to decide how the DDL descriptions are to be mapped onto or transformed into symbol-level representations. In Sec. 4.5 we discuss the relation between the DDL and some KR formalisms. In addition, we discuss some issues concerning the semantics of the DDL.

4.2 Existing Approaches to Data/Knowledge Modelling

In this section we look at the merits and restrictions of the following three approaches to data/knowledge modelling:

1. Entity-Relationship (ER) Modelling
2. KL-ONE
3. Extensions to ER Modelling: Semantic Database Modelling

4.2.1 Entity-relationship model The Entity Relationship Model [Chen, 1976] has its roots in database design. The Entity Relationship Model provides[2] an organisation of information in entities, relationships, attributes, and values:

1. An *entity* in the model is the representation of some object in our mind. In the original version of the ER model, entities are classified into entity *sets* by means of an associated predicate (and this not by sub-type hierarchies). Entity sets are not necessarily mutually disjoint.

2. A *relationship* in the model is an association between entities. Entities have a *role*, which describes their function in the relationship (compare this with the use of *role* in KL-ONE). This role can be explicit (the role is given a name) or implicit (the role is determined by the order in the relationship). Relationships belong to a set of relationships (i.e. grouped in classes).

3. The other two important concepts used in the Entity Relationship Model are the *attribute* and the *value*. An *attribute* is a function, that associates an entity or a relationship (!) with a *value*. Values are basic objects like a number, a colour or a name. Values belong to one or more value sets.

[1] It could seem more appropriate to use the term "knowledge modelling", but this is commonly used in a broader sense, namely to cover all knowledge types and not just the domain knowledge.

[2] Chen describes four levels on which data are modeled. The focus is here on Chen's level two.

Limitations of the ER model The ER-model is widely accepted as a powerful tool for data modelling. For modelling complex relations like rules and classification hierarchies, the ER-model has a limited set of modelling constructs (see [Koster, 1990]).

4.2.2 KL-ONE KL-ONE constitutes a representation language for the description of "structural" domain knowledge. For a good overview of KL-ONE see [Brachman & Schmolze, 1985] The major characteristics of KL-ONE are:

1. KL-ONE separates *is-a* relations from other types of relations. The *is-a* relation is used to build inheritance lattices of entities (in KL-ONE: *concepts*).

2. All other relations (both entity $\leftrightarrow$ entity and entity $\leftrightarrow$ attribute) are represented as *roles* of an entity.

3. Sub-concepts can limit the values and/or the cardinality of an inherited role (role restriction). Also, a role can be split into two or more sub-roles (role differentiation).

4. KL-ONE distinguishes between concepts and instances. Concepts represent in fact universally quantified axioms concerning a set of instances. A generic classifier can be used to make inferences about instances of concepts using the knowledge provided by the sub-type hierarchy.

5. Structural descriptions are used to describe relations between roles.

Limitations of KL-ONE A limitation of KL-ONE is that relations other than *is-a* relations are described in an indirect way by specifying a role within an entity. For instance, the fact hat an employee works for exactly one department is expressed as follows (simplified for clarity).

entity *employee*
  role works-in
    value-restriction:
      department
    cardinality
      1,1

To represent the reverse relation a role *employee* has to be created in the *department* entity. It is possible to circumvent this problem by creating an entity *employee-department-relation*. This is even necessary, if you want to define a role-attribute, that is dependent on this relation. But introduction of these relations as concepts in a KL-ONE network reduces clarity.

Also, no distinction is made in KL-ONE between roles that symbolise a relation with another entity and attribute roles, i.e. roles, that have some kind of atomic value (integer, string).

4.2.3 Extensions of ER modelling in semantic database modelling The ER-model has given rise to a large number of descendents adding various other constructs. We discuss the extensions proposed in the field of *semantic database modelling*. In semantic data modelling research (for a good overview see [Hull & King, 1987]) the aim is to move beyond the level of the traditional data modeling techniques (the hierarchical, the network and the relational approach). Semantic data models discern (unlike, for example, the basic ER model) between different types of relations. Among the types of relations encountered frequently in these models are *Is-a* relations, set relations and aggregations.

The following set of primitives is viewed by [Hull & King, 1987] as the core of the various semantic data models;

Atomic types Atomic types represent the class of non-aggregate objects. Atomic types can have attributes.

Constructed Types From the atomic types other types can be constructed with the use of two types of operations: *aggregation* and *grouping*.

- Through aggregation a composite object can be constructed from other objects in the database. These objects can themselves be either atomic or constructed. An example could be an address, which is constructed from a street, a number, a city, and a postal code.
- Grouping is used to construct a type which constitutes a set of objects of another type, e.g. the set of addresses.

Attributes An *attribute* is viewed as a *function between types*. Some models make an explicit distinction between single- and multi-valued attributes. In other models attributes can only be single-valued. In this case the grouping relation should be used to construct the multi-valued type. In KL-ONE attributes (roles) represent by default a set of values (i.e. are multi-valued). Here, cardinality constraints are used to specify single-valued attributes. Note that attributes with multiple arguments can be constructed with the aggregation relation (e.g. a function from person to address).

Is-a Relations *Is-a* relations indicate that an object associated with the sub-class can also be associated with the super class. In most models *Is-a* relations are used to allow for inheritance of properties (attributes) from super class to sub-class. *Is-a* relations specify a directed graph in which undirected cycles may occur (multiple inheritance). Some models distinguish two types of *Is-a* relations: a *specialisation* and a *generalisation*.

- A specialisation of C_{super} into one or more classes C_{sub} defines, possibly overlapping, roles that an object of the super type can play: e.g. a person can be a father, a scientist, and a bridge player.[3].

[3] Another example solves an intriguing identity problem that kept bothering me for a long time in prep school, namely that of the Holy Trinity: the super type God can play the role of both the Father, the Son, and the Holy Ghost, as my catholic school teacher explained to me many years ago (GS).

- A generalisation defines that a general term (the super class) can be used to refer to objects belonging to the sub-classes: e.g. the sub-classes *car* and *plane* have the super class *vehicle* as a generalisation. The sub-sets are assumed to be disjoint.

Limitations of Semantic Database Modelling In comparison to ER modelling, semantic database modelling offers a number of additional primitives which could prove to be useful for data modelling in AI. The main limitation lies in the fact that it is still difficult (as will be pointed out in the next section) in these modelling languages to model relations such as cause/effect, time.

4.3 Requirements for a Domain Description Language

Knowledge-based applications impose a number of additional requirements on a data modelling language when compared to conventional systems. A knowledge base may contain various types of knowledge structures: a causal network, taxonomical relations, "rules", etc. Most conventional systems also have some knowledge structures such as salary scales or product prices, but these are usually of a much simpler nature than the structures encountered in a KBS.

In this section we discuss two types of difficulties which frequently occur when modelling the knowledge base, namely (i) the modelling of *expressions*, and (ii) the distinction between relations between *concepts* (in the KL-ONE sense) on the one hand and relations between *instances* on the other hand.

Modelling expressions Taxonomies can often be handled well with existing data modeling techniques, e.g. those offered by KL-ONE or semantic database modelling. (see previous section). Knowledge structures like causal networks and "rules" are a much more difficult subject. A simple example might help to clarify this point.

Suppose we have the following set of rules, each of which denotes an abstraction relation between rough data and a more general feature:

```
IF temperature(patient) > 38.0
  THEN fever(patient) = true
IF diastolic-blood-pressure(patient) > 90
  THEN blood-pressure(patient) = high
IF heart-rate-per-minute(patient) > 100
  THEN heart-rate = high
```

It will be clear that there is a general structure behind these rules which we would like to capture in our data model, when building an application. It allow us to limit the knowledge engineering effort to a specification of this general structure, and leave the actual "filling in" of this structure until a later refinement phase, possibly with the help of automated techniques.

However, it turns out to be quite difficult to use an existing data modelling technique to model such a structure. The major problem that arises is the fact that existing data modelling techniques offer no adequate means for modelling expressions like

temperature > 38.0 in a general way. Often, a knowledge engineer tries to overcome this problem by introducing artificial concepts with very long name labels such as:

```
temperature-higher-than-38.0
```

This long-label approach hides however important structural properties of the domain and is thus sub-optimal. What seems to be needed is the explicit introduction of the notion of *expression* in our modelling language, so that we would be able to say something like:

> A qualitative abstraction relation is a relation between an *expression* about a quantitative attribute and an *expression* about a qualitative attribute (a "feature").

Another frequently occurring example in which expressions play a role are cause/effect relations. Below some simple example causal relations in a medical (heart disease) domain are listed:

```
% The percentage of coronary-artery obliteration is the degree
% to which the diameter of the feeding artery of the heart has
% been reduced, e.g. because of atherosclerosis.

coronary-artery-obliteration = 70% CAUSES angina-pectoris = true
coronary-artery-obliteration = 100% CAUSES myocardial-infarction = true
```

Again, we would like to describe in our data model, that a causal relation is a relation between states and that states are in fact *expressions* about particular system attributes.

Relations between concepts vs. relations between instances Another characteristic of KBS data models is that in these models it occurs much more often that one wants to record statements about concepts (classes, entity sets) and not just about instances of these concepts. Some example relation tuples from the room-planning domain (see Ch. 7) might help to clarify this distinction:

```
room-1 NEXT-TO room-3
room-2 NEXT-TO room-4
.....

head-of-project REQUIRES size(room) = small
head-of-project REQUIRES occupancy(room) = single
head-of-group REQUIRES size(room) = large

....
```

The first set of tuples represent relations between instances (actual rooms). Such relations are typically part of the input data for a KBS. The second set of tuples is in fact a set of universally quantified statements about all instances of a particular concept. For example, the statements about "head of project" could be read as: "all heads of projects need to get some small, single room".

To model the *structure* of such statements as the above-mentioned requirements, requires a distinction between relations between concept instances and relations between concepts.

4.4 Definition of the Domain Description Language

In this section we define a domain modelling language which is based on well-known constructs from existing data-modelling techniques plus some additional constructs that allow one to overcome the specific problems encountered in KBS development. The language supports a *highly structured, but informal* description of the structure of the domain knowledge.

4.4.1 Constructs in the DDL We use the ER-model as the basis of our DDL with a number of extensions provided by semantic data modelling and KL-ONE. The ER model has proved useful in practice and appears to provide a natural and understandable way of modelling domain concepts and relations. In addition, we want to introduce a number of other constructs necessary specifically for KBS applications. It should be noted that most of these additions are in fact specialised ER relations. The main reason for introducing them is to support the domain modelling activity in a natural way.

Proposed additions are:

Sub-type hierarchies Sub-type hierarchies occur in every domain and provide useful abstractions. For the moment we only include in the definitions of these hierarchies the notion of *differentiation* and *restriction* as defined in KL-ONE. We omit for the moment refinements from semantic database modelling such as *specialisation* and *generalisation*. These may be introduced in a later stage, if the need arises.

Grouping The grouping construct is useful. It allows us to explicitly name *sets* of things, e.g. the set of conditions in an abstraction relation.

Aggregation The aggregation construct is useful to model the frequently occurring structural relations in a domain. For example, an audio system is structure consisting of various components with various types of interrelations (e.g causal relations).

Expression Statements like "blood pressure higher than 90" can be modelled by allowing an explicit declaration of expressions when defining relation arguments. This should enable modelling the examples sketched in the previous section.

Relations between various types of constructs The DDL should allow modelling relations between various types of constructs, not just instances.

This leads to the following three groups of modelling constructs in the DDL.

Intensional objects Intensional objects are used describe the structure of the prime ingredients of the data model. Intensional objects usually represent universally quantified statements about a class of (extensional) objects. We distinguish four types of intentional objects:

Concept Used here as a synonym for an entity set or class. The term *concept* is used because it is more common in cognitive science.

Set The set is the grouping construct. A set contains objects of one particular type, e.g. instance, concepts, tuples,, structures, sets.

Relation Relations of various types: between concepts, between instances, between expressions, between sets, etc.

Structure A structure is an aggregate with a number of parts. These parts can be of any type: concepts, instances, relations, other structures, sets, expressions.

A concept is in the terminology of semantic database modelling an atomic type; structure and set are constructed types.

Extensional objects Extensional objects are object of which the type structure is defined by intensional objects. We distinguish two types of extensional objects:

Instance An element of the entity set denoted by a concept or a particular set or structure.

Tuple An element of a relation. E.g a row in a relation table.

Auxiliary constructs The third group of constructs is used in the definition of intensional objects. We distinguish four of such auxiliary constructs:

Expression Simple expressions consisting of three parts: an operand, a logical operator and a property of some construct.

Sub-type-of Sub-type hierarchies of concepts, relations, structures, and sets.

Property Properties are functions defined on various types of constructs: concepts, relations, structures, sets. The range of the function is a value of a (predefined) value-set.

Value and value-set Value sets are the ranges of property values (attribute functions in ER modelling). Some value sets like value-sets *string*, *natural-number*, *integer*, *real* and *boolean* are assumed to be predefined.

In the following sections the language for writing down these constructs is defined using BNF grammar rules and is illustrated with examples in the audio domain. The notation used in these grammar rules is given in Table 4.4.1. In addition, we propose a notation for a graphical representation of DDL definitions.

In the rest of this chapter we will use the term "object" in a very wide sense, referring to both extensional and intensional objects, unless explicitly stated otherwise.

Construct	Interpretation
::= ⋆ + [ ] ⟨ ⟩ | .	Symbols that are part of the BNF formalism
X ::= Y.	The syntax of X is defined by Y
[X]	Zero or one occurrence of X
X⋆	Zero or more occurrences of X
X+	One or more occurrences of X
X | Y	One of X or Y (exclusive-or)
⟨ X ⟩	Grouping construct for specifying scope of operators e.g. ⟨ X | Y ⟩ or ⟨ X ⟩*.
symbol	Predefined terminal symbol of the language
symbol	User-defined terminal symbol of the language
symbol	Non-terminal symbols

TABLE 4.1: Synopsis of the notation used in BNF grammar rules

4.4.2 Concept The notion of "concept" is a central construct in the DDL. It is used to represent a class of objects in the real or mental world of the domain studied. The term "concept" corresponds roughly to the term ' 'entity" in ER-modelling and "class' in object-oriented approaches.

Every concept has a *name*, a unique string which can serve as an identifier of the concept, possible super concepts (multiple inheritance is allowed), and a number of *properties*. Note that a property is a (possibly multi-valued) function into a value set. A number of value-sets are assumed to be pre-defined, such as strings, integers, natural numbers, real numbers and booleans. A newly defined value-set can be a range of integers or reals or an enumeration of strings, When inheriting properties, the KL-ONE notions of value restriction, cardinality restriction, and differentiation are supported by the DDL.

Relations of a concept with other other concepts, with structures, etc. should be modeled separately with DDL relation definitions (see further).

concept-def ::= **concept** *concept-name*;
[**sub-type-of:** *concept-name* ⟨, *concept-name*⟩⋆;]
[properties].

properties ::= **properties:** [property-def ⟨, property-def⟩⋆].

property-def ::= *property-name*: value-set-def;
[cardinality-def]
[differentiation-def;].

value-set-def ::= **number** | **integer** | **natural** |
string | **boolean** | **universal** |
number-range(*number*, *number*) |
integer-range(*integer*, *integer*) |
{*string-value* ⟨, *string-value*⟩⋆ }.

cardinality-def ::= **cardinality:** [**min** nat] [**max** ⟨nat | **infinite**⟩];.

differentiation-def ::= differentiation of *property-name*(*concept-name*).

Some example concepts from the audio domain are given below. We support the KL-ONE notion of value restriction implicitly by allowing redefinition of the value set or the cardinality of a property in sub-concepts.

```
concept component;
  properties:
    state: universal;

concept amplifier;
  sub-type-of: component;
  properties:
    state: {ok, not-ok};
      cardinality: min 0 max 1;

concept signal-transmitter;
  subtype-of: component;
  properties:
    signal: {present, absent};
      differentiation of state(component);

concept input-port;
  sub-type-of: signal-transmitter;

concept output-port;
  sub-type-of: signal-transmitter;

concept volume-system;
  subtype-of: component;
  properties:
    state: {on, off};

concept power-system;
  subtype-of: component;
  properties
    state: {on, off};
```

The example concept definitions above specify a sub-type hierarchy of components. This hierarchy is represented graphically in Fig. 4.1. The hierarchy contains also some additional components that appear in examples further on.

4.4.3 Set A set is a composite object. A set has zero or more members. A set can usually be modelled (implicitly) with cardinality constraints on a relation. Often however, the knowledge engineer will want to introduce a set as an explicit notion and thus be able to give it a name and a status (e.g. property values) of its own.

The members of a particular set should be of the same type. This member-type can be any object, including a set. Both properties and sub-types can be defined on sets.

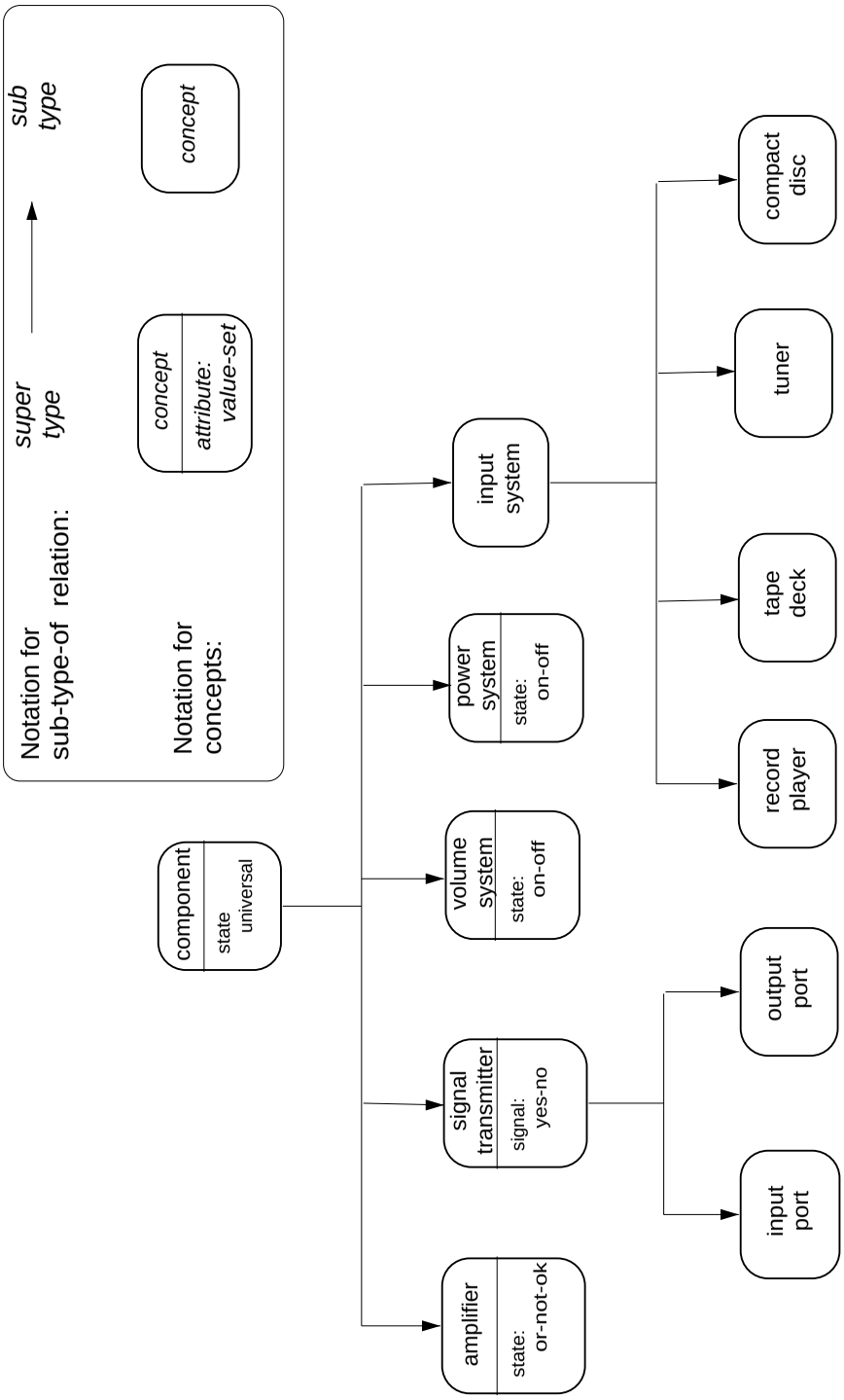

FIGURE 4.1: Example sub-type hierarchy of components in the audio domain

set-def ::= **set** *set-name*;
[**sub-type-of:** *set-name*;]
element-type: object-ref;
[cardinality-def]
[properties].

object-ref ::= object-name | **instance**(object-name).

object-name ::= *concept-name* | *set-name* | *structure-name*.

An example set in the audio domain is the set of input systems of a particular audio system, e.g. two tape decks, a compact disc, and a tuner.

set *input-systems;*
element-type: instance(input-system);

Note that the elements of the set are in this case not components, but instances of components (e.g. tape-deck-1, tape-deck-2, cd-1, tuner-1). This set could be used to store information about the configuration of the audio system being diagnosed. Fig. 4.2 gives a graphical representation of this set definition.

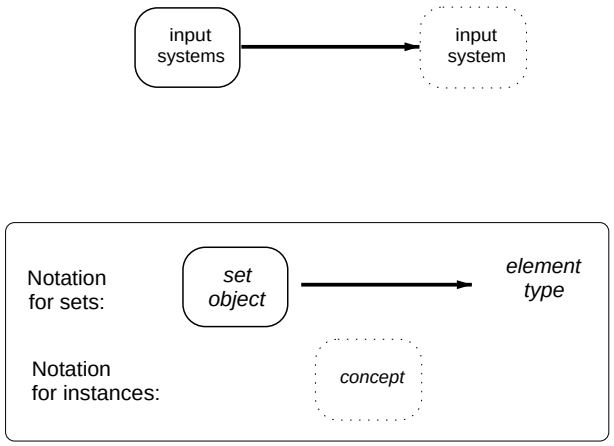

FIGURE 4.2: Graphical representation of the definition of a set. This set is a set of *instances* of the specified element-type. An alternative would be a set of sub-concepts of this element-type.

4.4.4 Relation The *relation* is an central construct in modelling a domain. In the DDL we allow various forms of (binary) relations to cater for the specific requirements imposed by knowledge-based systems. The relation construct is used to link any type of objects to each other, including concepts, instances, sets and structures. As was pointed out in Sec. 4.3, we allow explicit distinctions between relations between concepts and relations between instances.[4] For example, it is should be possible to define in the audio domain both relations between sub-concepts of *component*. e.g. to specify prototypical

[4] In ER modelling most relations (except for the sub-type relation) are defined to hold between instances.

configurations of an audio set, as well as relations between component instances, in order to represent a specific configuration.

Apart from defining relations between objects, the DDL also allows relation arguments that represent *expressions about* objects. The notion of expressions as a domain modelling construct in the specification of relations is introduced because, as pointed out before, these occur often in "domain rules". One of the major points of the domain modelling enterprise is to describe the structure of these domain rules. This type of domain description is currently lacking in many KBS development projects. The *expression* construct provides a suitable way of modelling the structure of domain knowledge in which simple expressions such as $age(patient) > 65$ and $temperature(patient) = high$ appear. The general form of expressions is $< operand >< operator >< value >$ where the *operand* is a property of an object, the *operator* is one of $=, \neq, <, \leq, >, \geq, \in, \subset, \subseteq, \supset, \supseteq$, and *value* is a sub-set of the value-set of the function.

The grammar rules below specify the DDL for defining relations. A DDL relation is always a binary relation. The relation is directional in the sense that the relation name should be chosen in such a way that it can be read as "argument-1 *relation name* argument-2". An inverse name is optional. A relation can inherit information from a super-type. The most interesting part of the DDL definition of relations is the definition of its arguments. We provide three possible types of arguments: (i) a single object (e.g. concept, instance, structure), (ii) an expression about an object, and (iii) sets of objects or expressions. Expressions can be restricted to particular properties of an object. If no properties are specified, it is assumed that the expression may concern any property of the object.

Relations can themselves also have properties. The classic example of such an property is the wedding date of two married people. Also, it is possible to define standard semantic properties of the relation (transitive, symmetric, etc.), known tuples of the relation, and/or some additional constraints or interpretations connected to the relation (the *axioms* field[5]).

relation-def ::= **relation** *relation-name*;
[**sub-type-of:** *relation-name*;]
[**inverse:** *relation-name*;]
argument-1: argument-def
argument-2: argument-def
[properties]
[**semantics:** semantic-property ⟨, semantic-property⟩⋆;]
[**tuples:** text;]
[**axioms:** text;].

argument-def ::= ⟨ argument | **set**(argument) ⟩ ;
[**role:** *role-name*;]
[cardinality-def].

argument ::= object-ref | expression-argument.

[5]The fact that this field is defined as text in the DDL is not as inconsistent as it looks. The idea is that the knowledge engineer often has additional semantic information about a relation which she should be encouraged to write down as unambiguously as possible, preferably in logic but not necessarily.

expression-argument ::= **expression**(object-ref) |
expression(*property-name* **of** object-ref).

semantic-property ::= symmetric | transitive | reflexive.

Below we give some example relation definitions. The first example concerns a aggregation relation between component *types* (as opposed to instances). This relation could be used to store knowledge about the prototypical structure of an audio system in the knowledge base. Some examples tuples of this relation are listed as well.

relation *has-sub-part;*
inverse: part-of;
argument-1: component;
role: aggregate;
cardinality: min 1; max 1;
argument-2: component;
role: part;
cardinality: min 0; max infinite;
semantics: transitive;
tuples:
amplifier HAS-SUB-PART input-port
amplifier HAS-SUB-PART output-port
amplifier HAS-SUB-PART volume-system
amplifier HAS-SUB-PART power-system;

The second example defines a causal relation as a relation between state properties of components. The first argument constitutes a set of expressions representing the conditions for the causal transition. The intended interpretation of the first argument is described in the *axioms* field. One example tuple of the causal relation is listed, involving components defined in earlier in this chapter.

relation *causes;*
inverse: caused-by;
argument-1: set(expression(state of component));
role: cause;
argument-2: expression(state of component);
role: effect;
tuples:
signal(input-port) = present
state(power-system) = on
state(volume-system) = on
CAUSES
signal(output-port) = present;
axioms:
first argument should be interpreted as a conjunction;

Both example relation definitions are graphically represented in Fig. 4.3,

A last example shows how the qualitative-abstraction rules presented earlier can be modeled:

relation *qualitative-abstraction*

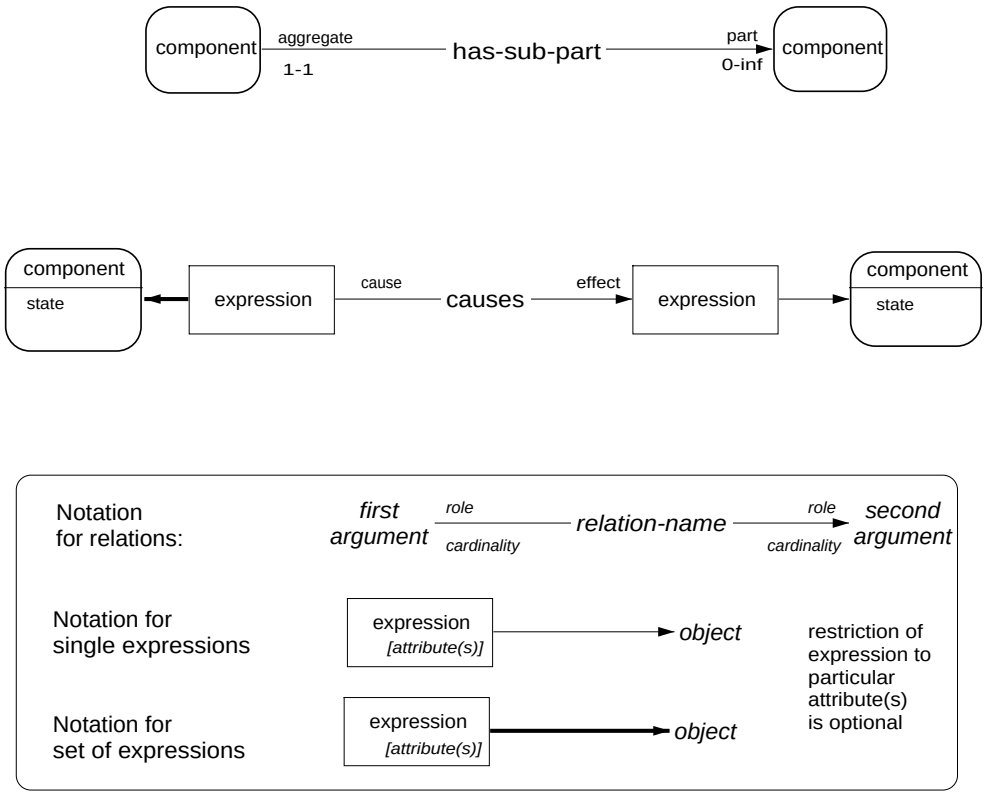

FIGURE 4.3: Graphical representation of two example relations in the audio domain.

inverse: specification
argument-1: set(expression(quantitative-data))
role: conditions;
argument-2: expression(feature)
role: conclusion
axioms:
first argument should be interpreted as a conjunction;

4.4.5 Structure A structure is a composite object. It has a number of parts which contain one or more single objects or sets of objects (concepts, instances, tuples, sets, other structures)

Like the other objects, structures have a unique name and a possible super structure. Parts are inherited and overruled in a similar fashion as roles in relations. Structures can also have properties.

structure-def ::= **structure** *structure-name*;
[**sub-type-of:** *structure-name*;]
parts: part-def+
[properties]
[**axioms:** text;].

part-def ::= *part-name*: part-element-def+.

part-element-def ::= ⟨ single-element-def | set-element-def ⟩;.

single-element-def ::= object-ref | tuple-ref.

set-element-def ::= ⟨ **set**(object-ref) | **set**(tuple-ref) ⟩
[cardinality-def].

tuple-ref ::= **tuple**(*relation-name*).

An example of a structure is a causal network in the audio domain, that consists of two parts: (i) *nodes* containing the components involved in the network, and (ii) *causal relations* containing tuples of the relation *causes*.

structure *causal-network;*
parts:
nodes: set(component);
causal-relations: set(tuple(causes)));

This structure is graphically represented in Fig. 4.4.

Another example could be a structure for modelling the input data for audio system diagnosis:

structure *input-data*
parts:
complaint: instance(observable)
initial-data: set(instance(observable))

This structure can contain a number of instances of observables, where one is regarded as the actual complaint and the others are additional data available at the start of the diagnostic process.

4.4.6 Graphical representation In Fig. 4.5 the graphical representation introduced for the DDL is summarised. Although there is some loss of information in the pictures constructed in this fashion, these type of pictures tend to be an important communication vehicle throughout the knowledge engineering process.

Fig. 4.6 shows how the domain schema for the audio domain as presented Sec. 3.4.1 can be represented graphically. In this thesis additional examples are given of the use of the DDL, especially for the Sisyphus application (Ch. 7, Appendix A). An earlier version of this DDL has also been used to model a domain concerning submarine detection through acoustic analysis [Schoenmakers, 1992]. The DDL has also formed the basis for the domain modelling language being developed in the KADS-II project.

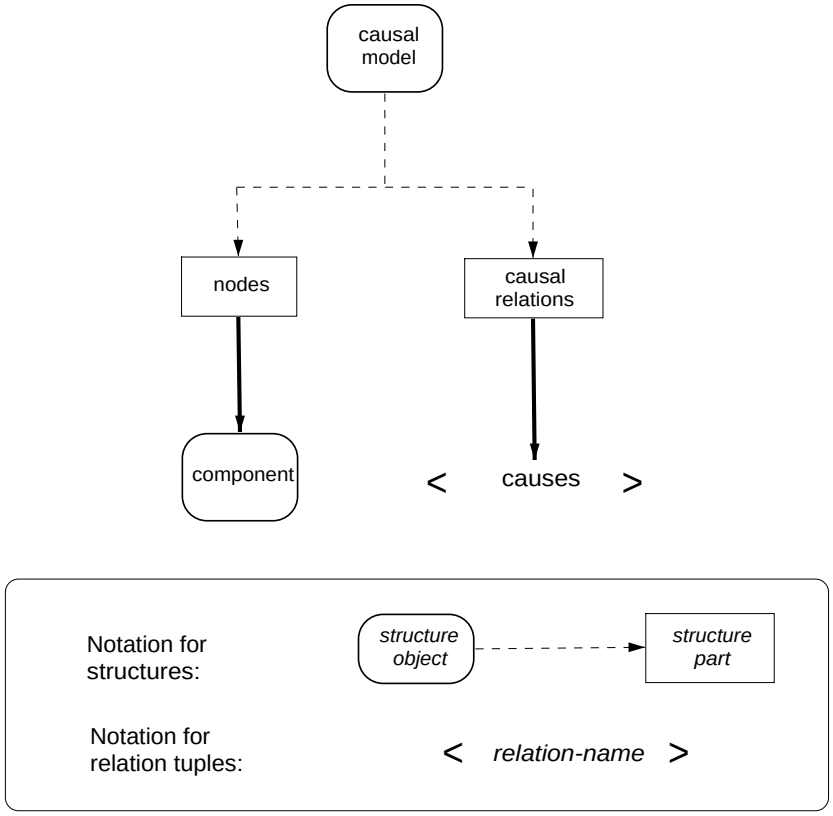

FIGURE 4.4: Graphical representation of DDL structures.

4.5 Discussion

Clancey remarks in his most recent analysis of NEOMYCIN [Clancey, 1992] that a key point in knowledge engineering (in his particular case: structuring the set of meta rules) is the process of making finer-grained distinctions in the domain knowledge. One could view this as a process of detailing the role differentiation of pieces of domain knowledge. He uses the relational representation language MRS underlying NEOMYCIN to describe these distinctions. We would argue this is typically a situation were one would like to use a more general tool to describe these distinctions. This enhances the reusability of the theory that is uncovered by Clancey in his analysis.

The DDL is precisely built for this purpose. As remarked before, the DDL presented in this chapter is part of the modelling framework and is not meant to serve as another knowledge representation formalism. Its aim is to provide a generalisation over such formalisms in such a way that one can specify aspects of the domain knowledge without committing oneself to a particular symbolic representation. In this section, we briefly address this generalisation aspect and also discuss some points concerning the semantics of the DDL.

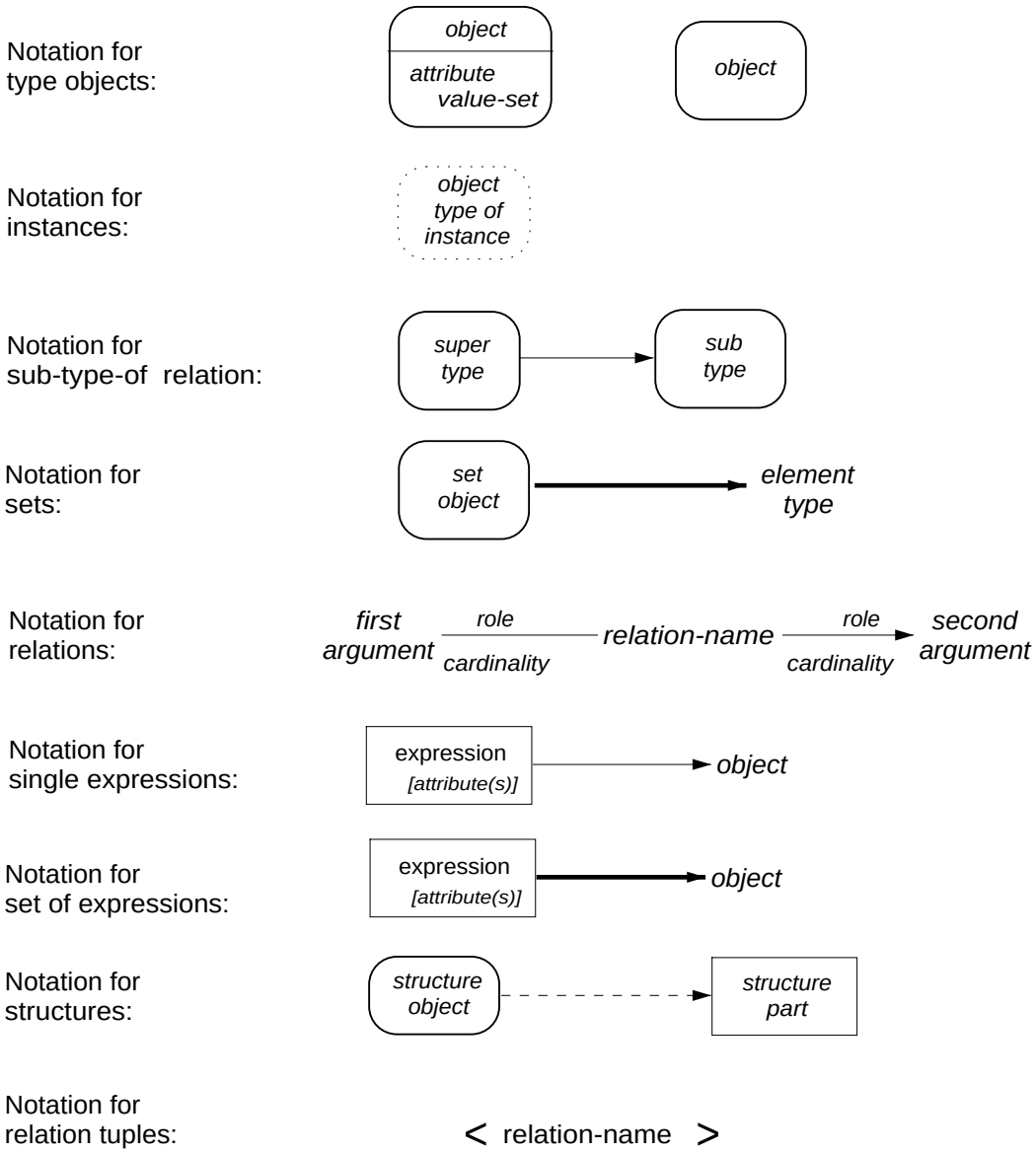

FIGURE 4.5: Legend of the graphical representation of a domain description

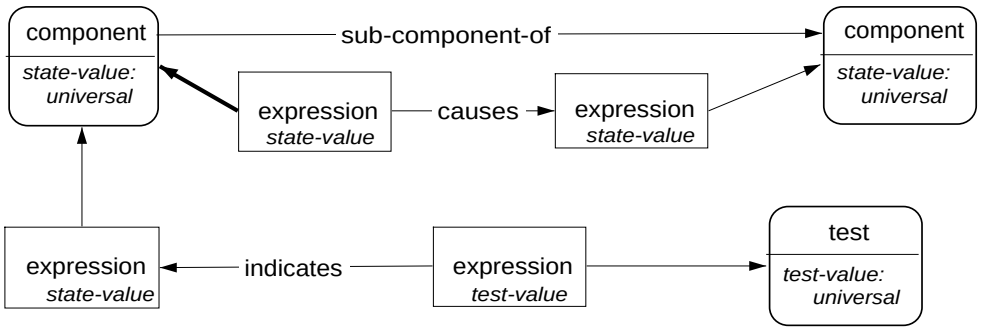

FIGURE 4.6: Graphical DDL representation of the major part of the domain schema presented in Table 3.1. The sub-type relations between components were shown in Fig. 4.1.

Relation between DDL and some KR formalisms

KL-ONE The DDL contains most constructs offered by KL-ONE, albeit in a somewhat scruffy form: concepts, instances, (multiple) inheritance, roles (either through properties or through relations), role restrictions (both value restrictions and cardinality restrictions) and role differentiations.

A mapping from a DDL description to a KL-ONE representation would lead to a first, at some points incomplete, version of a definitional hierarchy. The main additional decisions that one would need to take is whether all relations in which a concept is involved should indeed be mapped onto KL-ONE roles. Values of roles are used by the classifier of concept instances and it often turns out that some relations are irrelevant for classification purposes. For example, in classifying car types such as sedans, hatch backs, etc, an *owner* relation of the concept *car* is irrelevant. We have noticed, also from personal experience in the StatCons domain, that using KL-ONE representations already during analysis often leads to commitments to the symbolic representations: e.g. in the car example, the *owner* role is defined as an optional one (because classification fails otherwise), even if every car has some (unknown) owner.

KEE KEE offers an organisation of production rules in a hierarchy of rule sets. This relates quite well to the DDL representation of relations between expressions, which in fact define a similar organisation. However, in a mapping onto KEE the schematic description of the structure of the rules in particular set gets lost, because KEE does not offer facilities for this type of structural description. E.g. when implementing the audio domain-knowledge in KEE, one would create rule sets for both the *cause* and the *indicates* relation, but the information about the internal structure of these type of rules would be lost.

A cumbersome aspect of KEE is that it does not make an explicit distinction between concepts and instances: both map onto KEE *units*.

Semantics of the DDL

An important question still remains open, namely what do this DDL descriptions mean: i.e. what are their semantics? A full definition of the semantics of the DDL is clearly out of the scope of the present work. A few remarks are however necessary.

Firstly, many constructs in the DDL are derived from data modelling research in which the semantics of these constructs have been studied in detail. The semantics of the KL-ONE constructs has been a major research topic in AI over the past decade. Also, in semantic database modelling the semantics of various types of sub-types relations, aggregations, groupings, etc. have been described extensively, e.g. [Abiteboul & Hull, 1987; Davis & Bonnel, 1990].

The main blind spot with respect to semantics concerns the various additional types of relations introduced in the DDL, especially those between concepts and between expressions. We have give some informal descriptions about how such relations should be interpreted. E.g the *requires* relation in Sec. 4.4.4 between a *department role* concept and an expression about a room property should be interpreted as a universally quantified statement about all objects that are denoted by these concepts. Sowa's "conceptual structures" [Sowa, 1984] provide a logical formalisation that can be used for specifying this type of semantics. The disadvantage is that such representations require much more detail and are less schematic. It is doubtful whether a complete, detailed domain-knowledge specification is useful in the analysis stage, where the focus can sometimes change rapidly.

In the context of KADS, the role of domain-knowledge semantics is also slightly different than is the case traditionally. As all inferences that are made with the domain knowledge are specified through knowledge sources, one could take the point of view that the intended semantics of domain-knowledge is externally attributed to it through these inferences. We touch here upon a very delicate topic, which has given rise to many debates, especially in the study of formal languages for KADS models. It is possible that the the statement in Sec. 3.4.1 that "adding a simple deductive capability would enable the system ...to solve all problems solvable by the theory" may need some qualification.

Chapter 5

Model Construction

In this chapter the focus is on the construction of inference structures. First, some extensions of the graphical notation of inference structures are proposed to overcome a number of ambiguities in these diagrams. We then discuss a top-down model construction process in which an inference structure is successively refined. This description is based on observations made by Patil (1988). Decision criteria that influence the construction process are discussed. The result is a KADS inference structure for heuristic classification, which we relate to observations Clancey (1992) makes about this model. In the discussion, we suggest that such a top-down construction process can be supported by a library of generic model components of a smaller grain size than interpretation models. We discuss some examples of these generic components.

This chapter will be published in a collection of articles on KADS. It is co-authored by Bob Wielinga. Reference: Schreiber, A. T & Wielinga, B. J, (1993). Model construction, In Schreiber, A. T., Wielinga, B. J., & Breuker, J. A., editors, *KADS: A Principled Approach to Knowledge-Based System Development.* Academic Press, London.

5.1 Introduction

Generic components of the model of expertise can be used to support top-down knowledge acquisition. In Ch. 3, *interpretation models* were discussed that provide such template models. In this chapter we explore the construction of inference structures and the role of generic model components in more detail.

We start in the next section with a discussion on ambiguities that may arise in the current (graphical) representation of inference structures, We propose an extended notation to disambiguate the representation of inference structures. In Sec. 5.3 we discuss some aspects of the model-construction process in general: operations, methods, criteria.

In Sec. 5.4 we discuss adaptations of the inference structure of an existing interpretation model. We describe two example adaptations of the interpretation model for systematic diagnosis. In Sec. 5.5 we describe a top-down model-construction process in which one starts off with a relatively simple hypothetico-deductive model for diagnosis, and subsequently refines this model step by step. This work is based on an analysis of medical diagnosis by [Patil, 1988]. In addition, we discuss some points concerning the KADS inference structure for heuristic classification, which resulted from the top-down model construction process and relate this to Clancey's observations about this model [Clancey, 1992].

In Sec. 5.7 the observation is made that inference structures in fact consist of a number of reoccurring template model components of a smaller grain size than interpretation models. We identify some examples of such model components and discuss their role in knowledge engineering.

5.2 Disambiguating the Graphical Representation of Inference Structures

Inference structures are among the most frequently-used ingredients of KADS. In almost any presentation of an application of KADS, the description of the inference structure plays a dominant role. Inference structures are, however, informal diagrams. There does not exist a full set of composition rules for specifying inference structures. The following list presents a set of composition rules which most KADS developers and users agree upon.

- A knowledge source can have any number of inputs, including zero.
- The set of inputs of a knowledge source is interpreted as a conjunction.
- A knowledge source has only one output.
- The name given to a knowledge source should be a member of the typology defined in [Breuker *et al.*, 1987].
- If more than one knowledge source of the same type appears in an inference structure, these should be numbered in order to avoid confusion (e.g. *select-1* and *select-2*).

This list is incomplete, as the many discussions about representation of inference structures in papers and during KADS user meetings [Ueberreiter & Voß, 1991; Bauer & Karbach, 1992] show. In this section we investigate a number of frequent sources of ambiguity in inference structures and propose some additional graphical notations to amend these problems. These extensions are used in the diagrams in the rest of this chapter.

5.2.1 Transfer tasks Traditionally, inference structures were supposed to contain only "real" inferences: derivations by the system of new "knowledge". This meant that transfer tasks, such as obtaining the value of an observable, could not be included in an inference structure. This can obscure parts of an inference structure. An example can be found in the inference structure for systematic diagnosis (Fig. 3.5). The *specify* inference in the lower-left part of this figure (repeated for convenience in Fig. 5.1a) takes as input a *hypothesis* and an *observable* and produces as output a *finding*. In the corresponding task structure (see page 38) it can be seen that this inference is in fact a concatenation of a knowledge source and a transfer-task invocation:

specify(hypothesis → observable),
obtain(observable → finding)

In the model of expertise, transfer tasks are treated as black-box functions (see Sec. 3.4.3). Knowledge sources and transfer tasks together form the lowest level of functional decomposition in the model of expertise. One could say that transfer tasks are basic functions that do not apply domain knowledge to make inferences. Thus, it seems appropriate to include transfer tasks in an inference structure. A dashed-ellipse notation is used to distinguish transfer tasks from knowledge sources. Fig. 5.1b shows the extended representation of the original figure using this notation.

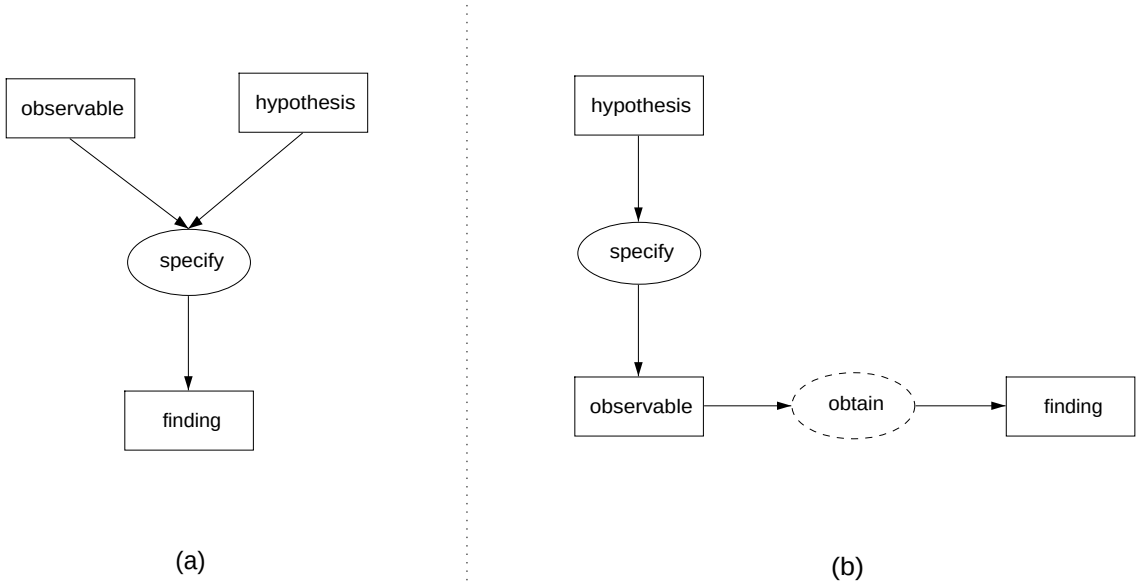

FIGURE 5.1: Two representations of a *specify* inference in systematic diagnosis. At the left (a) the transfer task is implicit. At the right (b), an explicit obtain step is introduced.

5.2.2 Meta-class element vs. set Another issue that has arisen with respect to inference structures concerns the nature of meta-classes. As pointed out in Sec. 3.4.2, a meta-class constitutes a functional name (a role name) for a set of domain objects that can play this role. Some knowledge sources operate on or produce *one particular* object, others work on a *set* of these objects. This can lead to ambiguities in inference structures, for example if one inference produces one object and another inference works on a set of these objects, possibly generated by some repeated invocation of the first inference. An example of this kind is described in Fig. 5.2a.

The intended interpretation of this figure is that the *abstract* inference can generate from a set of findings a more abstract finding, and that the *select* inference selects a *specific finding* from the full set. Such a specific finding could, for example, be used in triggering a hypothesis.

KADS users have tried to overcome this problem with elements and sets in various ways:

- By specifying all inferences as working on sets instead of elements. E.g `abstract(findings → findings)`.
- By introducing additional set operations such as *join* into the inference structure.

Both solutions are sub-optimal. The first one hides the fact whether the inference *inherently* operates on a set or rather on a single element of the set. The second one introduces additional complexity and makes inference structures less readable. We propose to use a slight variation in the connection between a meta-class and a knowledge source to indicate cardinality: a thin line represents a single meta-class element; a thick line a set of meta-class elements. An example of this notation is shown in Fig. 5.2b. This notation avoids the disadvantages of the solutions mentioned above.

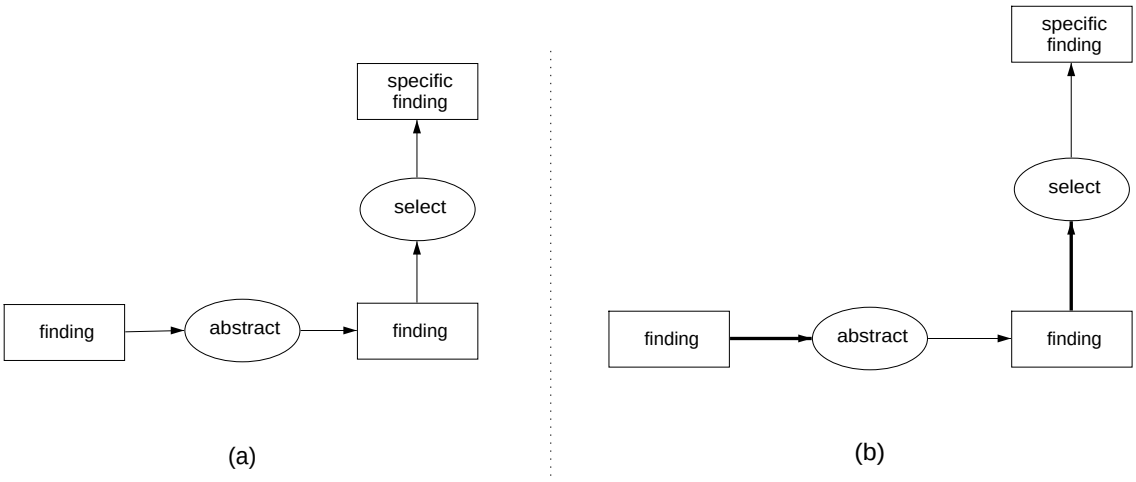

FIGURE 5.2: Two partial inference structures. In part (a) the fact that the input for both *abstract* and *select* is a *set of* findings, is not represented. In part (b) the thick-line notation is used to indicate a set of some meta-class element.

5.2.3 Meta-class names Another problem arises from the names given to meta-classes. There are two types of names that one can give to a meta-class:

1. A *general* role name of elements involved in carrying out a task. In diagnosis, such categories could be *observable*, *finding*, and *hypothesis*. These names can probably best be viewed as abstract data types.
2. A *specialised* role name for elements in an inference. These names constitute a specialisation of the general categories: e.g. *test observable*, *discriminating observable*. Specialised names describe roles that are specific for the particular inference *process* depicted in an inference structure.[1]

Specialised names such as *test observable* are useful and make the inference structure easier to interpret. On the other hand, some inferences may operate on the general category (e.g. observable). One would like to be able to specify both general and specialised role names and still be able to clearly show the dependencies between inferences. A potential solution is to allow the knowledge engineer to write the specialised role names on the arrow connecting a knowledge-source and a meta-class. An example of this notation is shown in Fig. 5.3.

5.2.4 Domain knowledge used by knowledge sources Inference structures only show the dynamic data that are being manipulated by a knowledge source (the meta-classes). Sometimes, it is useful to show also what type of domain knowledge the knowledge source uses to derive the output from the input (cf. [Linster, 1992]). This domain knowledge is specified in the domain view (cf. Sec. 3.4.1).

One could argue that this is an unwanted extension of the inference structure, as knowledge sources are in fact domain-independent generalisations of the application of

[1] In KADS-II terms one would say that the specialised names are introduced by the methods applied to achieve the task.

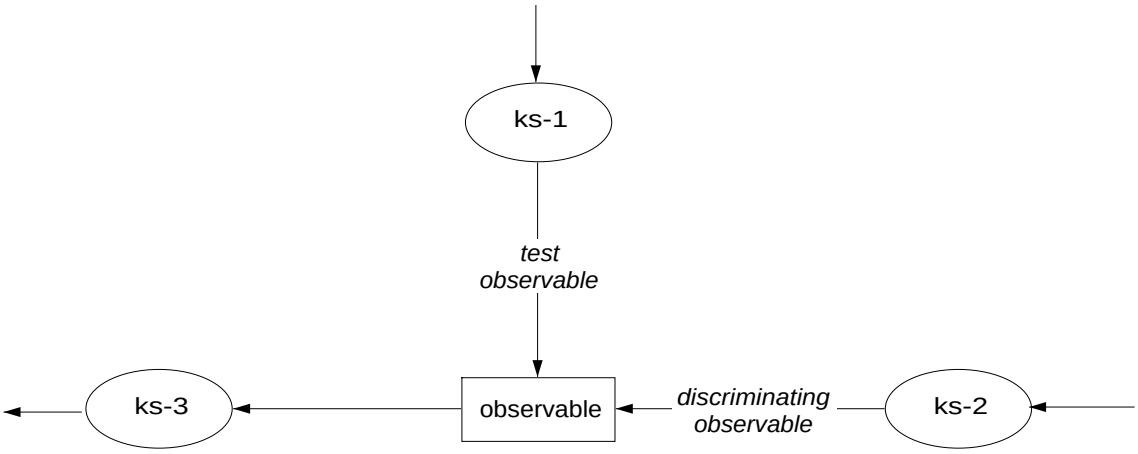

FIGURE 5.3: Introducing general and specialised meta-class names in an inference structure. An object of meta-class observable can be generated by two inferences, that each give a specialised name to this meta-class (*test observable* and *discriminating observable*). Every *observable* object can serve as an input for the third knowledge source. This last point would be difficult to represent if the specialised name had been written directly into (two separate) boxes.

domain knowledge. However, it can be useful at some points during knowledge engineering to make the nature of the domain knowledge explicit, although this destroys the domain-independence of an inference structure. We use a dashed arrow to indicate the domain knowledge used by a knowledge source. An example of this notation is given in Fig. 5.5 in the next section.

5.3 Model-Construction Process

In this section we discuss briefly three aspects of the model-construction process:

- The types of (knowledge-engineering) operations on inference structures.
- Some knowledge-engineering methods that can be used in constructing and adapting an inference structure.
- The types of criteria that are used in making decisions during model construction.

These three aspects can be summarised respectively as the "what, how and why" of model construction.

5.3.1 Types of operations on inference structures

indexinference structure, operations on

In general, five types of modifications can be made to a prototypical inference structure:

Renaming Sometimes, the metaclass and knowledge source names used in the inference structure are too general. In that case, the knowledge engineer might want to use another term to give a more precise specification of the role of a knowledge element in the inference process.

Refinements In some cases, an inference in the inference structure is too coarse-grained to describe the inference process required in the application domain. In that case,

a refinement of this inference in the inference structure is appropriate. Such a refinement provides additional terminology (in terms of knowledge sources and/or meta-classes).

Additions/Augmentations If inferences are required that can not be specified as a refinement of the current inference structure, additional inferences might need to be added to the inference structure.

Simplifications A simplification is the reverse process of a refinement. This should be done if a set of inferences is too fine-grained for the purposes of the application.

Deletions A deletion is the reverse of an addition. Sometimes, a part of an inference structure of an interpretation model is not relevant for a particular application. In that case, this part of the inference structure should be left out.

Figure 5.4 summarises these different types of operations on inference structures.

5.3.2 Model-construction methods Two knowledge engineering methods are often applied in the construction of inference structures: (i) task decomposition, and (ii) knowledge differentiation.

Task decomposition Sometimes, the analysis will reveal that some inference in a (provisional) inference structure constitutes a task which can be decomposed in a number of parts. This leads to a more detailed inference structure with additional vocabulary. What was originally conceived as an inference, often reappears as a task in the task knowledge.

Task decomposition is a part-of decomposition, in which a task and its sub-parts do not need to have anything in common, except for a mapping between the input/output of the top-task and its parts. Task decomposition typically involves a *refinement* operation on the inference structure (see above).

Knowledge differentiation Knowledge differentiation is the process of introducing new knowledge roles during the modelling process. Finer-grained distinctions are introduced in the inference structure. The difference with task decomposition is that in knowledge differentiation the basic structure of the model stays the same. Knowledge differentiation can involve various types of operations: *renaming* meta-classes and/or knowledge sources, *adding* a new meta-class and knowledge sources that operate on that meta-class, or *refinements*.

One particular form of knowledge differentiation is *inference differentiation*. In inference differentiation, an inference is differentiated into a set of sub-inferences. Inference differentiation involves, just like task decomposition, a refinement operation on an inference structure, but there is an inherent difference between the two. In inference differentiation, the differentiated inferences share the features defined for the general inference. It is probably best viewed as a sub-type relation (see Fig. 5.12).

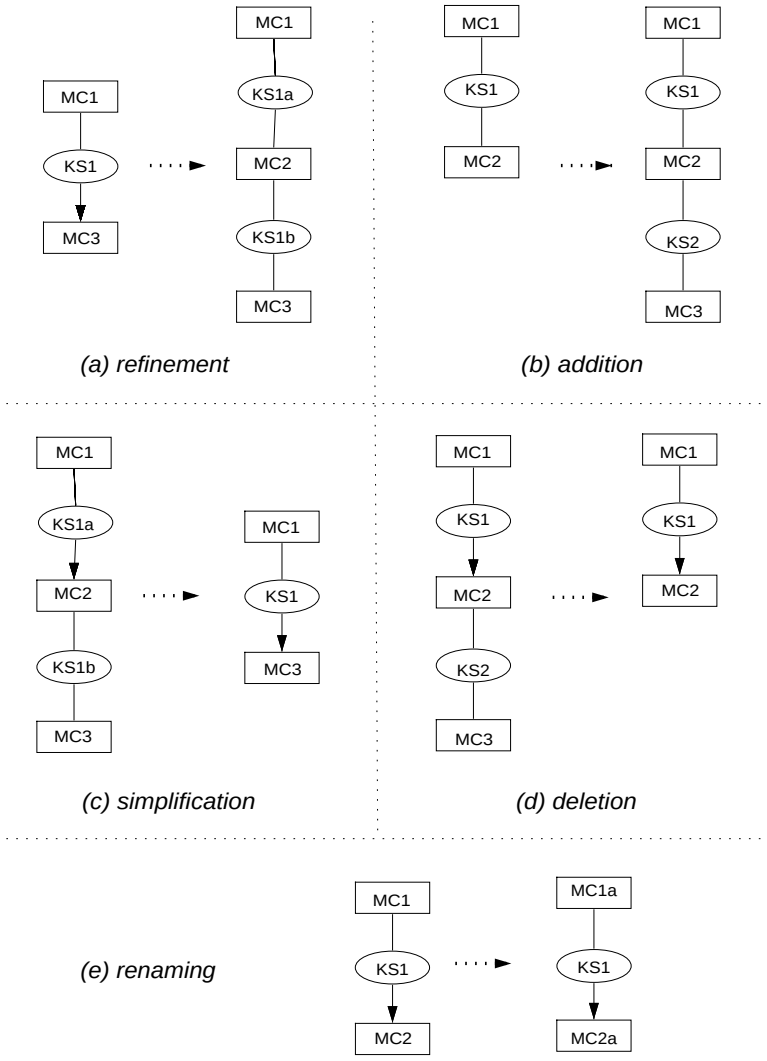

FIGURE 5.4: Examples of five types of modification operations on an inference structure

5.3.3 Decision criteria Model construction is guided by three types of characteristics of the application domain:

- The nature of the knowledge in an application domain (e.g. "are causal models available?").
- The constraints posed by the task environment (see Sec. 3.3): e.g. the required certainty of a solution.
- Computational constraints: is it possible to find computational techniques that realise the specified behaviour?

These characteristics can be used as decision criteria and specify the *rationale* for decisions in the model construction process. In the "Components of Expertise" framework [Steels, 1990] these characteristics are called "task features" and the three categories

epistemic, *pragmatic* and *computational* task features respectively.[2]

In the next section we discuss the adaptation of the inference structure of the interpretation model for systematic diagnosis. In Sec. 5.5 we discuss a top-down model construction process. After each operation on an inference structure, the type of operation, the method and the type of criteria used will be summarised in a tabular form.

5.4 Tuning the Inference Structure for Systematic Diagnosis

The use of template descriptions such as interpretation models provides a powerful tool for knowledge acquisition. However, applying such a template to a particular domain will often reveal that the model does not completely fit the data on human expertise. Most interpretation models embody only a minimal set of inferences necessary for solving a problem with this method. The model needs to be adapted.

The model of systematic diagnosis discussed in Ch. 3 can be adapted in various ways. We discuss two adaptations relevant for the audio domain.

5.4.1 Dynamic system-model assembly The plain model of systematic diagnosis (of which the inference structure is shown in Fig. 3.5) presupposes that the applicable system model is *selected* using knowledge about fixed decompositions of the system being diagnosed. However, the configuration of an audio system is usually not fixed (i.e. a constraint of the task environment). System elements such as a CD-player, a second tape-deck, head phones or additional speakers may or may not be present. This potential problem can be handled by replacing the simple *select* inference with a more complicated *assemble* inference (Fig. 5.5). This operation involves adding an extra meta-class (*initial data*) and renaming the knowledge source (*select* becomes *assemble*)

In this assemble step additional data (*initial data* in Fig. 5.5) about the audio system are used to construct an applicable system model. An epistemic requirement for this differentiation is that additional domain knowledge can be made available, namely:

- A definition of potential system elements of an audio system, possibly hierarchically organised (cf. the *sub-models* in Fig. 5.5).
- Configuration rules for assembling an actual model from the possible system elements.

This modification of the plain inference structure of systematic diagnosis thus leads to a slightly more complex model with additional domain knowledge requirements.

Dynamic system-model assembly	
Operation	Addition (of a meta-class) + renaming
Method	Knowledge differentiation
Criteria	Task environment

[2] In the Components approach the task features are used for dynamic run-time task-decomposition in an actual system. In KADS, model construction is primarily seen as a knowledge engineering activity, which could be (but does not need to be) reflected in the design of the KBS (in other words, it could result in fixed task decompositions).

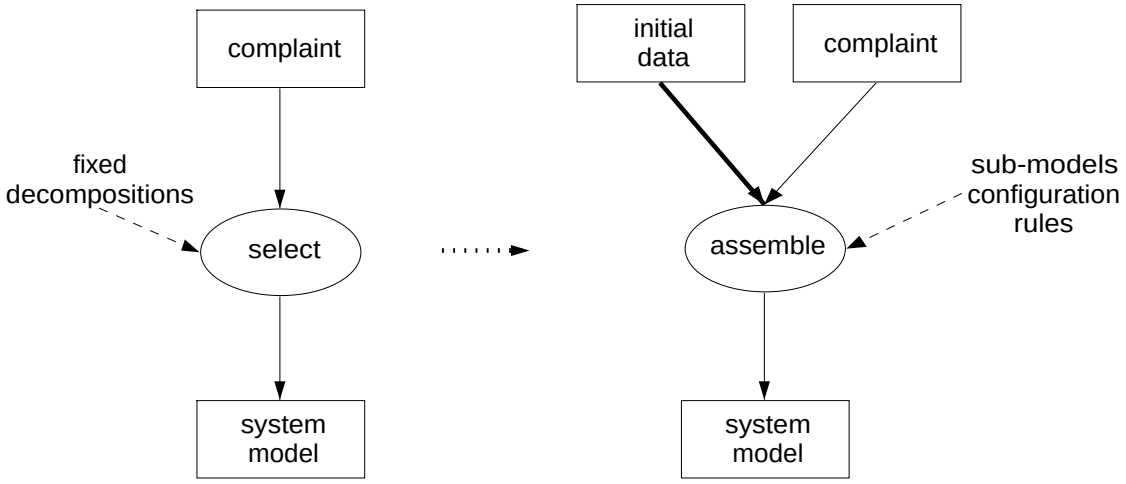

FIGURE 5.5: Adapting the model of systematic diagnosis: *system model assembly* instead of selection. The dashed arrow is used to denote the nature of the domain knowledge used by the *assemble* inference. The dotted arrow indicates a connection with other parts of the inference structure.

5.4.2 Multiple system decompositions A second, more complicated, adaptation concerns the introduction of multiple system *views*. Often, there are various ways of decomposing a device. Each decomposition represents a different view on the system. Well-known views are functional and physical decompositions. The faulty component can only be found if the right view is selected. The rationale for the adaptation is thus based on an epistemic criterion: the system to be diagnosed cannot be decomposed in one single way.

Allowing multiple views implies an additional decision in the inference process concerning view selection. Davis suggests that initial view selection should be done on the basis of characteristics of the problem (the *complaint*) using domain heuristics [Davis, 1984]. Fig. 5.6 shows the adapted part of the inference structure for handling multiple views. This adaptation involves adding a view selection inference to the inference structure in Fig. 5.5. In this case the epistemic requirement on additional decomposition knowledge is even stronger than for the previous adaptation: for each view sub-models and configuration rules should be present in the domain theory. In addition, heuristics about how to select a view need to be made available.

Introducing multiple views also involves additional task structure complexity. If one view fails to provide a solution, another view needs to be selected and the process is repeated.

View selection	
Operation	Addition
Method	Knowledge differentiation
Criteria	Epistemic

5.5 Top-down Construction

In this section we focus on the top-down construction of inference structures that support a hypothetico-deductive strategy for solving a diagnostic problem. The description is

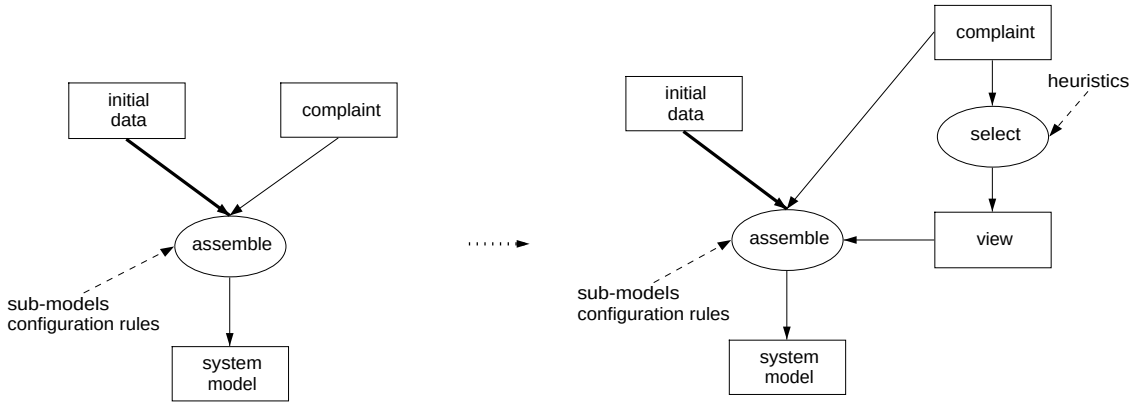

FIGURE 5.6: Adapting the model of systematic diagnosis: introducing multiple system models representing different views.

based on an analysis by Patil. He gives a historic account of the evolvement of medical AI systems for diagnosis [Patil, 1988]. He shows how one can start with a simple model of diagnosis, such as generate-and-test, and start a gradual refinement process of this model on the basis of application characteristics, such as the ones discussed earlier. This section constitutes an interpretation of Patil's analysis within the context of constructing KADS inference structures.

5.5.1 Vocabulary in diagnosis Before describing various models for diagnosis, it is useful to define a number of terms that are used in describing the diagnostic task:

Diagnosis A diagnosis is a solution of a diagnostic problem-solving process. There appear to be two different types of diagnoses, namely:

- A diagnosis as the causal explanation of the occurrence of some system state. In this case the diagnosis is either the ultimate cause or a full causal pathway.
- A diagnosis as a *label* for an internal state or a set of internal system states. It also occurs that a diagnosis is a label for some unknown internal state, e.g. in poorly understood syndromes in medicine.

Fig. 5.7 shows an example of the two types of diagnosis: *atherosclerosis* and *angina pectoris*. Atherosclerosis can represent the actual cause of findings observed in a patient. Angina pectoris is a label for an internal state, namely myocardial ischaemia (insufficient blood supply for the heart muscle). What should be considered as a potential diagnosis typically depends on the context in which the diagnostic task is being carried out. For example, angina pectoris is a relevant diagnosis in a medical emergency situation; atherosclerosis is a diagnosis that is useful in deciding on corrective action in a non-emergency situation.

Observable An observable is a property the value of which can be observed for the system (patient, device) being diagnosed. Example observables are weight, length, blood pressure, position of a knob, etc.

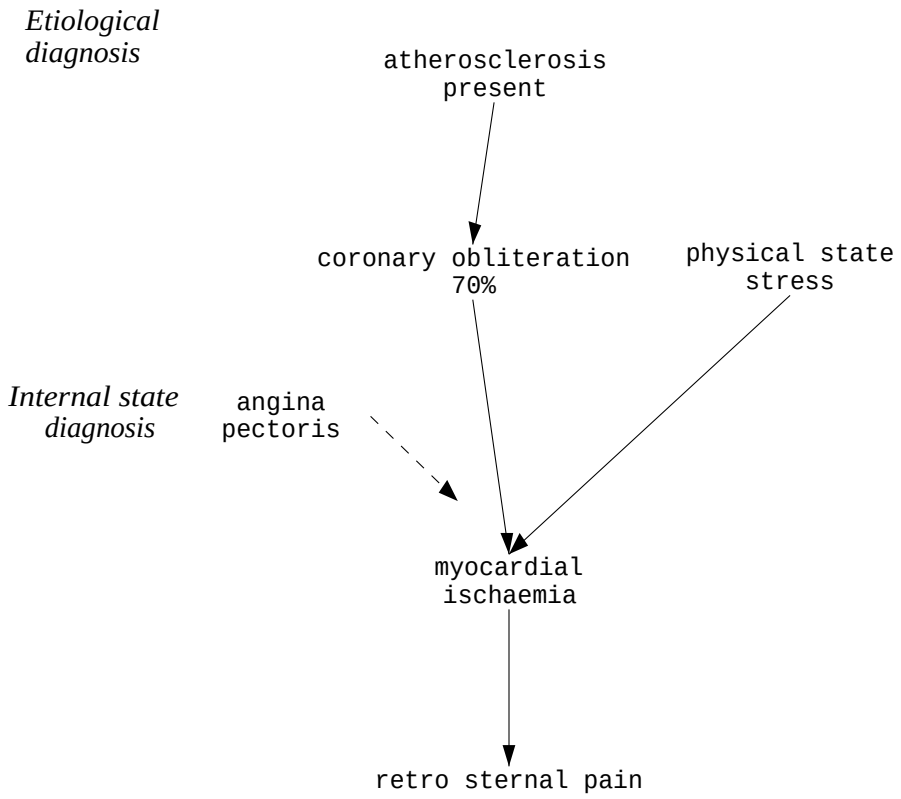

FIGURE 5.7: Example diagnosis of chest pain

Finding A finding is a value expression about an observable, e.g. $weight = 80, length = 1.90$.

Hypothesis A hypothesis is some object that is either considered as a potential solution for a diagnostic problem or constitutes some relevant intermediate state.

Differential The differential is the set of hypotheses that is considered for a particular diagnostic problem.[3]

In the rest of the chapter we use these terms to describe models of diagnosis and introduce, where necessary, additional terms.

5.5.2 Diagnosis through direct matching Diagnosis is a problem solving task in which the input is formed by a set of findings (values for observables) and the output represents a diagnostic category (a fault class) which explains the findings. The simplest model for diagnosis consists of a direct match between findings and solution through classification (Fig. 5.8). A set of findings is input to a *classify* knowledge source and this inference can produce a solution. It uses some body of classificatory knowledge. Findings are generated by obtaining a value (a transfer task) for a selected observable. The *select*

[3]In medicine, the term "differential diagnosis" is used in a similar sense, although this term also tends to imply an ordering of the hypotheses.

observable knowledge source typically selects the observable with the highest information content (i.e. the one that discriminates best between potential solutions) and/or lowest costs.

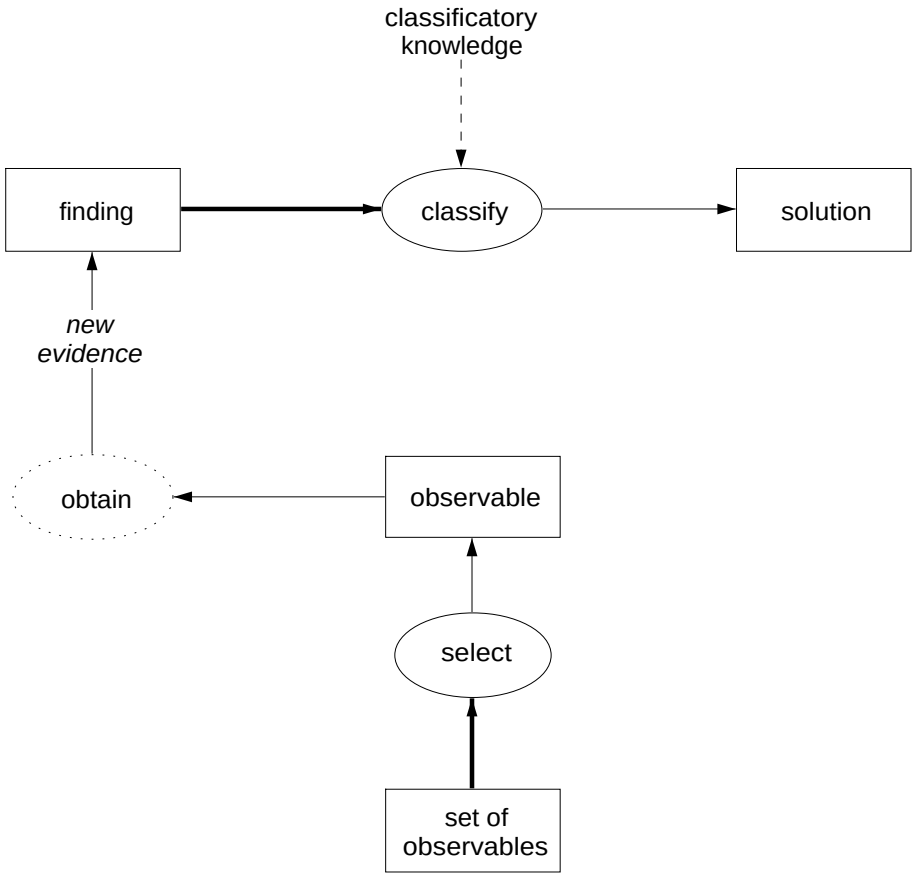

FIGURE 5.8: Diagnosis through direct matching

A typical task structure for this form of diagnosis is given below.

task *diagnosis-through-direct-match*
  control-terms:
    observables: set of all possible observables
    findings: set of all findings currently known
  task-structure:
    REPEAT
      select(observables → observable)
      obtain(observable → finding)
      findings := finding ∪ findings
      classify(findings → solution)
    UNTIL some solution has been found

In [Patil, 1988] it is observed that computational techniques applying Bayes theorem have been used for realising this type of diagnosis. Bayesian techniques have a number of limitations. For each hypothesis knowledge concerning the prior probability and the

conditional probability given each possible finding needs to be available. This implies that large amounts of statistical data are required. Also, this technique is computationally very inefficient unless a number of simplifying assumptions are made, such as mutual exclusivity of potential solutions and conditional independency of findings. These assumptions do not hold in most real-life domains. Another computational drawback is that the complete set of potential solutions is evaluated after every new finding (see the task structure above). Due to these limitations, this simple direct-match model of diagnosis is unsuitable for most diagnostic applications.

5.5.3 Diagnosis through generate-and-test Most diagnostic experts do not evaluate all potential solutions at once. Instead, they build a differential containing only a limited number of *hypotheses*. Typically, the number of hypotheses in the differential is not more than five or six. The hypotheses in the differential are then tested to find out whether additional evidence exists for supporting a particular hypothesis. This hypothetico-deductive approach appears to be a very general method used in problem solving.

If one wants to introduce this idea of generating and testing hypotheses into the model, this implies that the *classify* inference in Fig. 5.8 should be refined into a number of other inferences (i.e. a task-decomposition):

- Inferences for generating hypotheses for which at least some evidence (findings) is present.
- Inferences for testing hypotheses by specifying findings that would support the hypothesis, and subsequently finding out whether these findings are in fact present through obtaining a value for the corresponding observable.

A first inference structure for this generate-and-test approach is shown in Fig. 5.9. The new elements when compared to Fig. 5.8 are indicated with grey boxes and ovals. The *associate* inference generates a new hypothesis, given a finding. The *test* step is realised through the specification of a set of *conjectured findings* (*specify-1*, note the use of the thick line to indicate a set) and the specification of a set of corresponding test-observables for a conjectured finding (*specify-2*).

The generate-and-test approach is usually much more efficient than the direct-match approach as it does not require the evaluation of the complete set of possible solutions (i.e. a computational criterion). Also, the generate-and-test approach is much closer to the way in which humans carry out diagnosis.

Generate-and-test	
Operation	Refinement
Method	Task decomposition
Criteria	Computational

5.5.4 Differentiating findings In the generate-and-test model described in the previous section, every finding can potentially be used to generate a hypothesis. Patil [Patil, 1988] remarks that this often leads to a differential that is too large. For example, in a medical domain general findings such as "headache" could generate a large number of hypotheses.

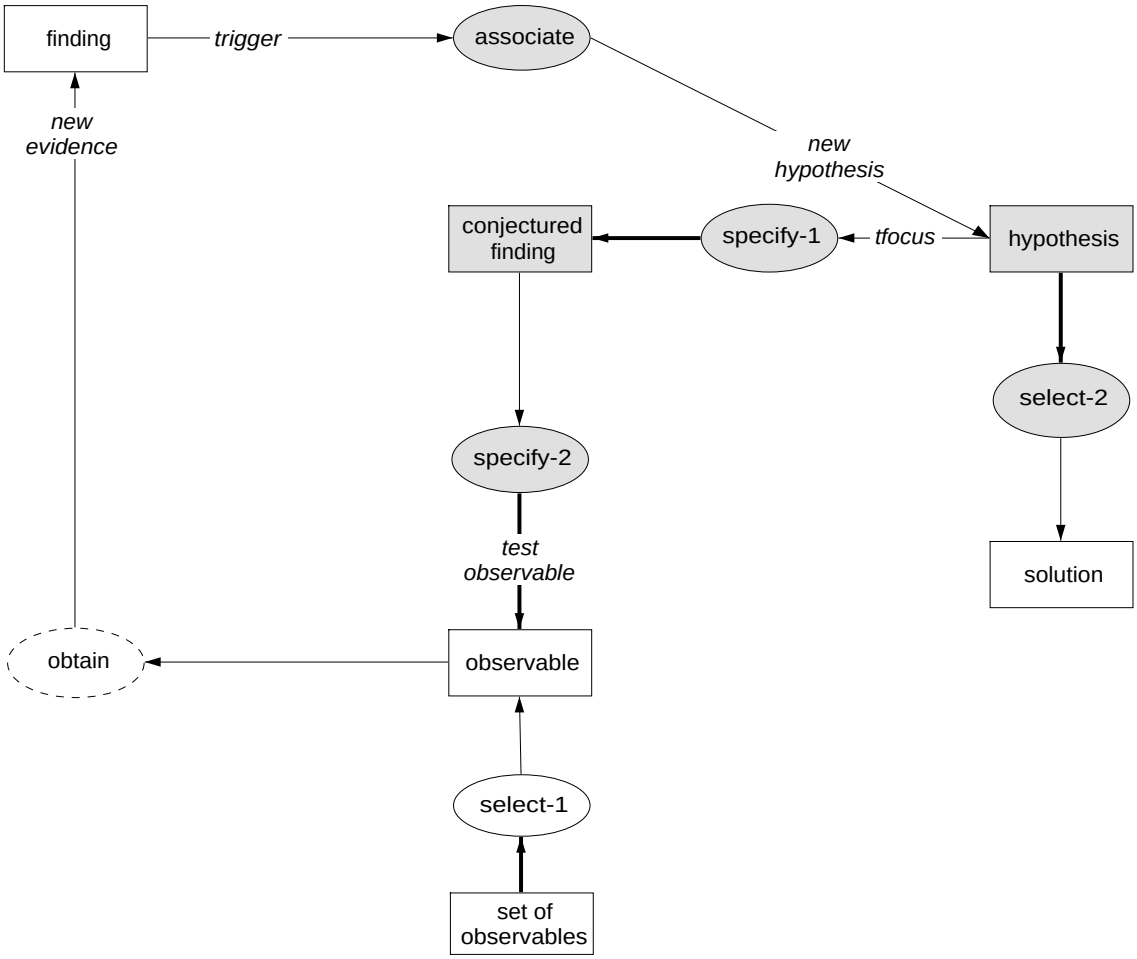

FIGURE 5.9: First inference structure for a generate-and-test approach to diagnosis. New elements are indicated with grey boxes/ovals.

A solution to this problem is to differentiate the set of findings by introducing the notion of *specific finding* and to use only these findings for generating hypotheses. This is similar to the way experts generate hypotheses. The non-specific findings are only used to confirm activated hypotheses.

Fig. 5.10 shows an extension of the previous inference structure. It contains one additional select inference *select-3* which selects a specific finding from the set of findings. Only a specific finding can generate a new hypothesis. Again, the rationale for this modification is of a computational nature: limitation of the size of the differential. It is also a good example of the concept of role-limiting as described in Ch. 2.

Finding differentiation	
Operation	Addition of knowledge source Addition of specialised role ("specific finding")
Method	Knowledge differentiation
Criteria	Computational

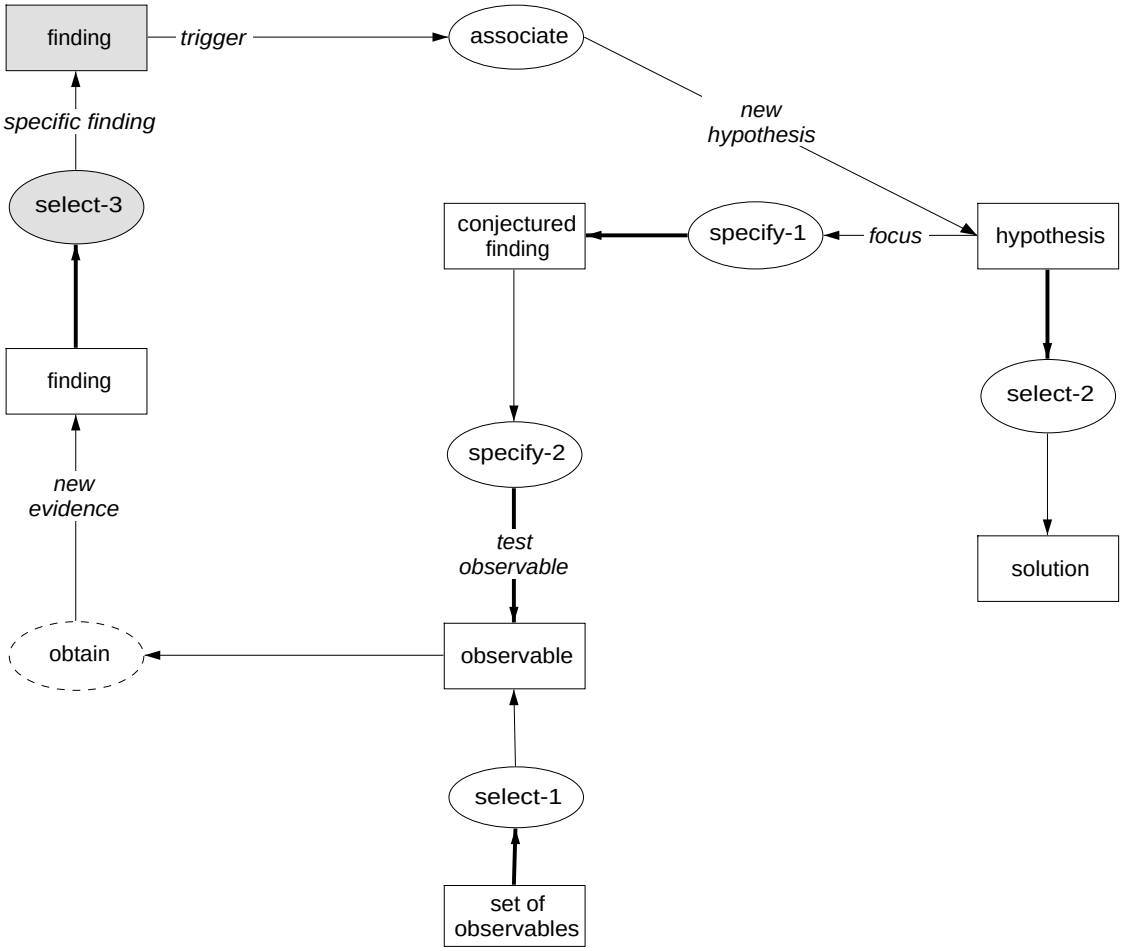

FIGURE 5.10: Differentiating findings

5.5.5 Introducing finding abstractions In many domains, human experts employ data abstraction as a technique for reducing a large data set. Data abstraction is a powerful technique that limits the search space and also reduces the size of the differential. Introducing finding abstraction in the generate-and-test model requires the specification of one additional inference *abstract* which takes as input a set of findings and produces a new, more abstract, finding (see Fig. 5.11). From the task-knowledge point of view, abstraction typically has a recursive structure. An abstracted finding can be the input for another invocation of the abstraction knowledge source.

Clancey describes three types of abstraction [Clancey, 1985b]:

(i) Qualitative abstraction, in which an abstraction is made from a (set of) qualitative findings to a qualitative finding. E.g. a value for the diastolic blood pressure is abstracted into the finding whether the blood pressure is elevated or not.

(ii) Definitional abstraction, in which an abstract name (label) is assigned to a finding. E.g. hypertension is defined as an elevated blood pressure.

(iii) Generalisation, in which several findings are defined as sub-types of a more general finding. E.g. hypertension and edema are both circulatory signs.

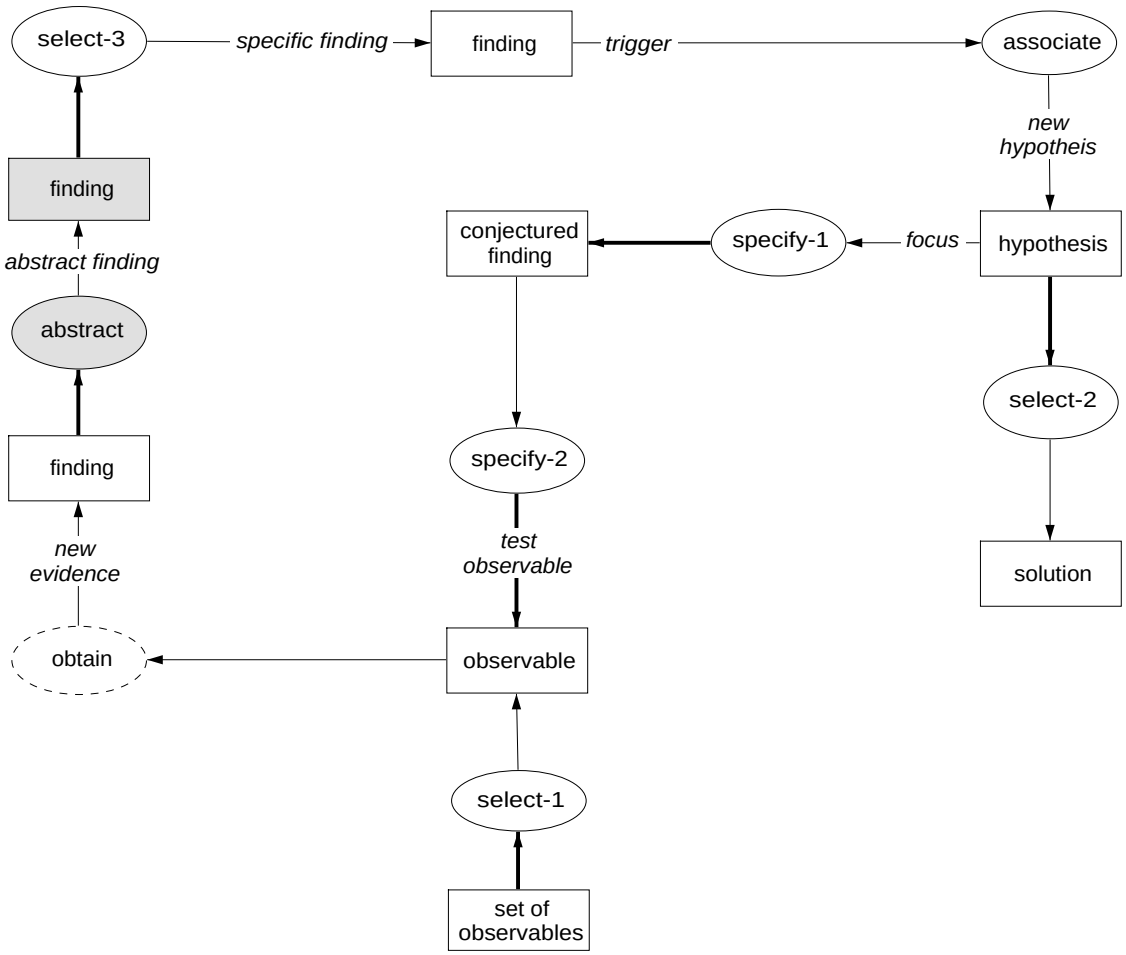

FIGURE 5.11: Introducing finding abstractions

In addition, *syndromatic abstraction* could be considered as a fourth type of abstraction. In syndromatic abstraction, a cluster of findings is treated as one aggregate finding. Clustering of findings is probably the most efficient way of limiting the number of hypotheses in the differential. For example, the combination of physical stress and retro-sternal pain triggers the hypothesis *angina pectoris*. Retro-sternal pain in isolation would generate all ischaemic heart diseases and possibly some additional ones as well.

These four types of abstraction can be seen as a special type of differentiation of the *abstract* inference, which we have called *inference differentiation* (cf. Sec. 5.3). In this case, *abstract* can be differentiated into four sub-inferences, based on the type of domain knowledge used by the inferences (see Fig. 5.12).

Finding abstractions	
Operation	Addition of knowledge source Addition of specialised role ("abstract finding")
Method	Knowledge differentiation
Criteria	Computational

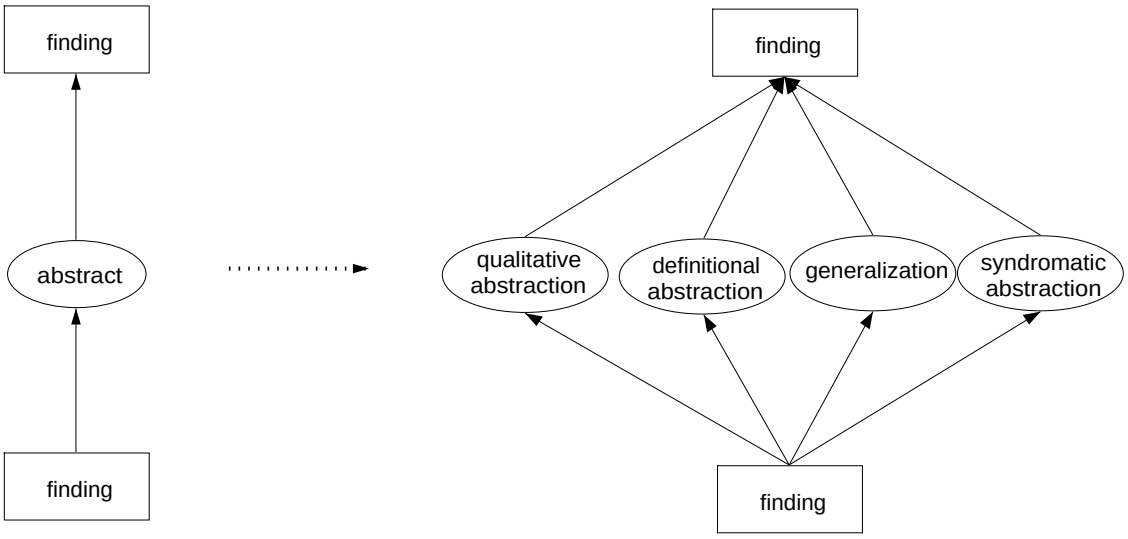

FIGURE 5.12: Differentiating the *abstract* knowledge source into four sub-types.

Abstraction sub-types	
Operation	Refinement
Method	Inference differentiation
Criteria	Epistemic

5.5.6 Hierarchical organisation of hypotheses Yet another way of limiting the size of the differential is to organise hypotheses in a definitional hierarchy (taxonomy) such as provided by KL-ONE [Brachman & Schmolze, 1985]. Such a hierarchy contains general hypothesis categories. Each general hypothesis category specifies commonalities among more specific hypotheses.

Such an organisation has several advantages:

- The differential can be limited in size through the activation of a general hypothesis category, which represents in fact a class of more specific hypotheses.
- If a general hypothesis category is ruled out, then its sub-classes are also ruled out.
- Hierarchies provide a natural way for representing differentiating knowledge: e.g. identifying an observable of which the value would differentiate between alternative hypotheses.

A hierarchical organisation of hypotheses leads to the introduction of three additional inferences in the generate-and-test inference structure (see Fig. 5.13). Two knowledge sources, *refine* and *generalise* constitute operations on the differential. The *refine* knowledge source enlarges the differential by replacing a general hypothesis category with a set of more specific hypotheses. The *generalise* knowledge source can be used to reduce the size of the differential through the inverse process: replacing in the differential a set of specific hypotheses with a more general hypothesis category that subsumes this set. The third knowledge source, *differentiate*, generates a discriminating observable between two or more hypotheses that have a common parent in the hierarchy.

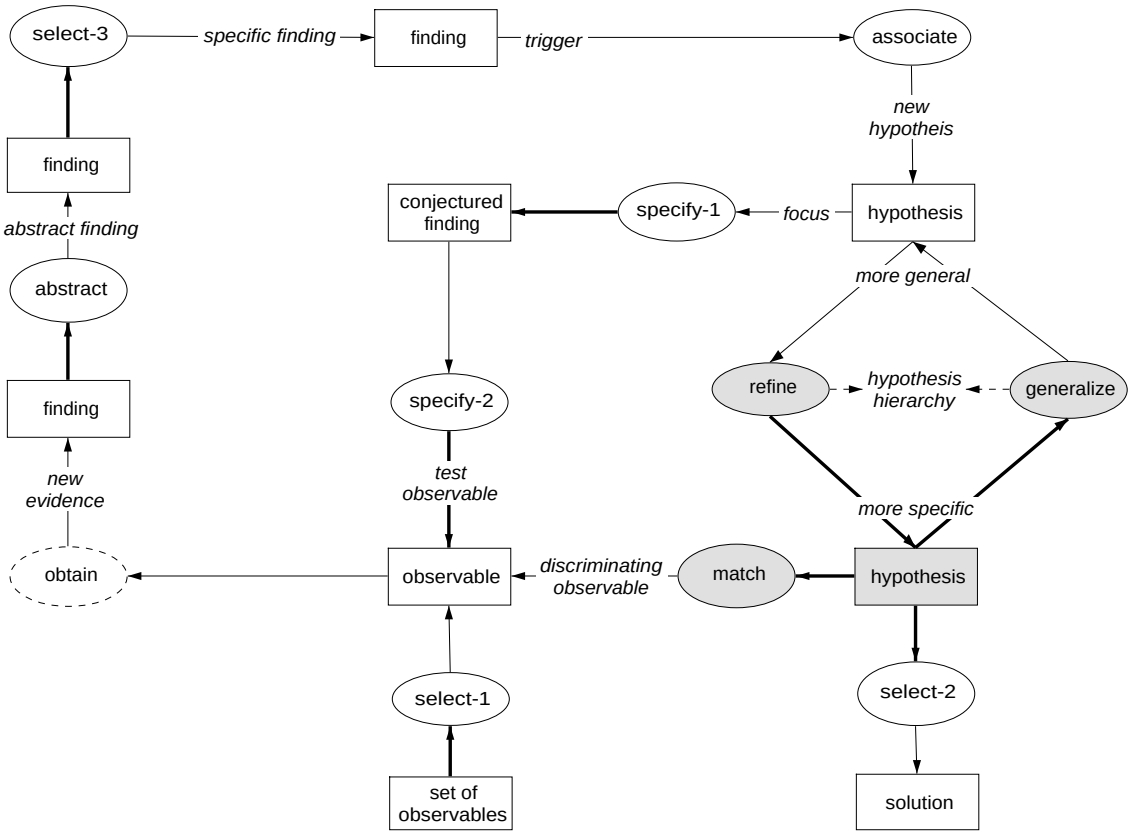

FIGURE 5.13: Hierarchical Organisation of Hypotheses

In principle, a definitional hierarchy is a very powerful and efficient organisation of hypotheses. A major problem is however that in many domains it is not possible to define one single natural hierarchy of hypotheses. For example, in the disease hierarchy in NEOMYCIN [Clancey, 1985a] the distinction between levels is based on four dimensions along which diseases can be classified:

1. process type (e.g. infection)
2. localisation (e.g. meningitis)
3. time factor (e.g. acute meningitis)
4. etiology (e.g. acute bacterial meningitis)

Each ordering of these dimensions is however somewhat arbitrary. Any ordering implies that not all useful general disease categories are available in the hypothesis hierarchy. For example, given the ordering in the list above, the hypothesis "acute infection" cannot be represented. This can be repaired by reversing the localisation and time-factor level, but that modification would imply that we lose the general disease category "meningitis" (see Fig. 5.14).

In domains where there are no natural hierarchies, knowledge engineers often keep reorganising the hierarchy, but are unable to find a satisfactory organisation exactly for the reasons given above. In the CADUCEUS system an attempt is made to overcome this

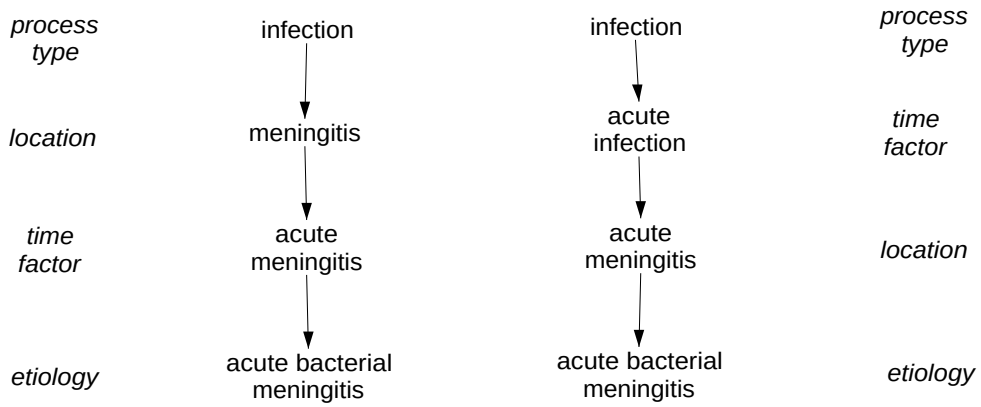

FIGURE 5.14: Two possible hierarchy organisations, given the dimensions process type, localisation, time, and etiology.

problem through an organisation of hypotheses along various dimensions [Pople, 1982]: e.g. a time-directed hierarchy, an etiological hierarchy, etc. Additional inferencing is in that case necessary for combining the (partial) classifications derived from the different hierarchies.

Hierarchical organisation	
Operation	Additions
Method	Knowledge differentiation
Criteria	Computational

5.6 A KADS Inference Structure for Heuristic Classification

The model construction process described in the previous section resulted in what one could call a KADS inference structure for the model of heuristic classification (HC) as realised in the NEOMYCIN system [Clancey, 1985a]. NEOMYCIN contains in fact one additional inference used in the "clarify finding" task: the specification of a number of observables that are dependent on a known finding. For example, if the finding "chest pain = present" becomes known, then this inference is able to specify related observables such as the duration, nature and radiation of the pain. The full inference structure for HC is shown in Fig. 5.15.

This figure contains more detail than the "horse-shoe" figure (see Fig. 5.16). This last figure is considered by many as an equivalent of a KADS inference structure.[4] We think that this interpretation of the horse-shoe figure is incorrect. Clancey's figure describes dependencies between the main functional objects (data and solutions) in a more global way than is required in KADS inference structures.

The main refinements in Fig. 5.15 when compared to the horse-shoe figure are:

- The specification of some inferences in inverse directions:
  - *specify conjectured finding* and *match*: from a solution (or a solution abstraction) to data (or data abstractions)

[4] It was also included (in a slightly different form) in the interpretation model library [Breuker *et al.*, 1987].

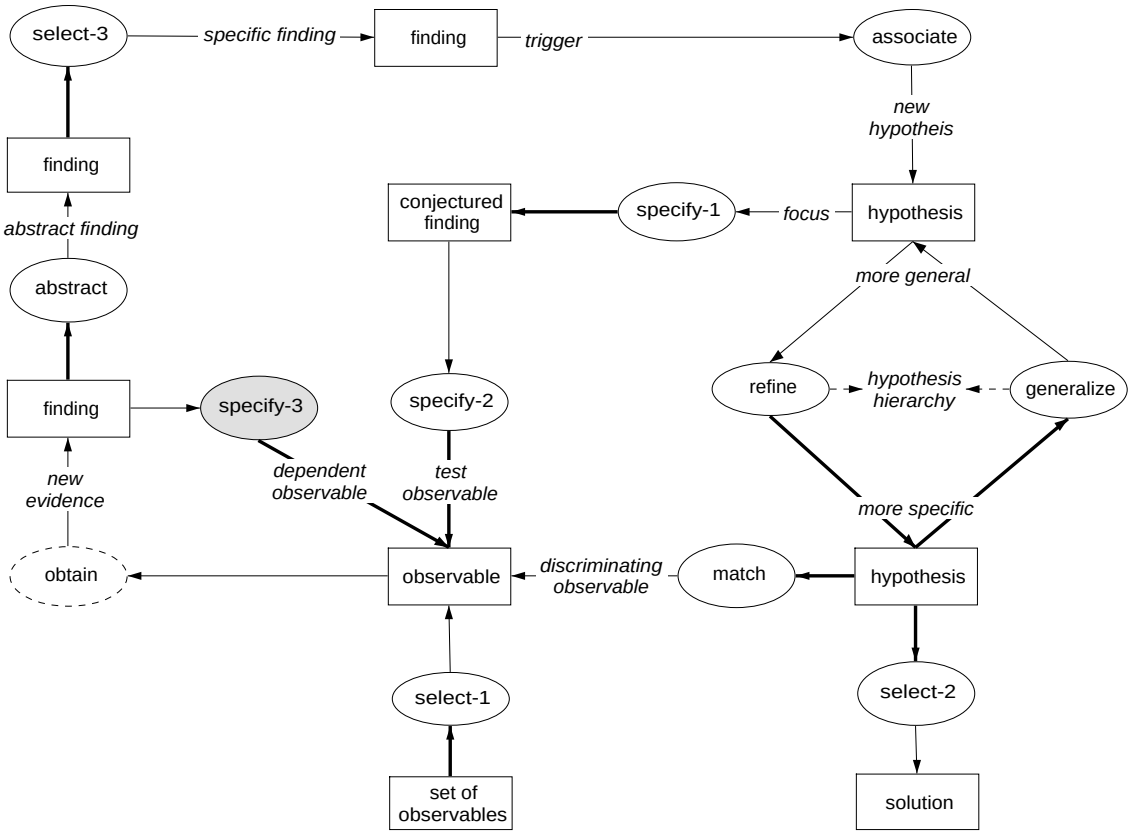

FIGURE 5.15: KADS inference structure for heuristic classification as implemented in NEOMYCIN.

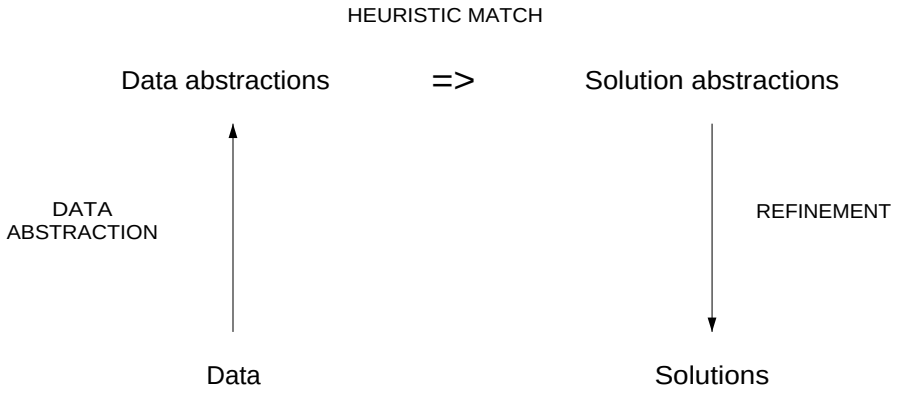

FIGURE 5.16: Clancey's "inference structure of heuristic classification", also often called the "horshe shoe". Source: [Clancey, 1985b; p. 296]

  - *generalise hypotheses*: from solutions (or solution abstractions) to a (more general) solution abstraction.

- The explicit distinction between *findings* and *observables*. This distinction allows one, for example, to pinpoint the basic inference used in NEOMYCIN's *generate questions* task (a task which goal is to get new evidence, when the current set of data is insufficient to arrive at a solution). This task applies the selection of an observable from the universe of observables known to the system (*select-1* in Fig. 5.15).

As remarked before, KADS knowledge sources constitute a generalisation of the use of domain knowledge. One interesting point, that arises from observations about the KADS inference structure in relation to the analysis of heuristic classification in [Clancey, 1992], is that this inference structure can be used to generate a situation-specific model (SSM) such as advocated by Clancey.

An example of such a KADS SSM is given in Fig. 5.17. This example describes a SSM for the trace of NEOMYCIN's reasoning given in [Clancey, 1992; p. 18] (repeated for convenience in Table 5.1). This particular trace shows how, given a hypothesis "meningitis", the system tries to find supporting evidence (i.e. whether the patient has experienced "seizures"). This new finding leads to a focus shift: it triggers a new hypothesis ("increased intercranial pressure"), which in turn starts a process of establishing evidence for this hypothesis.

Fig. 5.17 shows a KADS version of the SSM generated by this example. The numbers in the figure indicate the ordering in which the various arcs where put in the model. The main difference between the KADS SSM and the SSM's in [Clancey, 1992] is that the relations between the nodes in the KADS SSM are labeled with inference vocabulary: knowledge source and meta-class names. In [Clancey, 1992] domain rules provide the relations, with an additional annotation of the task that invoked the rule. One can view SSM's as a particular kind of "knowledge-level" trace of the reasoning process.

The KADS SSM and Clancey's SSM's are in fact complementary. Together these provide three different and important viewpoints on the rationale behind the reasoning process:

- The domain-knowledge view point: what domain knowledge is used?
- The inference view point: what kind of derivation is made and what is the role of the object being manipulated?
- The task view point: what is the goal that is being pursued with this reasoning step?

5.7 Discussion: Generic Model Components

The question arises whether a top-down model construction process as described above can be supported by template model components. This would require a different organisation of the library of template models, namely not as a flat set of interpretation models but as a set of generic model components of a smaller grain size. Such model components could be inserted into a model and result in a more complex model.

The two refinements of the model for systematic diagnosis (Sec. 5.4) can be viewed as examples of such model components. Model components can be identified also on a more general level. If one studies the inference structures of systematic diagnosis (Fig. 3.5) and of monitoring (Fig. 3.8), it becomes clear that these share a common set of related inferences, namely the process of checking the expected value of a parameter against the

{1.Top of the line of reasoning: we are pursuing meningitis as a generalization of some hypothesis triggered by the initial data.}

CONSULT
MAKE-DIAGNOSIS
COLLECT-INFO
ESTABLISH-HYPOTHESIS-SPACE
GROUP-AND-DIFFERENTIATE
TEST-HYPOTHESIS [Meningitis]
APPLY-RULES [Rule060, Rule 323]
APPLY-RULE [Rule060]

{2. After finding out about seizures to apply rule 60, we consider other data-directed inferences: the follow-up question (#9) about seizures duration is generated; the rule 262, marked "antecedent", is applied.}

FORWARD-REASON
PROCESS-FINDING [Seizures]
APPLY-RULES-ANTE [Rule262]
APPLY-RULE [Rule262]

{3. Rule 262 concludes that seizures might also be caused by increased intercranial pressure: is that linked to anything else we have been considering? It might be explained itself by an intercranial mass lesion, but more evidence is required before the rule can be applied. Test-hypothesis is now invoked recursively: a focus change has occurred.}

FORWARD-REASON
PROCESS-HYPOTHESIS [Increased-Intercranial-Pressure]
APPLY-RULES-ANTE [Rule239]
APPLY-RULE [Rule239]
FINDOUT [Increased-Intercranial-Pressure]
TEST-HYPOTHESIS [Increased-Intercranial-Pressure]
APPLY-RULES [Rule209, Rule233, Rule373]
APPLY-RULE [Rule209]

{4. Rule 209 requires information about papilledema; the inquiry is generalized to fundoscopic abnormal, question #10.}

FINDOUT [Papilledema]
FINDOUT [Fundoscopic-Abnormal]

TABLE 5.1: Trace of a consultation of NEOMYCIN. Source: [Clancey, 1992; p. 18].

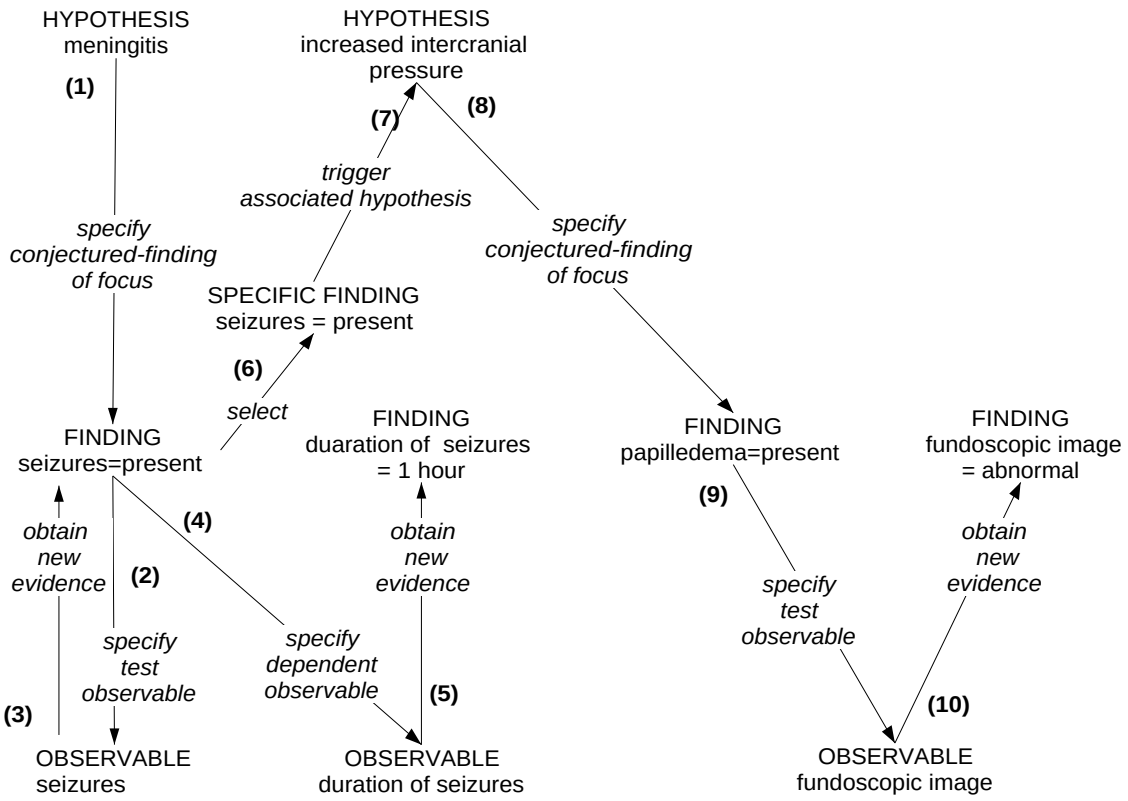

FIGURE 5.17: Situation-specific model for the trace of NEOMYCIN shown in Table 5.1 using the KADS inference structure of HC. The numbers indicate the sequence in which the arcs where placed in the SSM.

observed value. Fig. 5.18 shows this set of inferences. One could view this set as a potential generic model component.

Representing template models in the form of such generic components is attractive because it captures the way in which knowledge engineers actually build these models. The identification of such generic model components can help making a number of aspects of the model construction process more explicit, namely:

- The ingredients (model components) from which a model is built.
- The rationale behind the inclusion of a particular component (e.g. reducing the size of the differential).
- The domain-knowledge requirements of model components. For example, *refine* and *generalise* require a particular hierarchical organisation of hypotheses.

In Table 5.2 an effort is made to describe the various modifications in the top-down construction of the model for diagnosis (Sec. 5.5) in terms of model components.

The components described in Table 5.2 have a number of features in common with Chandrasekaran's generic tasks [Chandrasekaran, 1988]. Their grain size is similar. For example, "hypothesis generation" could be realised through the GT "abductive hypothesis assembly". The GT "knowledge-directed information passing" can be used for "finding abstraction". The main difference is that the GT's are tied to a particular symbolic representation.

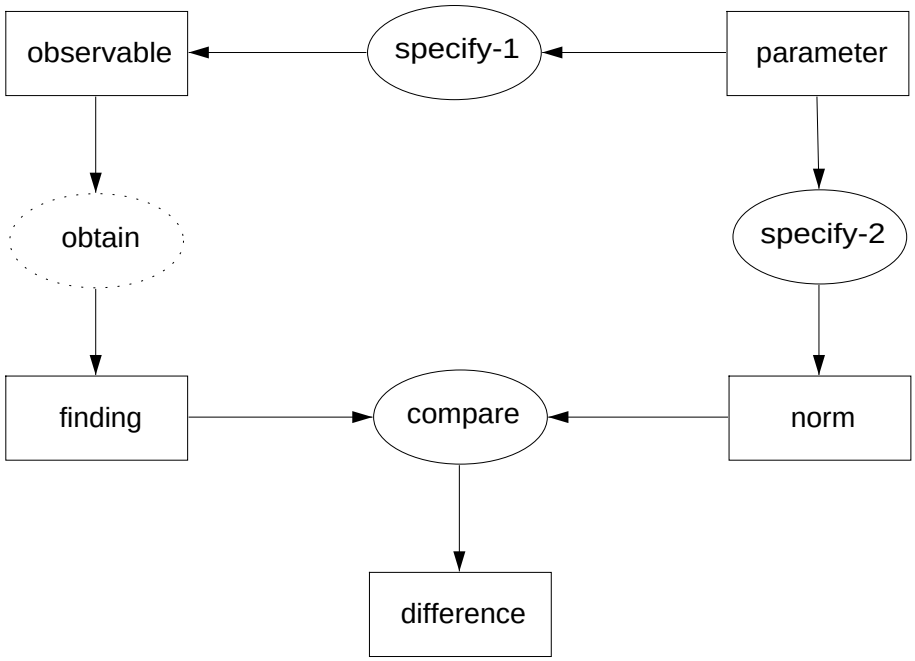

FIGURE 5.18: Example model component: checking the expected value of a parameter against the observed value.

Component	**Fig.**	**Knowledge sources**	**Rationale**
question generation	5.8	select({O} → O) obtain(O → F)	generates (new) input data
hypothesis generation	5.9	associate(F → H)	put new hypothesis in differential
hypothesis testing	5.9	specify(H → {F}) specify(F → {O}) select({H} → H)	find evidence such that a solution can be found
finding differentiation	5.10	select({F} → F)	specific finding triggers less hypotheses
finding abstraction	5.11	abstract({F} → F)	reduction of search space
differential reorganisation	5.13	refine(H → {H}) generalise({H} → H)	reducing/enlarging differential
hypothesis differentiation	5.13	match({H} → O)	find observable that discriminates between two or more hypotheses

TABLE 5.2: Summary of the model components used in the differentiation of the generate-and-test-model. The symbols 'O', 'F' and 'H' denote respectively an observable, a finding and a hypothesis. The '{..}' notation denotes a set.

The Generalised Directive Models (GDM's) as proposed in the Acknowledge project [van Heijst *et al.*, 1992] support a similar top-down approach to model construction. The grammar in which these GDM's are expressed can be used to carry out refinement operations on (provisional) inference structures.

In principle, a library of generic model components would allow the knowledge engineer to derive in a top-down fashion the inferences needed in an application domain. For example, in the construction process that led to the HC model, the knowledge engineer could decide for some application to leave out the inferences related to a hierarchical organisation of hypotheses, when such hierarchies cannot be found or constructed. In other domains, abstraction could turn out to be unnecessary or testing of hypotheses could be carried out through causal models.[5]

However, much work still needs to be done to support the use of generic components in top-down model construction in a principled way. It would require at least:

- The construction of a comprehensive library of generic model components.
- A description of decision criteria that would lead to including particular components in the inference structure.
- A set of composition rules for configuring and modifying inference structures from smaller components.

The grammar for Generalised Directive Models developed in the Acknowledge project [van Heijst *et al.*, 1992] provide a first step to this type of support. The further exploration of this approach to model construction is currently a major research topic in the KADS-II project.

[5] Although the relations between a hypothesis and a set of corresponding findings are called *causal* relations in NEOMYCIN, these should be viewed as direct associations and do not represent a causal model in the usual sense of the word.

Chapter 6

Operationalising Models of Expertise

Knowledge-level models currently play an important role in the development process of knowledge-based systems. In this chapter we investigate issues concerning the design and implementation ("operationalisation") of such models. We characterise the nature of the KBS design process and distinguish various types of decisions that have to be made. We define structure-preserving design, i.e. preserving the structure of the knowledge-level model in the artefact, as the principle that should underly the operationalisation process, because it facilitates reusability of code, maintenance, explanation and knowledge refinement. We discuss several existing environments that support the operationalisation process and outline their drawbacks. We sketch an alternative route for computerised support and illustrate this for a class of diagnostic tasks.

This chapter will be published in a collection of articles on KADS. Reference: Schreiber, A. T. (1993). Operationalising models of expertise In Schreiber, A. T., Wielinga, B. J., & Breuker, J. A., editors, *KADS: A Principled Approach to Knowledge-Based System Development.* Academic Press, London.

6.1 Introduction

Model-based development has become over the last few years the prevailing paradigm for knowledge-based system (KBS) construction. With "model-based" a development approach is denoted in which a prime role is played by a "knowledge-level" [Newell, 1982] model of the problem solving behaviour in an application domain.[1] In this chapter we investigate the problem of how to *operationalise* such knowledge-level models in the process of KBS construction.

Model-based KBS development consists of at least two different types of activities (see Fig. 6.1):

1. *Knowledge modelling* activities aimed at constructing a knowledge-level model of the application.

2. *Design & implementation* activities aimed at operationalising a particular knowledge-level model through the selection and implementation of appropriate computational and representational techniques. In this process requirements not directly related to problem solving are also taken into account (e.g. efficiency).

[1] On-going debates on the precise nature of the knowledge-level and its role in KBS development are discussed in Ch. 2.

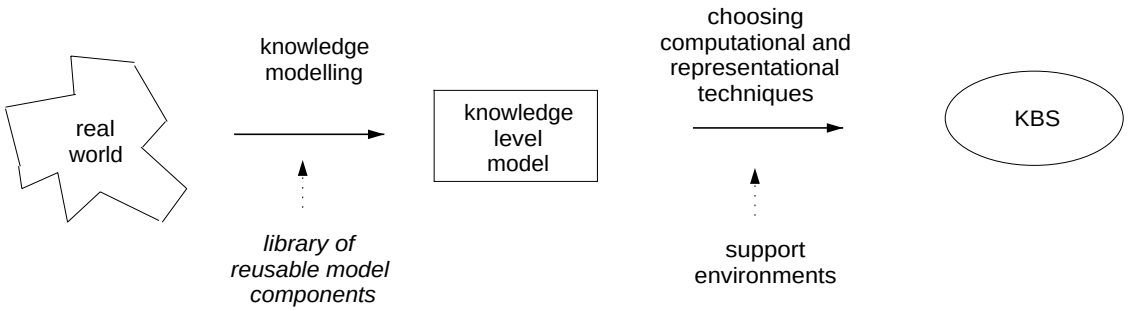

FIGURE 6.1: A bird's eye view of KBS construction based on knowledge-level models

In this chapter we focus on the latter part of the KBS development process which could be called the "operationalisation problem". Generally, we use the term "operationalisation" to denote the process of designing and implementing the final system. However, at some points we will also take approaches into account that aim at "making knowledge-level models run": prototype systems used for validation and knowledge refinement.

Sec. 6.2 discusses the nature of the KBS design process and characterises the various types of decisions that have to be taken during design. In Sec. 6.3 we discuss structure-preserving design as a general principle that should underly the entire design process. In Sec. 6.4 and Sec. 6.5 we discuss how this principle influences the design decisions when operationalising KADS models of expertise.

Computerised support is an important issue in a KBS development methodology. Sec. 6.6 describes existing environments for supporting this operationalisation process and discusses their drawbacks. In Sec. 6.7 an alternative approach is suggested. This approach is illustrated through an example support environment that we developed. Sec. 6.8 discusses results and future work.

6.2 The Design Process

The major input for the design process in KADS is the model of expertise, which can be viewed as a specification of the problem solving requirements. Other inputs are user interaction requirements (i.e. the model of cooperation, see Ch. 3) and also a set of external requirements, such as costs, software and hardware.

The nature of the design process can be characterised by dividing it into a number of interrelated design decisions that have to be made during the design process. We distinguish two major groups of design decisions: (i) decisions with respect to the overall system *architecture* and (ii) decisions with respect to the selection of suitable *computational techniques*.

6.2.1 Architectural options In the literature the term "architecture" is used in many different ways. We use the term to denote a global description of the main components of the artefact to be developed and their inter-relations. The nature of the prime ingredients described in an architecture varies depending on the *architectural paradigm* that is being used. Such a paradigm prescribes what the building blocks are from which the artefact will be assembled and what relations exists between them. An architectural

paradigm can also prescribe how the analysis input should be mapped onto the architecture. Two well-known architectural paradigms in software-engineering are the functional approach [Yourdon, 1989b] and the object-oriented approach [Coad & Yourdon, 1991; Rumbaugh *et al.*, 1991]. In the functional approach the prime architectural components are functions. Functions are linked to each other via data flows. In the object-oriented approach the main components are objects. Objects are related to each other via associations (relations) and message connections. Another architectural paradigm that has been proposed for certain AI programs is the so-called "blackboard" architecture. In the latter paradigm the emphasis lies on the distribution of control in the system.

In fact, the three example paradigms described above symbolise what appear to be three fundamental perspectives that one can take when describing a system [Rumbaugh *et al.*, 1991; Yourdon, 1989a] (see also Ch. 8), namely:

- the *data perspective*,
- the *functional perspective*, and
- the *control perspective*.

These three perspectives can be summarised as the "what, how, and when" views of a system. Choosing one of the paradigms does not mean that only this particular type of information is present in the architecture. For example, in the functional approach the data manipulated by functions must be described as well. Also, in the object-oriented approach operations (functions) and messages (control) must be defined for each object. The object-oriented approach in fact groups data, functions and control together in small untis. The main distinguishing factor between the approaches is the *decomposition principle* that is employed when describing a system. In the functional approach the functional perspective provides the decomposition principle: the system is decomposed into a hierarchical structure of functions and sub-functions. In the object-oriented approach the data perspective provides the entry point: system decomposition starts with building hierarchies of data objects.[2] Many of the debates in software engineering have been about the "right" decomposition principle. Later in this chapter we will argue for an architectural paradigm that represents a combination of these three perspectives. This is in line with recent proposals in software engineering, such as advocated by [Rumbaugh *et al.*, 1991].

Thus, the specification of the architecture entails two steps:

Choice of the architectural paradigm The choice of the perspective that guides the decomposition in the architecture: e.g function-oriented, object-oriented. In real-life practice this paradigm choice is often not noted as an explicit design decision. It is often determined by the background and experience of the system designer or by software-engineering 'fashions".

Architecture specification Given an architectural paradigm, the designer will have to use the analysis input (the conceptual model) to specify a suitable architecture. From a methodological point, this activity can be supported by providing the system designer with *skeletal architectures*: prototypical decompositions that have to be instantiated for a particular application. In this chapter we will give two examples of skeletal architectures that could be useful for "KADS" designers.

[2] This statement is not completely true for recent developments in the field of so-called 'user-interface management systems" such the kernel of the KEW workbench [Anjewierden, 1991].

6.2.2 Computational options Given an architecture, the designer will have to decide which computational techniques she is going to employ in the artefact. The nature of these decisions can be illustrated with an example concerning the skeletal architecture used in the early days of expert systems. This architecture (Fig. 6.2) was a very simple, naive one. The two main components of a KBS were in this view an "inference engine" and a "knowledge base". It is in fact a control-oriented architecture: the leading principle for the decomposition is the control relation between a declarative component (the knowledge base) and a procedural component (the inference engine) of the system.

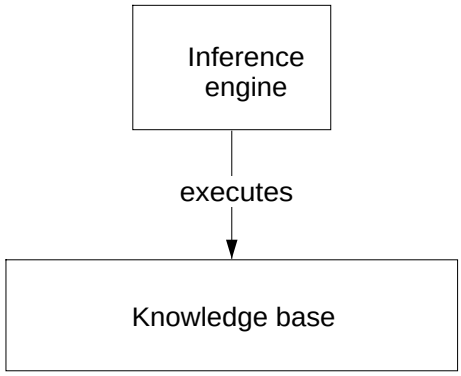

FIGURE 6.2: Naive expert-system architecture

The computational design decisions that have to be taken in this architecture are typically concerned with:

- the representation formalism for the knowledge base,
- the algorithm for interpreting the knowledge base, and
- the control regime for sequencing interpretations of the knowledge base.

Typically, one type of technique was selected for realising this architecture: e.g. a production system technique (e.g. production rules, rule interpreter with forward chaining, some form of conflict resolution) or an automated deduction technique (e.g. predicate calculus representation and a theorem prover).

One can view production systems and automated deduction as *computational paradigms*: they point to a group of related techniques and representations. Other well-known computational paradigms are *state-space search*, *parsing*, *constraint satisfaction* and *structure matching*.

Employing techniques that belong to only one particular paradigm simplifies the process of KBS design. In general however, one would not like to limit the designer to use one particular group of techniques. Some techniques are better suited for solving certain (sub-)problems than others.

In short, the designer has to make two types of computational decisions: (i) the choice of the computational paradigms that will be employed in the artefact, and (ii) the specification of the techniques that will be used for realising the various elements in the system architecture.

In Newell's terms, the computational techniques describe the symbol-level realisation of the agent. It is important to realise that there is a trade-off between (i) making the

knowledge-level model more specific, i.e. by introducing more refined elements, and (ii) a more elaborate specification during design. The decision whether to do the former or the latter depends on whether one needs to represent explicit control on certain operations in the model of expertise. The borderline between model of expertise and design is thus in a sense governed by the level of *granularity* that is required of the model of expertise. For example, in OFFICE-PLAN (a system for allocating offices to employees, see [Karbach *et al.*, 1989]) the actual allocation inference is modelled as one knowledge source *assign* in the model of expertise and is realised in the actual system through a complex constraint satisfaction technique. If it would have been necessary to exercise control on this technique, then one would need to model constraint satisfaction "at the knowledge level".

6.2.3 Choosing a software environment In addition, the designer has to choose (or construct) a software environment that is to be used for implementation. With a software environment we mean some programming language (with possibly additional components, e.g. libraries) that is used as the basis for implementation: e.g. KEE [Fikes & Kehler, 1985], EMYCIN [van Melle *et al.*, 1981], Prolog, LISP, etc. A software environment supplies the designer with a number of predefined (=implemented) computational constructs and representations. In [Schreiber *et al.*, 1987] a distinction is made between *closed* and *open* environments.

In a closed environment the set of available techniques and representations is fixed and not expandable. An example of such an environment is EMYCIN. An open environment offers possibilities for expanding the set of methods and representations. Open environments can be further divided into *weak* or *strong* environments, depending on the size of the set of predefined techniques. Prolog can be viewed as an example of a weak environment. Its built-in facilities are very general-purpose: unsorted Horn clause logic, unification and SLD resolution. KEE can be seen as an example of a strong environment. It provides higher-level primitives for implementing computational techniques, such as frame and production rule representations, hierarchical structuring of frames and rules, a classifier and various interpreters,

There is a clear dependency between the computational decisions on the one hand and the choice of an environment on the other hand. The techniques chosen can influence the choice of an environment and vice versa. Ideally, the chosen environment should offer primitives for realising a large variety of techniques and thus limit the amount of implementation effort. In a weak environment such as Prolog or LISP, it is necessary to build the techniques on top of the language.

It should also be noted that in real-life KBS development the choice of a software environment is often dictated by external requirements: costs, availability, etc. This environment then acts as a constraint on the overall design process: e.g., if, for some external reason, a system like EMYCIN has to be used, this implies that the designer has only limited freedom in making computational decisions or has to devise clever ways of implementing the desired constructs using the limited means that the environment offers. A typical example of this last phenomenon is the use of parameters to represent control implicitly in MYCIN [Clancey, 1983].

6.2.4 Overview of the design process Fig. 6.3 summarises the main steps that have to be taken during KBS design. The arrows in the figure denote dependencies between steps

in design. This does not mean that the actual process in time should follow the direction of the arrows. An example of this was given at the end of the previous section, namely that a early (because fixed) choice of an particular environment constrains other design steps.

The last step in Fig. 6.3 is the actual implementation of the system. The nature of this step depends very much on the way the design was carried out and on the chosen software environment. In Ch. 7 we discuss a full sample implementation based on the design principles outlined in the remainder of this chapter. The implementation can be supported in various ways. One important support tool for implementation given a "structure-preserving design" (see the next section) are transformational tools that map model-of-expertise descriptions on code fragments. Another useful tool is a dedicated domain-knowledge editor, which uses a symbol-level translation of the "domain schema" (see Sec. 3.4.1) to interact directly with an application expert. This type of functionality is offered by the OPAL knowledge acquisition tool [Musen *et al.*, 1987].

6.3 Structure-Preserving Design

Design thus consists of the specification of an system architecture and the selection of appropriate representations and computational techniques. In principle, the designer is free to make any set of design decisions that results in meeting the requirements formulated during analysis. However, from a methodological point of view a *structure-preserving design* should be strongly favoured. With "structure-preserving" we mean that the *information content and structure* present in the knowledge-level model is preserved in the final artefact. For example, it should be possible to reconstruct from the final system both the domain-knowledge structures specified during analysis as well as their relations with knowledge sources and/or meta-classes. Design thus should be a process of *adding* implementation detail to a knowledge-level model. The knowledge-level model is in fact interpreted as a skeletal architecture for the system.

Thus, preservation of information is the key notion. For this purpose, we investigate in the next section in more detail the nature of the information in the model of expertise. In Sec. 6.3.2 we discuss the main advantages of the structure-preserving approach.

In Sec. 6.4 we take the "executable specification" view on the model of expertise and try to define skeletal architectures for supporting structure-preserving design. In Sec. 6.5 we investigate how the structure-preserving approach influences the computational decisions.

6.3.1 Types of information in the model of expertise

In this section we take a second look at the structure of the model of expertise and characterise what kind of information it contains with respect to the three perspectives outlined in Sec. 6.2 (data, function, control).

Functional perspective The functional perspective is represented both in the inference knowledge and in the task knowledge. Knowledge sources constitute the leaves of the functional decomposition tree. Tasks represent the higher-level functions. An example of such a functional-decomposition tree can be found in Ch. 3 (Fig. 3.6).

Data perspective The domain knowledge represents a specification of all domain-specific data and forms the major part of the data perspective.

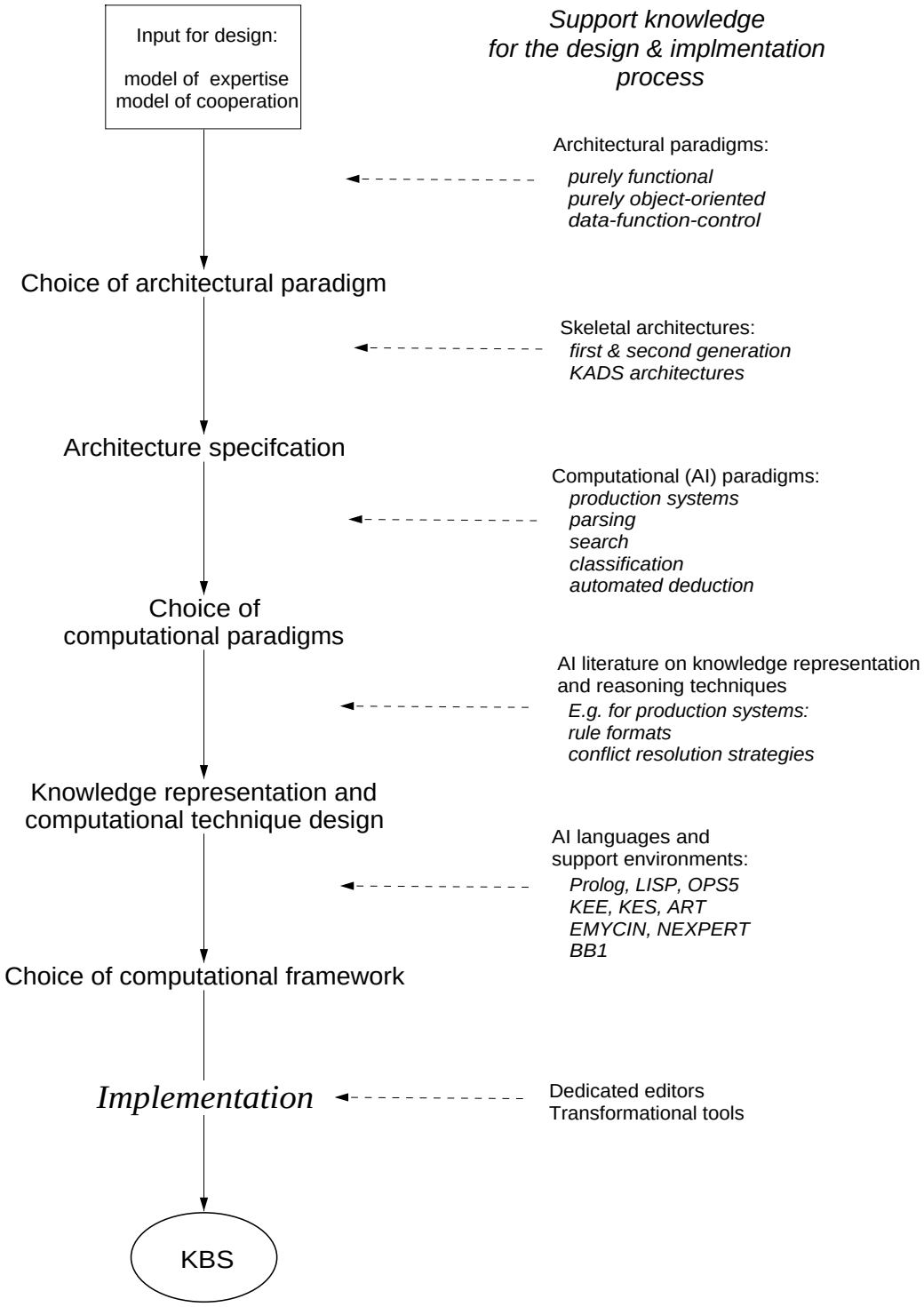

FIGURE 6.3: Dependencies between steps in the design and implementation process. For each step some sample support knowledge or support tools are listed.

The meta-classes and the domain view in the inference knowledge are also part of the data view, but these data are specified in an indirect way: i.e. by mapping functional names onto domain-knowledge elements. One could say that the name of a meta-class specifies a functional object (e.g. *hypothesis*) and that the *mapping* onto domain terms (e.g. *hypothesis* $\rightarrow$ domain class *disease*) assigns a data structure to such a functional object. The mapping provides in fact the link between the functional perspective and the data perspective. The situation in KADS is thus more complicated than in conventional data-flow diagrams, where the names of the data flows refer directly to elements of the data model.

The control terms (part of the task knowledge) specify similar functional objects, but through a multi-step mapping (e.g. the control term *differential* maps onto a set of objects of type *hypothesis*, which in turn maps onto a class of domain objects.)

Control perspective The control perspective (the "when" view) is present in the task knowledge (the task structure description of a task) and in the strategic knowledge.

By relating a description of control in a task-structure procedure directly to a task, an explicit link is made between the functional perspective and the control perspective. This is not by definition true for the control specified in strategic knowledge, as KADS provides no strict guidelines for the structure of this type of knowledge,

Table 6.1 summarises the information contained in the various parts of the model of expertise with respect to three viewpoints. In Ch. 8, we analyse how KADS relates to more conventional approaches with respect to these three perspectives.

Category	Element	Perspective
domain knowledge	concepts, properties relations, structures, etc.	data data
inference knowledge	knowledge source meta-class domain view	functional functional $\rightarrow$ data functional $\rightarrow$ data
task knowledge	control terms sub-tasks task structure	functional $\rightarrow$ data functional control $\rightarrow$ functional
strategic knowledge		control

TABLE 6.1: Characterisation of the information contained in a model of expertise with respect to the three perspectives on systems: data, function and control.

The model of expertise is thus not biased towards one particular perspective, although part of the functional view, i.e. the inference structure is often the starting point for building the model. The resulting architecture is thus a mixture of different types of components, each emphasising different perspectives of the system. This point is discussed in more detail in the comparison of KADS with conventional approaches (Ch. 8).

6.3.2 Advantages of structure-preserving design There are a number of reasons for following a structure-preserving approach to design:

Reusability of code Structure-preserving design prepares the route for reusability of code fragments of a KBS, because, in Smith's terms (see Ch. 1), the "semantical attribution" [Smith, 1985] of code fragments is explicit. Reusable code fragments can be of various types and grain size, ranging from implementations of inferences (knowledge sources) to implementations of an aggregation of inferences plus control knowledge. The layered structure of KADS models of expertise facilitates this type of reusability.

Maintenance The preservation of the structure of the knowledge-level model makes it possible to trace an omission or inconsistency in the implemented artefact back to a particular part of the model. This considerably simplifies the maintenance of the final system. It also facilitates future functionality extensions. Experience with the Fraudwatch system [Porter, 1992; Killin, 1992] indicates that systems built in this fashion are much easier to maintain than conventional systems.

Explanation A structure-preserving approach facilitates the development of explanation facilities that explain the reasoning process in the vocabulary of the knowledge-level model. For example, for some piece of domain knowledge it is possible to ask:

- in which elementary problem solving steps it is used and which role it plays in this inference;
- when and why it is used to solve a particular problem (control knowledge).

As the knowledge-level model is phrased in a vocabulary understandable for a human observer, a structure-preserving design can provide the building blocks for "sensible" explanations. This feature has been demonstrated by [Clancey & Letsinger, 1984] in the NEOMYCIN system. Several researchers have proposed generic strategies for generating such explanations from knowledge-level descriptions [Neches *et al.*, 1985; David & Krivine, 1990; Sprenger, 1991]. Generic explanation tools supporting questions like *why/when/how inference* by exploiting the structure-preserving property of the code.

In EES [Neches *et al.*, 1985] the model information is preserved in a separate "development history" generated by a program writer. This database stores facts about the relation between model elements and code fragments (i.e. OPS5 rules) generated from these model elements. This "compiler" approach to information preservation is attractive from an efficiency point of view.

Knowledge acquisition support Given a structure-preserving design, the knowledge-level description can fulfill the role of semantic information about pieces of code of the artefact. This additional information can be used to support knowledge acquisition in various ways. Some examples:

- One can construct editors for entering domain knowledge directly into the system which interact with the user in the vocabulary of the model, similar to systems like MOLE [Eshelman, 1988].
- One can build debugging and refinement tools which spot errors and/or gaps in particular parts of a domain knowledge base by examining its intended usage during problem solving.

- It is possible to focus the use of machine learning techniques to generate a *particular type* of knowledge, e.g. abstraction and specification knowledge (cf. the Acknowledge project [Shadbolt & Wielinga, 1990; van Heijst *et al.*, 1992]).

An example might help to illustrate the advantages of a structure-preserving design. Suppose we have in a medical application some pieces of domain knowledge of the following form (here phrased as logical implications):

temperature $\geq$ 38.0 $\rightarrow$ fever = present
diastolic-pressure $\geq$ 95 $\rightarrow$ blood-pressure = elevated
blood-pressure = elevated $\rightarrow$ hypertension = present

Suppose also that we have the following two inferences in our model of expertise applying such pieces of domain knowledge in the reasoning process:

(i) An *abstraction* inference in which a finding, e.g a quantitative property like the temperature of a patient, is abstracted into a more abstract finding such as fever.
(ii) A *specify* step which defines an inference that can be used to find-out which observable should be asked to the user once it becomes known that the patient has, for example, a fever.
Although the validity of the model is not the issue here, it might be useful to add that physicians usually want to get an answer to the question "What is the temperature" when they learn (e.g during physical examination) that a patient has a fever, as the precise value can be important in other parts of the reasoning process.

Fig. 6.4 depicts a simple inference structure containing these two inference steps.[3]. The italic annotations of the meta-classes denote the domain-knowledge elements that could fulfill the role of finding or observable. The domain implications listed above are used by both inferences in their "domain view" (see Sec. 3.4.1). The inference structure also contains an additional step (the oval with the dashed border), which denotes a transfer task (see Sec. 3.4.3) for obtaining the value of an observable (e.g. "What is the value of the temperature").[4]

A structure-preserving design dictates that this two-fold use of essentially the same pieces of domain knowledge is made explicit in the final implementation. For example, implementing these domain expressions in duplo as abstraction and as specification rules would violate the structure-preserving property. It would give rise to knowledge redundancy (the same knowledge is present twice in the system) and can lead to serious maintenance problems (when specification knowledge is changed, the abstraction knowledge needs to be changed as well). For a knowledge acquisition tool it could mean that the user needs to enter the same piece of knowledge twice (or it would require an undesirable *ad hoc* adjustment of the tool). For explanation purposes it is important to be able to explain the different usage of these implications. At different points in the reasoning process the explanations about the role of these implications can vary.

[3]This inference structure is in fact a small fragment of the KADS inference structure for heuristic classification (see Fig. 5.15).

[4]As pointed out in Ch. 3 such an inference structure only describes the data dependencies between inferences.

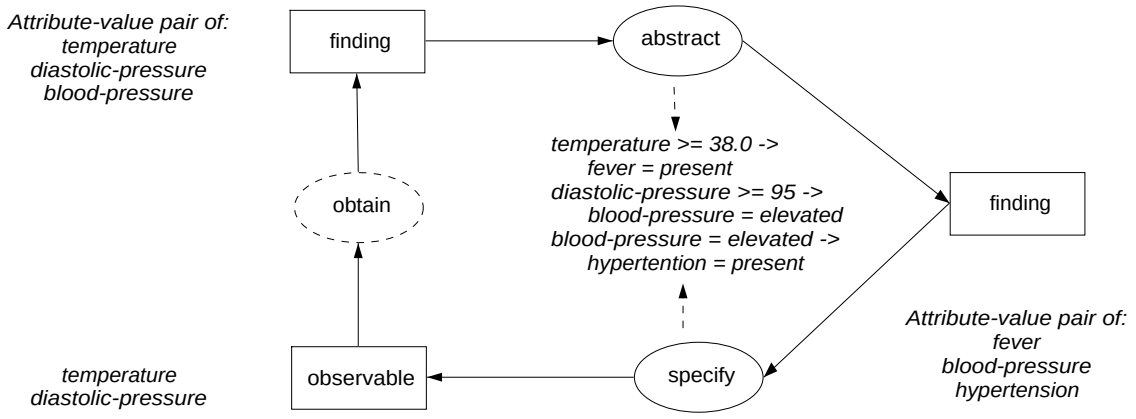

FIGURE 6.4: Two inferences in a medical application with some examples of related domain knowledge. Both *abstract* and *specify* use the same domain implications, but for different purposes. The dashed oval denotes a transfer task.

Structure-preserving design is currently also being advocated in conventional software engineering, especially in the area of object-oriented modelling and design [Rumbaugh *et al.*, 1991; Coad & Yourdon, 1991]. The motivation there mainly concerns reusability and maintenance. This point is discussed in more detail in Ch. 8.

6.4 Structure-Preserving Design: A Skeletal Architecture

In structure-preserving design, the idea is to use the model of expertise as a skeletal architecture of the artefact. In fact, most knowledge engineers using KADS think about the model of expertise as some (semi-)executable specification. One could say that they have internalised a particular architectural interpretation of the model and use this to explain how the model they have built can solve a particular problem. However, the information in the model of expertise is incomplete from the executable-specification point of view. This is not surprising, as this model is primarily meant for analysis purposes and not for design. The main ingredients that are missing and that need to be considered in architectural design are *inference methods*, *domain indexing and access functions*, *working memory*, and input/output functions:

Inference method Knowledge sources specify the nature of the input and output (the meta-classes) and the domain knowledge used in deriving the output from the input (the domain view). Knowledge sources do *not* specify *how* the inference will be achieved. This *how* description is typically something that has to be added during design. During analysis, the knowledge engineer often takes, what one could call, an *automated-deduction view* on a particular inference: the knowledge engineer specifies a knowledge source in such a way that she knows that it is possible to derive a conclusion, given the available knowledge, no matter how complex such a derivation in practice might be. In analysis, the emphasis lies on a competence-oriented description: can I make this inference in principle, and what is the information I need for making it happen. An inference method specifies a computational technique,

that actually does the job.[5] Some example inference methods mentioned in [Breuker *et al.*, 1987; p. 41] are inheritance, empirical association, matching algorithms, and generalisation.

One can take the view that inference methods are part of knowledge sources and thus should not have the status of separate architectural components. However, the relation between knowledge sources and inference methods is not one-to-one. Several knowledge sources may apply the same inference method, but for different purposes. For example, in the StatCons system for statistical consultancy two inference methods realised in total eight inferences [de Greef *et al.*, 1987; pp. 73-99]. Also, in the Sisyphus application described in Ch. 7 one method is used to realise three different inferences.

The reverse can also be true, namely that one inference function is realised through multiple methods. Thus, incorporating inference methods into operational knowledge sources prevents making full use of the reusability concept.

Domain access and indexing One of the subtle points of the model of expertise framework is the use of functional names (i.e. meta classes, domain views) to describe the inference process. In the knowledge-source specifications (see e.g. Sec. 3.4.2) the mapping between functional names and domain-specific terms is indicated. During design, one has to construct for these inference/domain mappings *indexing* mechanisms for the (operational) domain knowledge-base. Domain-access functions use this indexing information to retrieve the required domain knowledge for carrying out a particular inference. These access functions ensure that system elements that realise inferences (the "inference functions", see further) can be specified fully in a manner that is independent of the application-domain. This is a key point with respect to reusability (see also the discussion on data-function interactions in Ch. 8).

Working memory During analysis one is (and rightly so) sloppy in defining storage of the run-time data. This is mainly because a full specification of this type of information requires a detailed description of all kinds of data manipulations. The working memory (with which we mean the database of the run-time results of inferences and tasks) is only specified implicitly, in particular through meta-classes and control terms (see Sec. 3.4.3 for a definition of the notion of control terms). During design, one has to define however explicitly the nature of working memory, possibly adding other types of run-time information as well: e.g. which tasks have been executed etc.

HCI functions Although the emphasis lies in this chapter on the impact of the model of expertise on the design, a few remarks are in place here about the components that are concerned with the interaction of the system with external agents. The model

[5] If the model of expertise is fully specified in some strictly formal language such as ML^2 [van Harmelen & Balder, 1992; Akkermans *et al.*, 1992] it is in principle possible to use a dedicated theorem-prover for executing the model. In practice however, some inferences involve computations that are, given the state of the art, computationally intractable within such an approach . In the KADS-II project, we are developing a theorem-prover for simulating the problem-solving behaviour of a particular model, providing pragmatic short-cuts for intractable parts. This theorem-prover is meant to be used for model validation purposes and is not expected to fulfill the role of final system.

of cooperation contains the detailed specification of this part of the system. In the skeletal architecture (cf. Fig. 6.5 such components appear as *human-computer interface (*HCI*) functions*.[6] These functions implement transfer tasks such as obtaining a value and presenting a solution.

It should be noted that the model of cooperation can interact with the specification of control in the model of expertise. For example, it can be the case that the model of cooperation specifies a desired interaction strategy in which obtaining the value of certain observables should involve asking also for values of related observables (e.g. because this increases the plausibility of the line of questioning of the system). In fact, the model of cooperation often gives rise to an adaptation of the control knowledge as specified in the model. Typically, it influences the design of task knowledge and strategic knowledge specified. This point is discussed in more detail in [de Greef, 1989].

Fig. 6.5 depicts a typical skeletal architecture for structure-preserving design. We briefly discuss its various components below, except for the HCI functions which fall outside the scope of this chapter.

Domain knowledge-base and access functions The domain knowledge-base contains a declarative symbolic representation of the domain-specific knowledge. It contains both the actual domain expressions (e.g. relation tuples, concept instances) as well a description of the structure of the domain expressions (the symbolic representation of the domain schema, cf. Sec. 3.4.1).

This schema is used by the knowledge-base access functions to retrieve certain types of domain knowledge, using the mapping specifications provided by meta-classes and domain views. The access functions should be able to handle requests such as 'retrieve all abstraction knowledge" or "retrieve a domain entity that can play the role of observable".

The operationalised domain schema can also be used for explanation purposes (e.g. providing an answer to the question: "what domain knowledge was used in making this inference") and for the development of domain-knowledge editors that interact with the user in a domain-specific vocabulary, similar to OPAL [Musen *et al.*, 1987].

Inference functions and inference methods We use the term *inference function* to denote the design counterpart of knowledge sources. Inference functions should contain the same information as described for knowledge sources; types of input, output, and nature of the domain knowledge used (all in domain-independent terminology). In addition, every inference function defines how a particular *inference method* can be activated to realise the inference and what type of information needs to be retrieved (using the domain access functions) for successful execution of the inference method. In Sec. 6.5, we provide an example operationalisation of the inference functions for the abstract/specify example. More elaborate examples can be found in the description of the Sisyphus application (Ch. 7 and Appendix B).

[6] The skeletal architecture makes the simplifying assumption that the external agents the system will deal with are all humans.

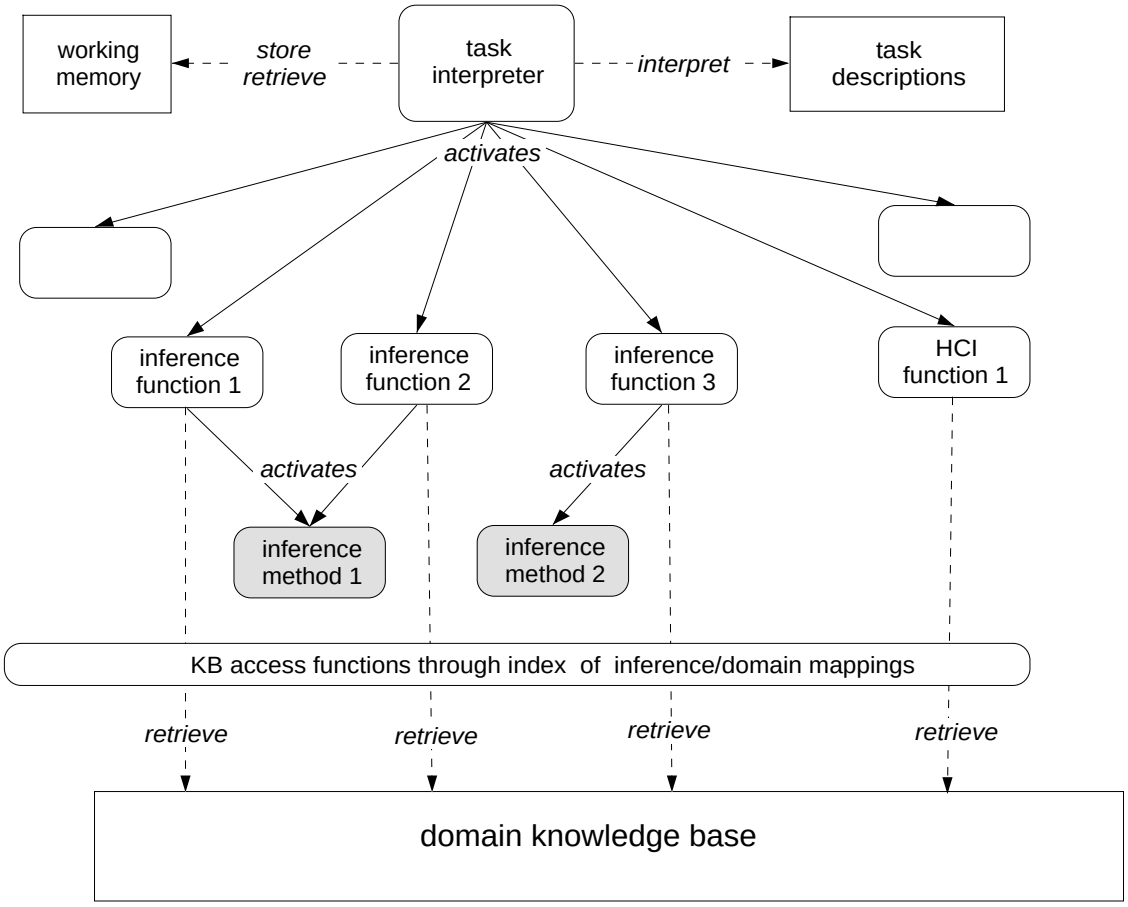

FIGURE 6.5: Skeletal architecture supporting structure-preserving design. The boxes with rounded corners denote system components that are active during reasoning; the sharp-edged boxes denote data stores. Non-dashed lines describe a control operation (*activates*) between components; dashed lines a data operation (*store*, *retrieve*, *interpret*).

Task interpreter, task descriptions, and working memory The task interpreter is the central control unit of the system in this architecture. It uses a declarative description of task structures and stores/retrieves run-time results of the reasoning process in/from working memory. The complexity of the task interpreter can vary, depending on the technique being chosen for implementing this component. In the simplest case, it is a straight-forward interpreter for KADS-type task structures. A more complex technique would be a blackboard-type of control technique. Choosing a particular control technique is discussed in more detail in Sec. 6.5.

6.4.1 Meta-level vs. object-level inferencing

The skeletal architecture shown in Fig. 6.5 has a meta-level flavour, in which the domain knowledge base represents the object-level and the inference- and task-elements represent the meta-level. Van Harmelen distinguishes three types of meta-level architectures based on the *locus of action* [van Harmelen, 1989]:

- Pure meta-level inferencing: all computation is carried out at the meta-level. The object-level has no interpreter and is only accessed to retrieve information.
- Pure object-level inferencing: the meta-level has no interpreter of its own, but is active at fixed points during the computation through the execution of predefined meta-predicates.
- Mixed inferencing: both object-level and meta-level have an interpreter. The object-level interpretation is a black box for the meta-level.

Within this classification, the skeletal architecture of Fig. 6.5 can be characterised as either pure meta-level or mixed, depending on whether the retrieval of domain knowledge involves computations in the domain knowledge-base. In the Sisyphus application the architecture is a mixed one: the domain knowledge-base contains additional axioms which are interpreted by the domain access functions: e.g. knowledge about properties of relation types such as transitivity is used to infer tuples of a relation.

6.5 Structure-Preserving Design: Computational Decisions

In this subsection we discuss some more detailed design decisions in a structure-preserving approach. The scope of the section is limited to decisions with respect to elements of the model of expertise. A more detailed description of these decisions can be found in [Schreiber *et al.*, 1987; Schreiber *et al.*, 1989a].

6.5.1 Inference and domain knowledge For each knowledge source a corresponding computational technique needs to be selected that can realise this inference. A technique consists of three types of elements: (i) an algorithm, (ii) input-output data structures and possibly additional temporary data structures, and (iii) a representation of domain knowledge. The algorithm embodies the method for realising the inference and specifies the local, symbol-level control (cf. Sec. 2.3).

As remarked before, a number of groups of computational techniques have been developed in AI research, such as production systems, state-space search, parsing, classification and matching. These groups can be viewed as computational paradigms. Detailed studies have been performed to unravel the criteria for choosing a technique within one group such as hierarchical classification [Goel *et al.*, 1987] or automated deduction [Reichgelt & van Harmelen, 1986]. In [Schreiber *et al.*, 1989a] criteria for chosing a particular type of technique are discussed. Often, knowledge engineers use within one system only one or two types of techniques. For example, in NEOMYCIN [Clancey, 1985a] four production system techniques are provided. Each inference[7] applies one of these production system techniques. Applying only one type of technique, such as production systems, in one particular KBS minimises the interaction problems with the design of other parts of the system (in particular the domain knowledge-base, as it requires just a single representational formalism), but apart from that there is no compelling reason to adhere strictly to this approach.

[7] NEOMYCIN distinguishes only one "inference" type: *apply-rule*. The lowest level of meta rules fulfills however a similar role as knowledge sources in KADS, namely performing the computations using domain knowledge.

A crucial design decision concerns the choice of the representation technique(s) for the domain knowledge. Often, the nature of the knowledge described in the domain schema indicates suitable symbolic representations. If the knowledge engineer works within a particular paradigm, such as production systems, the choice of the representational technique is usually obvious: the domain representation technique is the same as the representation used by the chosen type of computational technique.

With respect to the structure-preserving principle, the most difficult part is to preserve the information about the functional names (meta classes, domain views, the domain specific names and their mappings. As the analysis in the previous section has shown, these inference-to-domain connections specify in fact the relation between the functional perspective and the data perspective on the system and constitute crucial areas for reusability etc.

To illustrate how structure-preserving design can be achieved in this respect, we present an example of a simple Prolog implementation of the inferences in Fig. 6.4. A full listing of the code, together with some sample traces, can be found in Appendix C.

Example fragment of a model of expertise We assume that the inferences in Fig. 6.4 are specified in the following way in the conceptual model:

```
knowledge-source abstract
  input-meta-class:
    finding → some expression about patient-data
  output-meta-class:
    finding → some expression about qualitative-data
  domain-view:
    abstraction-knowledge →
      < relation(qualitative-abstraction), relation(definition) >

knowledge-source specify
  input-meta-class:
    finding → some expression about qualitative-data
  output-meta-class:
    observable → some property of qualitative-data
  domain-view:
    specification-knowledge →
      < relation(qualitative-abstraction), relation(definition) >
```

These inferences use the same domain knowledge (see the domain view), but for different purposes.

The domain knowledge is specified in the model of expertise using the domain description language (DDL) defined in Ch. 4:

```
concept patient-data ;

concept quantitative-data ;
  sub-type-of: patient-data ;
  properties:
    temperature: number-range(35 - 42) ;
    diastolic-pressure: number-range(0 - 300) ;

concept qualitative-data ;
```

sub-type-of: patient-data ;
properties:
fever: {absent, present} ;
blood-pressure: {normal, elevated} ;
hypertension: {absent, present} ;

relation *qualitative-abstraction*
argument-1: expression(quantitative-data) ;
argument-2: expression(qualitative-data) ;
tuples:
< temperature $\geq$ 38.0, fever = present >
< diastolic-pressure $\geq$ 95, blood-pressure = elevated > ;

relation *definition*
argument-1: expression(qualitative-data) ;
argument-2: expression(qualitative-data) ;
tuples:
< blood-pressure = elevated, hypertension = present > ;

The DDL description specifies two types of attributes of a patient and two relations that express relations between expressions about such attributes.

Representation of domain knowledge For this example we use a set of Prolog predicates that allow an almost direct translation from DDL statements onto the chosen knowledge representation. This representation was also used for the implementation of the Sisyphus application (cf. Ch. 7 and Appendix B)). The main idea is to keep the domain knowledge as much as possible in the form of a declarative theory of the domain, without any particular commitment towards specific use during reasoning.

```
% concept(Concept name, Supertypes)

concept(patient_data,       []).
concept(quantitative_data,  [patient_data]).
concept(qualitative_data,   [patient_data]).

% property(Concept,         Property name,        Valueset)

property(quantitative_data, temperature,          numberrange(35.0, 42.0)).
property(quantitative_data, diastolic_pressure,   numberrange(0, 300)).
property(qualitative_data,  fever,                [present, absent]).
property(qualitative_data,  blood-pressure,       [normal, elevated]).
property(qualitative_data,  hypertension,         [present, absent]).

% relation(Relation name,   Type first argument,     Type second argument)

relation(qual_abstraction,  expr(quantitative_data), expr(qualitative_data)).
relation(definition,        expr(qualitative_data),  expr(qualitative_data)).

% tuple(Relation name,   [First argument,            Second argument])

tuple(qual_abstraction,  [temperature >= 38.0,       fever = present]).
tuple(qual_abstraction,  [diastolic_pressure >= 95,  blood-pressure = elevated]).
tuple(definition,        [blood-pressure = elevated, hypertension = present]).
```

Automatic generation of this representation from the DDL would require little effort.

Domain index The domain index specifies the mappings from inference-level names (see next paragraph) onto domain-specific data types in the knowledge-base.

```
domain_index(expression, finding,       [ expr(patient_data) ]).
domain_index(entity,     observable,    [ property(patient_data) ]).
domain_index(relation,   abstraction,   [ relation(qual_abstraction)
                                        , relation(definition) ]).
domain_index(relation,   specification, [ relation(qual_abstraction)
                                        , relation(definition) ]).
```

The predicate *domain-index* has three arguments, namely:.

1. The inference-level data type (one of *entity*, *relation*, and *expression*).
2. The inference-level name (e.g. finding, observable).
3. A list of domain types that can play the role of this inference-level object.

Thus, the inference-level relations "abstraction" and ' 'specification" both map onto all tuples of two domain relations.

Inference knowledge The inference-knowledge representation consists of three elements. The first element is the declaration of inferences as defined in the model of expertise.

```
% inference(Internal name, External name)
% metaclass(Inference,     Input/Output,        General name, Specialised name).
% domain_view(Inference, , Inference knowledge).

inference(  abstract,  'Abstract').
metaclass(  abstract,  input(1),       finding,     'Specific finding').
metaclass(  abstract,  output,         finding,     'General finding').
domain_view(abstract,  relation(abstraction, finding, finding)).

inference(  specify,   'Specify').
metaclass(  specify,   input(1),  finding,     'Finding to be clarified').
metaclass(  specify,   output,    observable,  'Dependent observable').
domain_view(specify,   relation(specification, finding, finding)).
```

The domain view describes the static domain-knowledge used by the inference (cf. Sec. 3.4.2).

The second element of the inference-knowledge representation is the definition of an inference function that realises this inference by retrieving the necessary domain knowledge and activating an appropriate inference method:

```
inference_function(abstract, [In], Out) :-
    domain_retrieval(find_all, abstraction, Rules),
    rule_interpreter(Rules, In, Out, forward, single_pass, find_one).

inference_function(specify, [In], Out) :-
    domain_retrieval(find_all, specification, Rules),
```

```
rule_interpreter(Rules, In, Out, backward, multi_pass, find_one).
```

The predicate *domain_retrieval* represents a knowledge-base access function that retrieves the required domain knowledge by interpreting the domain-view description of the inference. The predicate *rule_interpreter* represents a call to a production-system technique, where the last three arguments indicate the required control regime in a similar way as is NEOMYCIN:

- *forward/backward* Derive conclusion from the premise of a rule (forward) or set up premise as a goal to achieve the conclusion of the rule (backward).
- *single/multi pass* Stop once a conclusion or a goal has been inferred (single pass) or invoke the rule interpreter recursively (multi pass).
- *find one/all* Stop evaluation (find one) or continue evaluation (find all) of the rule set when a conclusion or a goal has been found.

This production-system technique interprets a domain-relation tuple as $< premise, conclusion >$.

The inference *abstract* is thus realised by a single, data-driven, evaluation of the rule interpreter using the tuples of the two domain relations as rule set. The inference *specify* is realised by a recursive, goal-directed, execution of the same rule set.

Fig. 6.6 shows a a schematic view of a sample execution of the *specify* inference. The input is a finding. The corresponding inference function calls a domain-retrieval function to retrieve all specification rules. This retrieval function uses the domain index to collect the appropriate domain knowledge. Finally, the inference method (the rule interpreter) is activated which produces the output (in this case an observable).

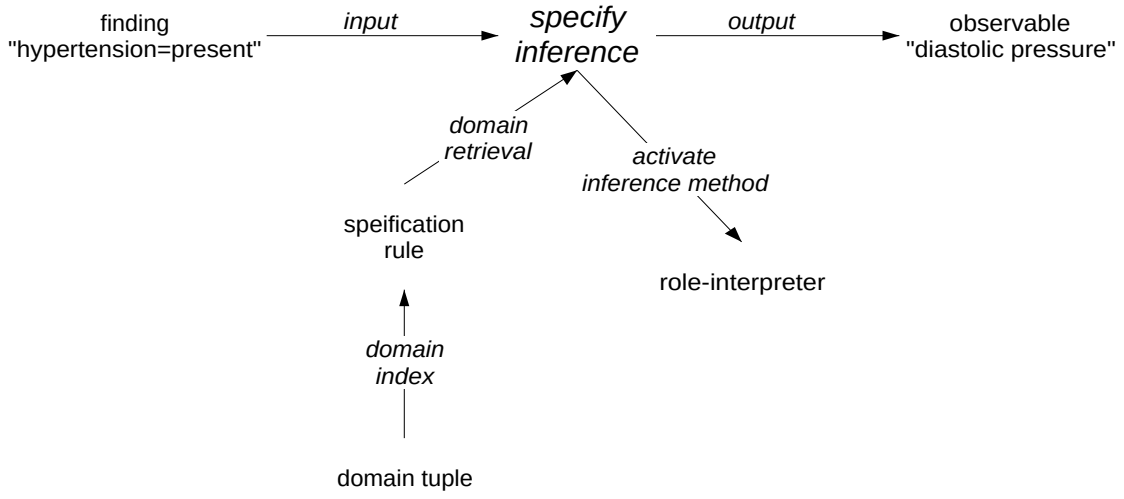

FIGURE 6.6: Schematic overview of a sample execution of the inference *specify*.

The third element of the inference-knowledge representation is the actual definition of the inference method. The corresponding Prolog predicates implementing the rule interpreter can be found in Appendix C.

6.5.2 Task knowledge Given the set of tasks specified in the conceptual model (consisting of both problem solving tasks and of transfer tasks) the designer has to make two – interrelated – decisions, namely:

1. The choice of a control technique for executing tasks. The simplest solution would be to define an interpreter for a representation of the task structures in the model of expertise. This solution is sufficient in the case where the model of cooperation does not impose additional requirements on the control regime (see the discussion earlier on HCI functions). This is the solution taken in the diagnostic system described in Sec. 6.7.
   If the model of cooperation imposes additional control requirements, it is usually appropriate to employ a more complex control technique, that integrates the execution of task structures and the control of HCI functions as defined in the model of cooperation. Control techniques that could be used for achieving this are an agenda mechanism, a blackboard or a skeletal planning technique. The StatCons system is an example KBS where the user interface requires complex control [de Greef *et al.*, 1988b].
2. The choice of how to represent and update the run-time data. These data can be viewed as the "working memory" of the KBS. This working memory contains the data that are manipulated by the tasks and the inferences: e.g. the current state of the differential, the findings, etcetera. The control terms and the meta-classes specified in the conceptual model form the basis for the representation of working memory: they often reappear in the final system as labels for (sets of) working memory elements.
   Most existing KBS's use a simple monotonic technique for updating working memory. We expect that in the next generation KBS's more complex techniques such as truth-maintenance techniques will be used more often. Note that the use of such a technique can pose additional requirements on the output produced by computational techniques realising primitive inferences. An example of such an additionally required output is the "justification" used in an ATMS [de Kleer, 1986].

In Appendix C an example task structure representation can be found for two tasks that apply respectively the *abstract* and the *specify* inference.

6.5.3 Strategic knowledge Most conceptual models that have been constructed do not contain much strategic knowledge, if any at all. In most systems the strategic part has been "compiled out" into fixed task decompositions with possibly a few strategic decision points that can be influenced by the user (cf. [de Greef *et al.*, 1987]).

If more elaborate strategic knowledge is present, the following techniques could be applicable:

- A production system containing a set of meta-rules with states of working memory as conditions and task activations and/or changes to working memory (e.g. assumptions) as actions.
- An extended blackboard technique such as the "Blackboard Control Architecture" [Hayes-Roth, 1985], where the scheduling part represents the strategic knowledge.

From the viewpoint of the skeletal architecture in Fig. 6.5, these techniques involve the development of a complex central control unit (the term "task interpreter" is probably not appropriate anymore within this context).

Strategic knowledge as meta-knowledge An alternative route is to view the strategic knowledge as a separate meta-system, The PDP system is an example of this approach [Jansweijer *et al.*, 1986]. This approach has been the focus of the REFLECT project [van Harmelen *et al.*, 1992]. In the REFLECT approach the strategic knowledge is viewed as a *meta-theory* about the three other knowledge categories in the model of expertise (the object-theory). This meta theory can be described with the same modelling framework as the object-theory. In architectural terms, this means that the skeletal architecture sketched in Fig. 6.5 is extended to a meta-level architecture in which a strategic meta-system reasons about and acts upon a model of the object system. This object-model is causally connected [Maes, 1987] to the corresponding constructs in the actual object system. In practice, this causal connection can only be achieved if the object-system has been can be operationalised in a structure-preserving way.

Note that the architecture of the object-system itself is also of a meta-level nature: the inference-level components reason about an d act upon the (object-level) domain knowledge (see the discussion in Sec. 6.4.1).

The term "knowledge-level reflection" has been coined for the REFLECT approach to distinguish it from most other meta-level approaches that reason directly about the actual code fragments of a system. The REFLECT architecture allows to build systems that carry out reflective tasks like competence assessment and competence improvement in a flexible way. For more details about this architecture, the reader is referred to [Reinders *et al.*, 1991; Schreiber *et al.*, 1991b; van Harmelen *et al.*, 1992].

6.6 Existing Approaches to Computerised Support

Although in principle a knowledge-level model can be viewed as a specification that can be implemented in a conventional manner, for most modelling approaches dedicated environments exist, that support structure-preserving operationalisation. In this section, we discuss the nature as well as the merits and limitations of each of approaches.

6.6.1 Types of support environments Existing support environments can roughly be be divided into three categories:

1. Task-specific shells
2. Task-specific programming languages
3. Task-independent programming languages

Task-specific shells Task-specific shells support the operationalisation of a range of application domains. In KADS terms, a task-specific shell can be seen as an operationalisation of an interpretation model. The model incorporated in a task-specific shell represents a problem solving method for solving a certain type of problems. Example task-specific shells are MOLE (method: cover & differentiate; [Eshelman, 1988]), SALT (method: propose & revise; [Marcus & McDermott, 1989]) and OPAL (method: skeletal

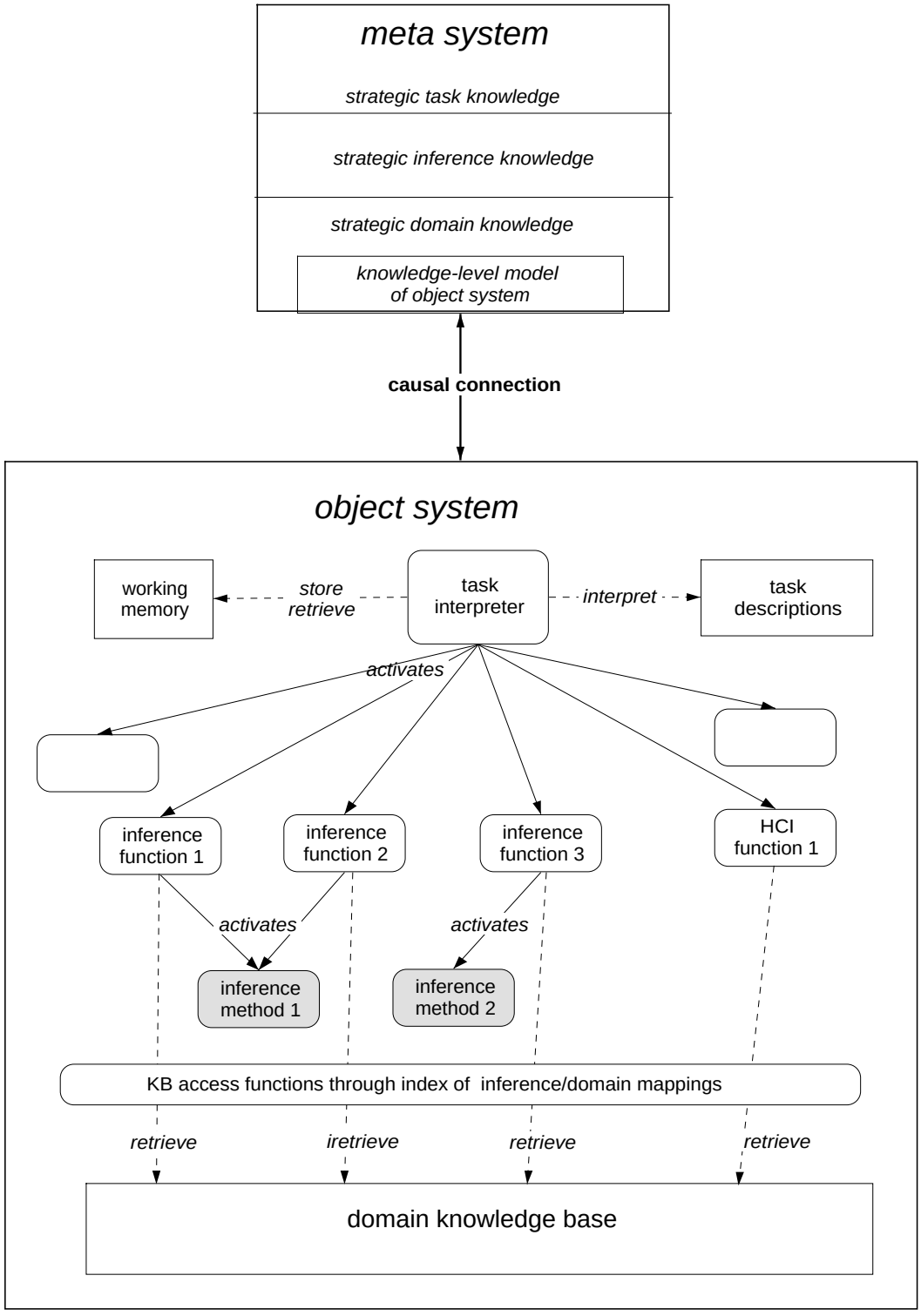

FIGURE 6.7: Skeletal meta-level architecture in which the strategic knowledge is operationalised through an separate meta system. This meta system uses a a causally connected knowledge-level description of the object system to reason about and act upon this system.

plan refinement; [Musen *et al.*, 1987]). In task-specific shells the structure of the task is fixed: the knowledge engineer cannot change the structure of the domain knowledge (i.e. the domain schema), the set of inferences or the control imposed on these inferences. Only the domain-specific knowledge needs to be entered in the predefined format.

It is assumed that the expert is capable of entering this domain knowledge directly into the system with the help of a dedicated domain-knowledge editor. [Musen *et al.*, 1987] makes the point that this can best be achieved by providing the expert with a support tool that communicates with the user in a domain-specific vocabulary. OPAL uses this approach. This is not the case in MOLE and SALT, where the interaction is (at least partially) in terms of functional objects.

The PROTEGE system [Musen, 1989] overcomes some of the problems associated with task-specific shells. This system allows the knowledge engineer to tailor a skeletal planning method to meet the demands of a particular application domain (within the area of managing protocols for medical treatment) and then generates an OPAL-like shell that can interact with the expert to acquire the domain-specific knowledge.

Task-specific programming languages A second type of support environments is represented by the task-specific programming languages such as developed within the Generic Task (GT) approach [Chandrasekaran, 1988]. These programming languages contain constructs specific for operationalising a particular (generic) task (see Sec. 3.7 for a discussion on the relation between GT and KADS). An example of such a language is CSRL [Bylander & Mittal, 1986], which supports the operationalisation of hierarchical classification problems.

The methodological viewpoint behind the GT approach is that problem solving consists of a relatively small set of information processing tasks (the generic tasks) and that a particular instantiation and configuration of these tasks can be used for realising a particular application. Each programming language supports such a generic task. The resulting generic task programs are integrated into one system in an object-oriented fashion.

Task-independent programming languages Task-independent programming languages are high–level programming languages that allow a (relatively) simple mapping from the knowledge-level model onto computational constructs in the language, For the KADS approach a number of such languages have been developed: e.g. Model-K [Karbach *et al.*, 1991] OMOS [Linster & Musen, 1992] and KARL [Angele *et al.*, 1991]. For the Components of Expertise approach the language described in [Vanwelkenhuysen & Rademakers, 1990] has been developed. ZDEST-2 [Tong *et al.*, 1988; Karbach *et al.*, 1988] can also be viewed as such a language, although it is not tied to a particular knowledge-level modelling approach.

The mapping from model to language elements is in these languages usually supported by giving the computational constructs names similar to the model elements. For example, Model-K offers computational constructs with KADS-specific names such as *knowledge source* and *metaclass*.

6.6.2 Merits and limitations of the approaches

Task-specific shells provide a high level of support to the knowledge engineer. If the application domain fits well with the

environment, then only relatively little effort is necessary for building a system. The price paid for the high level of support is a low level of flexibility: small mismatches between domain and tool can already render the tool unsuitable for the target application.

The task-independent programming languages leave the knowledge engineer with considerable freedom during operationalisation. Within the limitations of the computational techniques supported by the language, it is possible to operationalise a variety of applications. On the other hand, their level of support for the knowledge engineer is limited. Unlike the task-specific shells, which contain reusable computational constructs for one particular knowledge-level model, the current programming languages do not contain any reusable pieces of code, e.g. predefined domain structures or inferences. So, the higher level of flexibility is paired with a lower level of support.

The task-specific programming languages take more or less an intermediary position. This approach is flexible in the sense that the control knowledge is not fixed: the knowledge engineer has to 'program" each generic task as well as specify how these should be integrated to solve the overall problem. In terms of support, in particular the reusability of code, this approach is limited: the languages themselves are of course reusable, but there are no predefined pieces of code, such as provided by the task-specific shells.

The three approaches can be seen as as points on a spectrum determined by the level of support vs. the level of flexibility (Fig. 6.8).

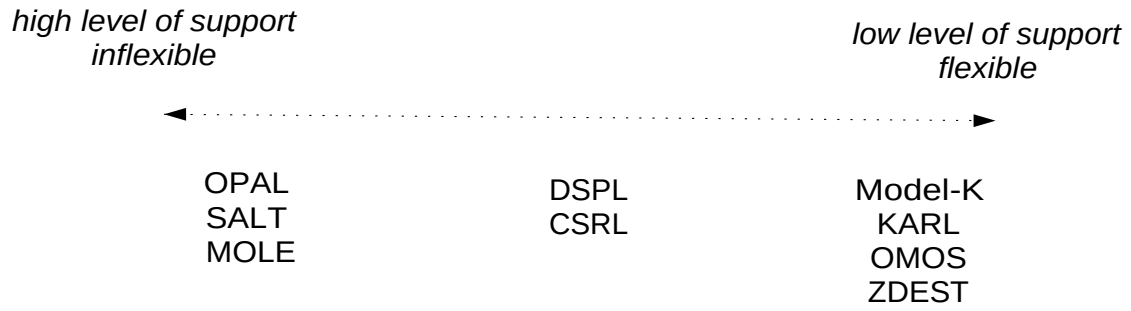

FIGURE 6.8: Characterisation of some support environments with respect to level of support and flexibility.

6.6.3 Some remarks about operational "knowledge-level" languages In the last few years, several languages have been proposed for operational and/or formal representation of KADS models of expertise. Some examples were given earlier in this section. Some additional remarks about this line of research are in place.

It is clear that it is desirable to have, as early as possible in the development process, some machine-executable version of the model of expertise. Such prototypes can be used for simulation of the problem-solving behaviour that is specified in the model and can thus play a role in validation of the model.

One should however separate two fundamentally different aspects of such languages:

1. The *modelling* aspects of the language: what constructs does the language offer for modelling the reasoning process in the application domain.
2. The *operational* aspects of the language: what is the operational interpretation of the language constructs.

There is a kind of trade-off between these two aspects. During modelling, one wants to be as flexible as possible in specifying the required problem-solving behaviour. One is primarily interested in the declarative semantics of the language. During operationalisation, one needs to restrict the language to expressions that have operational semantics.

The formal specification languages for KADS models of expertise such as ML^2 [van Harmelen & Balder, 1992; Akkermans *et al.*, 1992] emphasise the modelling aspects, although they have a (partial) operational interpretation. The task-independent programming languages are much more directed towards operational aspects. The languages KARL [Angele *et al.*, 1991] and FORKADS [Wetter, 1990] take an intermediate position, emphasising both modelling and operational aspects.

Fig. 6.9 characterises the different languages on a spectrum from formal specification (emphasising modelling) to executability (emphasising operationality).

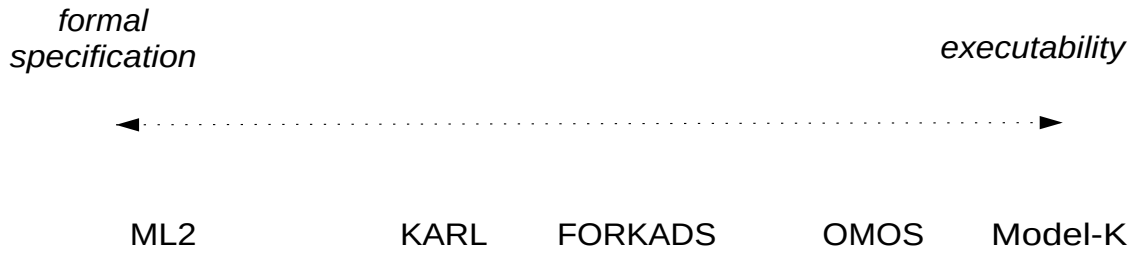

FIGURE 6.9: Formal languages for KADS models of expertise interpreted as points on a continuum from formal specification to executability.

Also, it should be noted that in most real-life applications external requirements dictate constraints on the software environment in such a way that the operational languages cannot be used for the final implementation. For example, the successful Fraudwatch system [Killin, 1992; Porter, 1992] was implemented partly in Cobol (while following the structure-preserving principle which indeed greatly facilitated the maintenance of the system [Killin, 1992]).

6.6.4 Design languages Another type of support for design can be given by a *design language* such as DESIRE [van Langevelde *et al.*, 1992]. In this approach, it is assumed that there exists some (informal) conceptual model of what the system should do. Given this input, the design language supports the formal specification of appropriate system components and their configuration into a system architecture. From this design specification the system code can then be generated automatically.

The DESIRE approach fits in fact quite well with the KADS approach. One major advantage is that there is a clear separation of roles: the model-of-expertise language used during analysis emphasises modelling aspects; the design language emphasises operational aspects. This circumvents the problems encountered with some afore-mentioned "knowledge-level programming languages" in which the distinction between these, fundamentally different, view points is not clear.

An interesting research question would be to study the possibility of predefining within a design language such as DESIRE the components of a skeletal architecture like the one in Fig. 6.5. This would simplify the mapping from the analysis input onto components in the design language.

6.7 Maximising Support and Flexibility: An Example

Ideally, one would want to combine the support provided by reusable pieces of code and the flexibility offered by the "knowledge-level" programming languages. The grain size of task-specific shells is too coarse: these constitute an implementation of a complete interpretation model. On the other hand, the programming languages do not provide any reusable code fragments such as ready-to-use implementations of inferences. What appears to be needed is an environment with a library of reusable modules that can be used to operationalise elements of a knowledge-level model and allow the knowledge engineer to configure these into a system that meets the demands of the application at hand.

Within the framework of KADS we have developed a prototype environment that can be considered as a first attempt in this direction. It contains a library of modules that can be used to operationalise inferences for a class of diagnostic tasks. These inferences appear in the interpretation model for systematic diagnosis. This model represents a method for diagnosis in which a device is examined in a systematic, top-down, manner to find a component that behaves abnormally. Systematic diagnosis can be seen as a form of generate and test. Hypotheses are generated through decomposition of a hierarchical device model into sub-components. A component is tested by predicting the value of an observable using a model of the normal behaviour of the device and comparing this norm value with the observed value. The control knowledge typically has a recursive structure: decomposition is carried out until a non-aggregate abnormal component is found.

This basic version of the model of systematic diagnosis can be extended in a number of ways. These extensions are adaptations of the model that could be necessary in a particular application. They modify the basic model of systematic diagnosis and define in fact a space of potential systematic diagnosis models. Two example extensions are (i) the introduction of complex tests that require system reconfiguration (e.g. reconnecting cables), and (ii) the use of multiple device models each representing a different view on the device (e.g. functional, physical). An overview of the complete set of inferences is given in Table 6.2. For a more detailed description, see Sec. 3.4.2 (basic version) and Sec. 5.4 (extensions).

The environment we developed contains modules for all potential inferences, both for the basic version and for those required by the extensions. Each module represents a computational technique for realising the inference. Each computational technique has particular domain knowledge requirements, e.g. the technique for realising the inference in which a norm value is predicted requires a causal model of the normal behaviour of the device.

The environment offers the following facilities to a knowledge engineer who wants to build a system for a particular instance of systematic diagnosis:

- The instantiation of techniques for the required set of inferences.
- A language to define tasks that specify the sequencing of inferences (control knowledge). The control language is an operationalisation of the modelling language used for describing task structures. It contains additional constructs necessary for implementation, such as invocation of user interface functions and storage and inspection of of intermediate results. An example fragment of this language can be seen in the top-right window of Fig. 6.10.

Inference	input	output	description	used in
select	complaint	system model	selection of an appropriate device model	basic version
decompose	system model	differential	hypothesis generation though decomposition of the device model	basic version
select first	differential	hypothesis	select a component for testing	basic version
specify	hypothesis	test	find a test for the a component to be tested	basic version
specify	test system model	norm	predict the normal test outcome	basic version
compare	value, norm	truth value	compare the observed and the expected value	basic version
assemble	complaint initial data	system model	dynamic creation of the device model	extension
transform	system model	system model	allow tests that require a reconfiguration of the device	extension
sort	differential criterion	differential	order the hypotheses in the differential	extension
select	complaint	view	allow multiple device models each representing a view	extension

TABLE 6.2: Inferences in systematic diagnosis models

- A simple editor for entering the domain knowledge required by the selected inferences.

The environment also supports the execution of the resulting system and an interface that allows a user to trace the reasoning process in the vocabulary of the knowledge-level model. Fig. 6.10 shows part of this interface. The interface allows the user to trace the reasoning with respect to various aspects of the model, such as:

- the task that is being executed and its internal control structure;
- the inference-structure diagram in which an inference is highlighted when it is being executed;
- the bindings of meta-classes such as "system model".
- the domain knowledge that is used by inferences that are executed.

A detailed description of the environment can be found in [Lemmers, 1991]. The architecture of the environment is an instantiation of the skeletal architecture of Fig. 6.5.

Future perspectives for support The environment for systematic diagnosis models discussed in the previous section is only a first step in the direction of flexible support for operationalising knowledge-level models. It has still a number of important limitations:

- For each inference only one computational technique is provided to operationalise it. This is too restrictive. For example, the inference in which a norm value is predicted can currently only be operationalised with a technique that uses a causal model of the device. For some applications other techniques might be more attractive, for example some form of qualitative reasoning. Ideally, the environment should support a range of techniques for realising some inference.

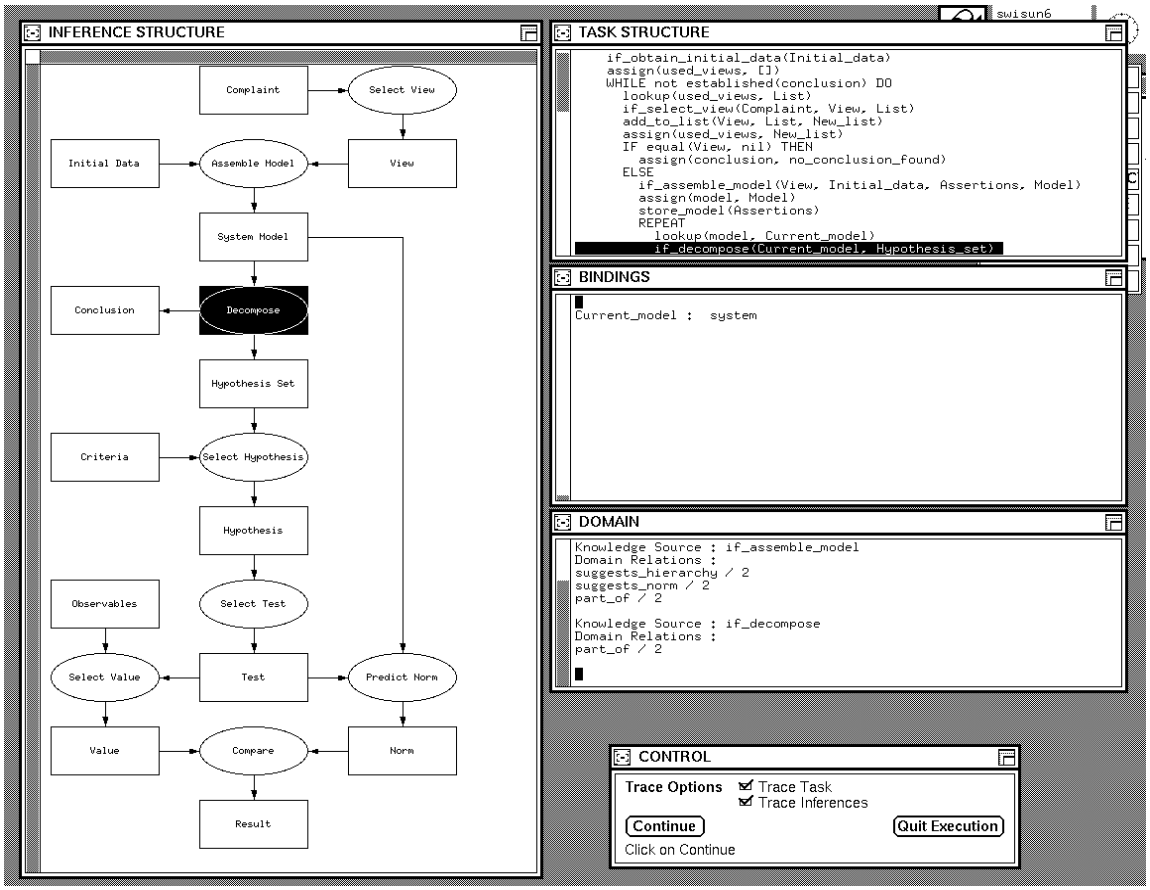

FIGURE 6.10: Prototype interface for tracing the execution of a system for systematic diagnosis in the vocabulary of the knowledge-level model. The inference structure is shown on the left. The *decompose* knowledge source is currently being executed. The task structure (control knowledge), the bindings (in this case the system model that is currently being decomposed) and the domain knowledge used by the decompose knowledge source (a part-of relation) are shown on the right. The window in the lower-right corner allows the user to trace the reasoning process at the task and/or inference level.

- The environment provides only limited support for specifying control knowledge. The notion of task decomposition methods, such as used in Components of Expertise [Steels, 1990], can be of value here. Task decomposition methods specify prototypical decompositions of a task into sub-tasks and/or inferences plus information about sequencing these. Incorporating such methods also as reusable constructs in an environment can support the knowledge engineer also in this respect.

- The environment supports the operationalisation of just a small set of models. For example, to be able to cover a large range of diagnostic applications it would be necessary to include also heuristic methods for diagnosis and combinations of systematic and heuristic methods (cf. [Benjamins *et al.*, 1992b]).

Within the context of the KADS-II project (ESPRIT project 5248) we are developing a more comprehensive environment. This environment will support the operationalisation of a large variety of models applying some form of generate and test. The environment

will contain a large set of techniques for both hypothesis generation as well as hypothesis testing. The knowledge-level models supported there are the result of a unification of the original KADS-I model of expertise (the "four-layer model") and the Components of Expertise framework [Wielinga *et al.*, 1992b].

Other researchers are working along similar lines. Both in Spark/Burn/FireFighter [Klinker *et al.*, 1991] and in PROTEGE-II [Puerta *et al.*, 1991] the aim is to overcome the limitations of the role-limiting methods described earlier by providing the knowledge engineer with a set of predefined computational "mechanisms" from which she can configure a system.

An important research question that arises is whether it it possible to come up with an appropriate typology of such mechanisms. Such a typology would pave the way for building a library of reusable operationalisations of model elements which is not ideosyncratic for one particular approach and can thus be shared by several groups.

6.8 Discussion

The structure-preserving approach as outlined in this chapter has been applied in a number of system development projects. The StatCons system [de Greef *et al.*, 1987; de Greef *et al.*, 1988b], developed in an early P1098 experiment, was developed along these lines and served as an important source of ideas for the theoretical background. The mixer-configuration system [Billault, 1989], also part of a P1098 experiment, used the structure-preserving approach and experimented with more flexible forms of control. The developers of the Fraudwatch application [Porter, 1992; Killin, 1992] which has been in commercial use for some time now remark that the KADS approach to design in fact leads to a system which is easier to maintain that a conventional system. This despite the fact that part of the system had to be be implemented in COBOL. Object-systems developed in the REFLECT project where all based on a structure-preserving design, and this was found to facilitate the construction of flexible meta-systems on top of these object-systems [van Harmelen *et al.*, 1992]. Also, the fact that in conventional software engineering researchers are advocating a design approach in which design is seen as adding implementation detail to an analysis model ([Rumbaugh *et al.*, 1991]), supports our view that structure-preserving design is a promising approach.

However, still a lot needs to be done to support designers in the actual process. The support environments discussed earlier are one way of providing support, but are currently only useful in the realm of prototyping. External requirements of the KBS development process are often in conflict with the constraints of such an environment. From our point of view, a promising line of research is to study mappings between knowledge-level models and design languages that guarantee the structure-preserving property for the resulting system.

Acknowledgements Marco Lemmers implemented the systematic-diagnosis system described in Sec. 6.7.

Chapter 7

Applying KADS to the Sisyphus Domain

In this chapter the KADS approach is used to model and implement the office assignment problem. We discuss both the final products (the model of expertise and the design) and the process that led to these products. Emphasis is put on modelling the problem in such a way that it closely corresponds to the behaviour of the expert in the sample protocol. The last section of the chapter addresses the evaluation points raised by the initiators of Sisyphus'91.

This chapter is a heavily revised version of a submission to the Sisyphus'91 project "Models of Problem Solving". Reference: Schreiber, G. (1992) Sisyphus'91: Modelling the office-assignment problem. In M. Linster, editor, *Sisyphus'91 Part II: Models of Problem Solving.*

7.1 Introduction & Approach

This chapter describes an exercise to model and implement the sample problem of the Sisyphus'91 project. The Sisyphus project was initiated at the European Knowledge Acquisition Workshop 1990 in the Netherlands. The aim of the project is to collect data for comparative studies of approaches in various fields. One of these fields is "Models of Problem Solving". Researchers were asked to model a domain of allocating rooms to employees and explain the rationale behind decisions made in this process. . A description of the Sisyphus'91 problem statement (drawn up by Marc Linster) is repeated for conveninece in Sec. 7.2.

This chapter is organised as follows. In Sec. 7.3 a brief account is given of the steps that were taken to arrive at the model for the office-assignment task-domain. Sec. 7.4 discusses some initial observations that came out of a first global analysis of the problem description. The next three sections describe the results of the process of modelling expertise: (i) description of the domain schema (Sec. 7.5), (ii) classification of the office-assignment task and model selection (Sec. 7.6), and (iii) model decomposition and resulting inferences and tasks (Sec. 7.7). Sec. 7.8 discusses the step from model of expertise to design and implementation. Finally, in Sec. 7.9 the proposed solution to the problem is evaluated with respect to the questions raised in the problem description.

7.2 Statement of the Sample Problem[1]

The members of the research group YQT of laboratory HNE are moved to a new floor of their château. Due to severe cuts in funding they only get a very limited number of offices. It will be quite a problem to cram them all in. To complicate matter even further some will have to share an office. After several vain attempts, that all ended in nightmares that would have impressed Freddy, the management of HNE is desperate. Sisyphus is their last hope. HNE implores the Sisyphus teams to provide knowledgeable systems that are up to the task. It is important that the systems' way to solve the problem follow the shining example of the wizard Siggy D., the only one ever managed to solve the problem. The system developers should be aware of the fact that YQT's members are used to be pampered. They all have their personal preferences and professional peculiarities that should better be observed, as the dungeons of the BABYLON tower are deep and lonely.

7.2.1 Data on people and offices Not all members of YQT can profit from this new office space in the château: about half of the group will stay in their old offices. Those that are concerned by the new assignment are:

```
Werner L.                        Juergen L.
Role = researcher                Role = researcher
Project = RESPECT                Project = EULISP
Smoker = no                      Smoker - no
Hacker = true                    Hacker = true
Works-with = Angi W.             Works-with = Harry C.
             Marc M.                          Thomas D.

< plus 13 other employees >
```

Within the subset of member we have the following organisational structure. Thomas D. is the head of the group YQT. Eva I. manages YQT. Monika X. and Ulrike U. are the secretaries. Werner L. and Angi W. work together on the RESPECT project. Harry C., Jürgen L. and Thomas D. work in the EULISP project. Michael M. and Hans W. work in the BABYLON Product project. Hans W. is the haed of this large project. Marc M., Uwe T. and Andy L. pursue individual projects. Katharina N. and Joachim I. are haeds of larger projects which are not considered in this problem.

The floor plan is shown in Fig. 7.1. C5-123, C5-122, C5-121, C5-120, C5-119 and C5-117 are large rooms that can host two researchers. Large rooms can be assigned to heads of groups too. C5-113, C5-114, C5-115 and C5-116 are single rooms.

7.2.2 Protocol Table 7.1 shows a sample transcript of a protocol in which the expert Siggy solves the problem.

Note 1 Our wizard Siggy D. seems to pursue a general strategy of assigning the head of group and the staff personnel first, followed by the heads of large projects, who through their seniority are eligible for single offices (some are more equal than others). The offices of the head of group and the staff should be close to each other. Heads of projects should, if possible, be allocated offices close to the head of group.

[1] Shortened version of the problem description drawn up by Marc Linster Copied with permission.

	The words of the wizard Siggy D.		**Comments, questions and annotations**
1	Put Thomas D. into office C5-117	1a	The haed of group needs a central offices that he/she is as close to all the members of the group as possible. This should be a large office.
		1b	This assignment is defined first as the location of the office of the head of group restricts the possibilities of the subsequent assignments.
2	Monika X. and Ulrike U. into office C5-119.	2a	The secretaries' office should be located close to the head of group. Both secretaries should work together in one large office. This assignment is executed as soon as possible, as its possible choices are extremely constrained.
3	Eva I. into C5-116	3a	The manager must have maximum access to the head of group and to the secretariat. At the same time he/she should have a centrally located office. A small office will do.
		3b	This is the earliest point where this decision can be taken.
4	Joachim I. into C5-115.	4a	The heads of large projects should be close to the head of and the secretariat. There really is no reason for the sequence of assignments of Joachim, Hans, and Katharina.
5	Hans W. into C5-114.	5a	The heads of large projects should be close to the head of and the secretariat.
6	Katharina N. into C5-113.	6a	The heads of large projects should be close to the head of and the secretariat.
7	Andy and Uwe T. into C5-120.	7a	Both smoke. To avoid conflicts with non-smokers they share an office. Neither of them is eligible for a single office. This is the first twin-room assignment as the smoker/non-smoker conflict is a severe one.
8	Werner L. and Jürgen L. into office C5-123.	8a	They are both implementing systems, both non-smokers. They do not work on the same project, but they work on related subjects. Members of the same projects should not share offices. Sharing with members of other projects enhances synergy effects within the research group.
		8b	There really are no criteria for the sequence of twin-room assignments.
9	Marc M. and Angi W. into office C5122.	9a	Marc is implementing systems; Angi isn't. This should not be a problem. Putting them together would ensure good cooperation between the RESPECT and the KRITON projects.
10	Harry C. and Michael T. into office C5-121.	10a	They are both implementing systems. Harry develops object systems. Michael uses them. This should create synergy.

TABLE 7.1: Transcript of protocol

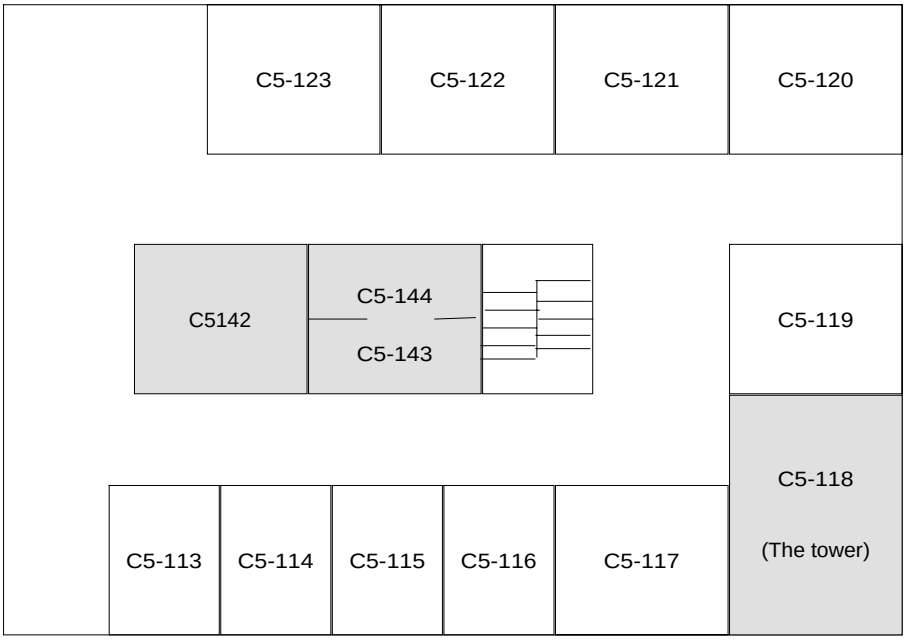

FIGURE 7.1: The part of the floor plan considered in the sample application.

Note 2 Twin offices are assigned to the members of research projects under the consideration that synergy among projects is boosted. This means that researchers that work in the same project, are if possible not sharing an office. Co-workers that work on related subjects can share an office. It is important not to put smokers and non-smokers together into twin offices.

7.3 Modelling the Office Assignment Problem

The problem description basically consists of two parts:

1. A description of the major entities (employees, rooms, projects) and relationships (hierarchies, project assignments, floor plan) in the sample domain.
2. A think-aloud protocol showing how an expert solves a particular office assignment problem.

As there is only one protocol, it can occur in the remainder of this chapter that there is not sufficient information to make a particular (modelling) choice. Such a situation usually gives rise to a *knowledge engineering (*KE*) goal*: a topic for which further knowledge elicitation and/or analysis is necessary. We will point to these KE goals in the text and state what kind of assumptions we have made about its outcome in building the model.

We should also mention here that it is our goal to build a model and a system that reflects as closely as possible the reasoning process of the expert. It is not our goal to find an algorithm that, given the input, would produce the same or similar output.

The process which led to the construction of the model of expertise presented in this chapter roughly consisted of the following steps:

Initial observations Firstly, the protocol was used to make some initial observations about the nature of the task, e.g.:

- What kind of task is it: analytic, synthetic?
- Are there clearly identifiable sub-tasks?
- What can be said about the information and the knowledge that the expert uses?
- Does the task resemble some known (generic) task? If so, what are similarities and differences?
- Does it seem feasible to automate (part of) the task?

Tentative domain schema Subsequently, a first sketch was made of the types of domain knowledge that play a role in solving this task. This characterisation of domain knowledge is done *before* any detailed model construction for a dual purpose:

1. To guide and verify the process of model selection and/or decomposition: is the knowledge available for achieving this task.
2. To prevent as much as possible that domain knowledge is only specified because it is required by the particular problem solving method that was chosen to achieve the task.

   The chosen problem solving method will of course influence the required representation of domain knowledge. Our goal is however to specify such representations as much as possible as a *viewpoint* on the available domain knowledge. For example, in the office-assignment domain relations exists between particular employees and their roles in the department (*employee X has the role of head-of-group*). The fact that this relation can be used as *classification* knowledge is a method-(or use-)specific viewpoint.

Model selection & top-down model construction The next step was to specify the top-level task (in this case office assignment) in terms of sub-tasks and primitive inferences required for solving the problem. This model construction process consists of one or both of the following activities:

1. Selection of a predefined generic decomposition in sub-tasks and inferences: an interpretation model. The selection of such model is guided by characteristics of the task such as the nature of the input and output of the top-level task (e.g. an enumerable set of solutions suggests an interpretation model for an analytic task) and of the required types of domain knowledge (e.g. a model of the normal behaviour of a device). In Ch. 3 (Fig. 3.9) these selection criteria are represented in the form of a decision tree.

2. A (repeated) process of model decomposition. In the worst case, no (partial) interpretation model is available for the task at hand. The knowledge engineer then has to decompose the top-level task into sub-tasks and inferences (primitive leaf tasks) on the basis of the elicited data (in particular protocols).

   There are however also a number of other situations in which decomposition plays a role:

- *The top-level task is not a generic task for which an interpretation model can be selected, but is a compound, "real-life" [Breuker* et al.*, 1987], task.*

  In that case, the knowledge engineer will first have to decompose the top-level task to the level of generic tasks.

- *The decomposition given by the selected interpretation model is too coarse-grained.*

  The "inferences" in such a model are in fact complex sub-tasks that need further specification and decomposition to arrive at inferences that can be linked to fragments of domain knowledge. For example, many models for synthetic tasks in the KADS library of interpretation models provide only a first level of decomposition.[2] Also, even if a detailed interpretation model such as systematic diagnosis is selected, it is possible that this model needs further detailing for the task-domain at hand.

Often, there is an interplay between the selection of generic components and model decomposition. In the office assignment case the emphasis was on decomposition, as there was no detailed interpretation model available.

Refinement When a first (partial) model of expertise has been established through selection and/or decomposition, it will need to be refined. This refinement was in this case performed in two ways:

1. By formulating task structures (i.e. control relations between sub-tasks) and checking whether these task structures could serve as plausible explanations of the behaviour of the expert.

2. By trying to identify the types of domain knowledge that would be needed to carry out the various inferences, and checking whether this knowledge could be derived from the domain schema. If it is not derivable, the question arises whether it can be formulated as an extension of this theory and whether expertise data are available for formulating this knowledge. Often, this involves additional knowledge elicitation (KE goal).

The refinement process acts in a sense as a verification of the chosen decomposition.

In the next section, the initial observations about the office assignment problem are discussed. In Secs. 7.5-7.7 a description is given of the major product of the modelling process: the model of expertise. These contains a description of the underlying domain knowledge, of the process of model selection and decomposition, and of the resulting inferences and tasks necessary for solving the problem.

[2] One could view problem solving methods such as "propose & revise" [Marcus & McDermott, 1989], "cover & differentiate" [Eshelman, 1988] and "skeletal planning" [Musen, 1989] also as partial interpretation models that can be used as a starting point for a model of expertise.

7.4 Initial Observations

Initially, the protocol is our focus of attention. While reading the protocol, we noted the following features of the problem solving process of the expert:

- A first thing to note is that the office assignment problem is of a synthetic nature: the solution is not chosen from a given set of predefined solutions, but is constructed using knowledge about employees, rooms and allocation constraints.

- The expert appears to solve the problem in two steps: (i) *selecting* a particular (group of) employee(s), and (ii) *assigning* this (group of) employee(s) to a room.

- It seems that the selection process is based on a global plan of the expert, namely assigning employees in a particular order. This plan is however not explicitly mentioned by the expert. This assumption would need to be verified in a future session with the expert (KE goal).

- The ordering in the allocation plan is not an ordering of specific employees, but of *types* of employees, e.g. head of group, manager, etc. The underlying knowledge on which this ordering is based seems to be quite subtle. For example, it is not just based on a simple hierarchy of employee types, as one could be inclined to deduce from the fact that the head of group is assigned first: this would not explain why the secretaries are assigned before the manager and the heads of projects.

- The elements of the allocation plan are not just single employees. These elements can also be sets of employees that are assigned in a random order (heads of projects) or groups of employees that are assigned in blocks to a room (secretaries, researchers).

  If one requires of the final model that it indeed models the behaviour of the expert as closely as possible, then this would exclude every model or method in which employees are assigned one at a time.

- The expert does not backtrack in the protocol. There is no evidence of a verification and/or a revision process. Most existing models and systems for synthetic problems, e.g. [Chandrasekaran, 1990; Marcus & McDermott, 1989], contain such a verify/revise step. The absence of this step could very well be an artefact of the sample problem solved in the protocol. This should be a major topic for future sessions with the expert (KE goal). We will come back to this issue in the discussion section.

This is by no means meant to be a complete or even correct list.[3] Such initial observations focus however the modelling process (see the next section).

7.5 Domain schema

In the description of the domain knowledge we are mainly interested in a structural description: what types of knowledge are available in the problem description? For this schematic

[3] It is in fact the list that the author presented at the EKAW'91 Sisyphus workshop in Crieff, Scotland after a first reading of the sample problem.

description we use the constructs of the data-description language (DDL) proposed in Ch. 4: concepts, sets, properties, and relations between concepts, instances and/or expressions. Fig. 7.2 gives a graphical overview of the structure of the domain knowledge described below.

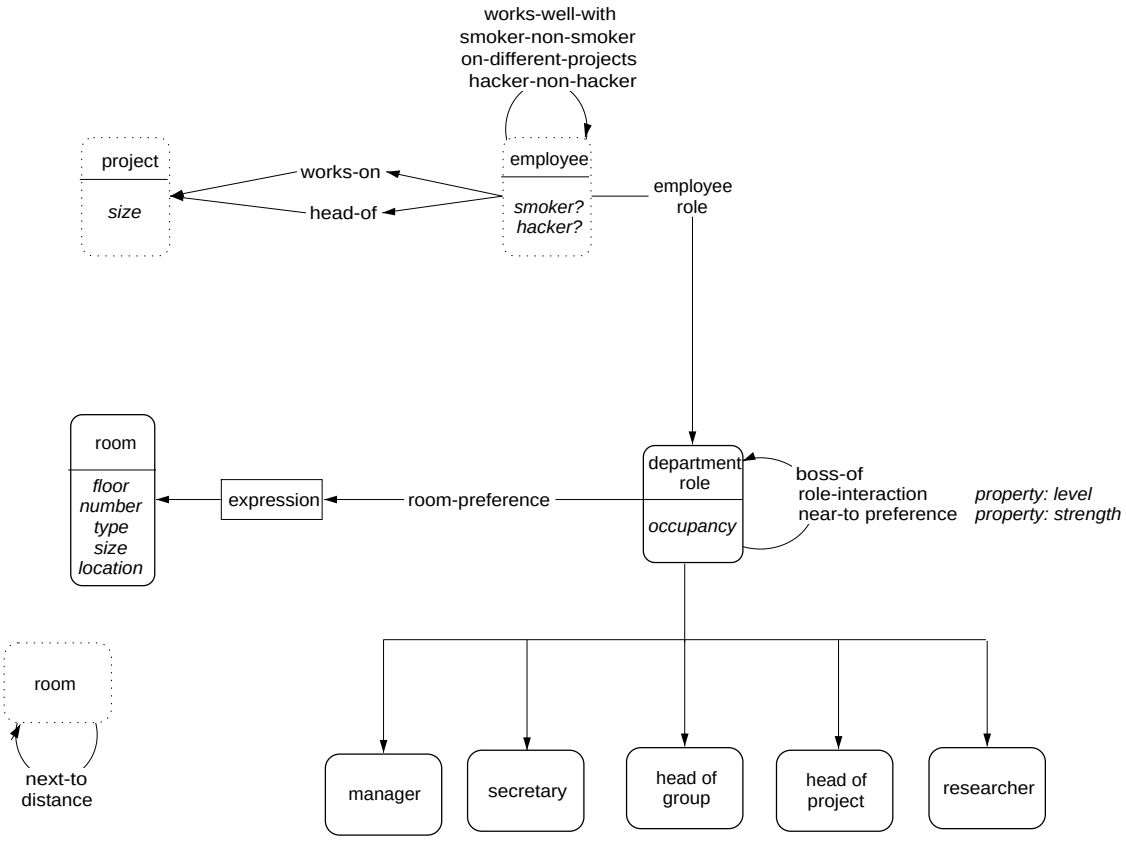

FIGURE 7.2: Schema of the domain knowledge in the office-assignment domain. See Fig. 4.5 for an explanation of the graphical notation used.

Employees and rooms stand out as central concepts in this domain. Employees have properties (such as whether they smoke or like to hack) and relations with projects they work on or are the head of. Also, a number of relations between two employee instances seem to be important: a smoker and a non-smoker, employees working on different projects, etc. Rooms have a number of properties (size, number, type, etc.) and relations with other rooms (distance, next to).

A DDL description of the concept *employee* and of one relation between employee instances is given below. A full DDL description of the domain schema is listed in Appendix A.

```
concept employee;
  properties:
    smoker: [true, false];
    hacker: [true, false];

relation on-different-projects;
```

argument-1: instance(employee);
argument-2: instance(employee);
semantics: associative;
axioms:
∀ E1,E2:employee, P1,P2:project
on-different-projects(E1,E2) ↔
works-on(E1, P1) ∧ works-on(E1, P1) ∧ P1 ≠ P2;

Another central concept in this domain is the notion of a *department role*: head of group, secretary, etc. As observed in the previous section, the expert seems to base most of his allocation decisions on properties of employee *types* and not on individual employees. The employee types are represented as sub-concepts of department-role (see Fig. 7.2).

Several types of relations concerning department roles seem to be important in the domain:

- A hierarchy of roles (e.g. the head of group is the boss of the manager)
- The amount of daily interaction (e.g. a high level of interaction between head of group and secretary)
- Positional preferences (e.g. the head of group should be near to a secretary)
- Relations between department roles and expressions about rooms, denoting room preferences (e.g. the head of group should have a large, central room).

This room-preference relation is represented in the DDL as follows:

relation *room-preference;*
argument-1: department-role;
argument-2: expression(room);
tuples:
< department-role, type(room) = office >
< head-of-group, location(room) = central >
< head-of-group, size(room) = large >
< head-of-project, size(room) = small >
< researcher, size(room) = large >
< manager, size(room) = small >
< secretary, size(room) = large > ;

The intended interpretation of such relation tuples is described in Ch. 4. For example, the statements about "head of project" should be interpreted as "all heads of projects need to get some small, single room".

7.6 Task classification and model selection

The office-assignment task takes as input a set of employee instances and a set of room instances and produces as output a set of allocations of rooms to employees. The office-assignment task can be classified as a design task: although the solutions are in principle enumerable for a given input problem, in practice the solution is not selected, but constructed.

In [Chandrasekaran, 1988] three classes of design tasks are described: creative design tasks, routine design tasks and a mix of routine and creative design. The prime property of routine design is that the elements from which the solution is constructed are known in advance. Office-assignment can thus be classified as a routine design task.

Puppe distinguishes three sub-classes of routine design tasks [Puppe, 1990]: planning, configuration and allocation (in German: "zuordnung"). According to Puppe, the main features that distinguish allocation from planning and configuration are:

- It operates on at least two disjunct sets of objects,
- The solution consists of allocation relations between objects of different sets that satisfy particular requirements.

Office-assignment is thus clearly an allocation task. The two disjunct sets of objects are in this case the employees and the rooms.

Unfortunately, the KADS interpretation model library in [Breuker *et al.*, 1987] does not contain a model for allocation. In such a case, it can be useful to look at a more general model for design tasks and use this is a starting point. Such a model for a more general task provides however only a first level of decomposition.

Chandrasekaran describes methods for routine design tasks [Chandrasekaran, 1988]. The general structure of the design task is presented as consisting of of three major sub-tasks: propose, critique and modify. For each sub-task a number of methods are described (informally) that can be used for realising the task. For example, the propose task can be realised with decomposition methods, with constraint satisfaction, etc.

The SALT system [Marcus & McDermott, 1989] implements a similar model for routine design called "propose & revise". The propose step proposes a value for a design parameter. Design parameters are linked to design constraints. When a constraint violation is detected, the revise task is activated to suggest changes ("fixes") to the design. This process is iterated until all design parameters have a value and no constraints are violated.

The mixer-configuration system [Wielemaker & Billault, 1988] design starts with building an ordered list of "duties" (i.e. design requirements). The first duty of the list (the "top duty": the requirement which is considered to be the most critical one) is used to generate an initial configuration, which is subsequently tested and refined on the basis of the other requirements. If a conflict arises, e.g. because some requirement cannot be satisfied, this duty becomes the top-duty and the design is modified.

In each of these models, the general structure of routine design appears to have an iterative structure: first, a (partial) solution is proposed, which is subsequently verified and if necessary adapted and /or refined, This leads to a new proposal and thus starts a new cycle of verification and adaptation/refinement.

As noted in the previous section, the expert in the sample protocol seems to carry out only the *propose* task. We limit the modelling enterprise in this chapter to a study of the nature of this propose task. However, this apparent absence of verification and revision should be a major focus for further knowledge engineering,

7.7 Model construction

Initially, we observed (Sec. 7.4) that this propose task seems to consist of two steps: selecting an employee and assigning her to a room. Also, the point was made that this selection step seemed to be based on a global allocation plan. In other models for design tasks the notion of a plan also appears. For example, in the mixer-configuration system [Wielemaker & Billault, 1988] the notion of a plan plays a role in terms of an ordering of

requirements. "Tackle the most difficult requirement first" appears to be a quite general strategy in design tasks. We will assume here that the expert indeed has some allocation plan. The precise nature of this plan is discussed below. As already pointed out, this assumption would need to be verified during further knowledge engineering (KE goal).

This gives us a first decomposition of the propose task (see Fig. 7.3). This figure (and also Fig. 7.5, see below) should be interpreted as a *provisional* inference structure. It fulfills the role of a working hypothesis in the knowledge engineering process. It can (and will) be refined in the process of model construction, e.g. through task decomposition and knowledge differentiation (see also Ch. 5).

As task and inference knowledge are described in a domain-independent way, we coin the general role names *component* and *resource* to talk at the task and inference level about employees and rooms. This is one way of enabling a potential reusage of (part of) the resulting model for another resource allocation domain.

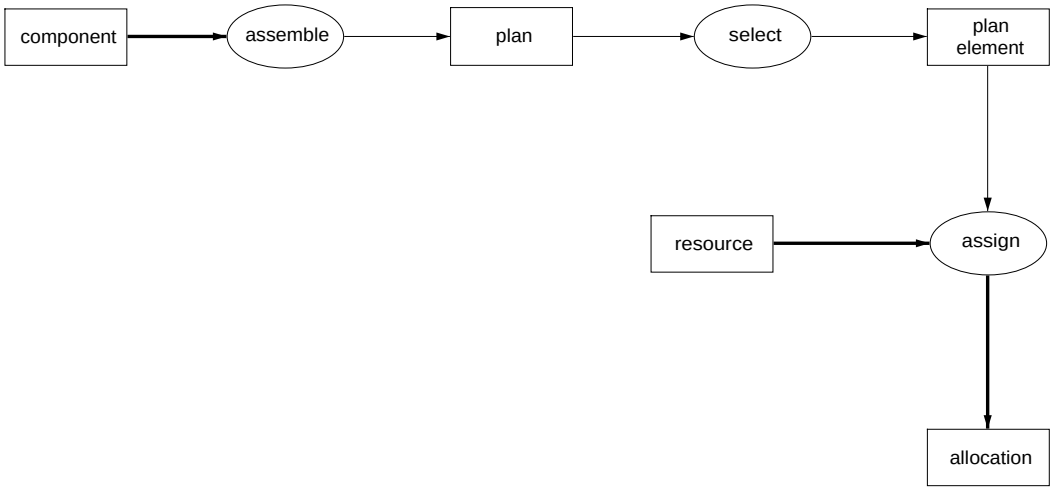

FIGURE 7.3: First provisional structure of the propose task. See Sec. 5.2 for a description of the graphical notation used.

The task structure of the propose task is specified below in a structured-English format. The top-level task *propose-allocations* consists of two major steps:

- *Assemble plan* which generates a plan in which the allocation order of components is specified
- *Assign resources* which produces parts of the solution. This last step is carried out for each element in the plan.

task *propose-allocations*
input:
components: set of components to be allocated
resources: set of available resources
output:
allocations: set of tuples <resource, set of components>
control-terms:
plan: list of (sets of) components representing an allocation ordering
plan-element: (set of) component representing an element of the plan
task-structure:

```
propose-allocations(components + resources → allocations) =
  assemble(components → plan)
  FOREACH plan-element ∈ plan DO
    assign-resources(plan-element + resources → allocations)
```

We use the format proposed in Ch. 3. The slots *input*, *output* and *control-terms* describe the data manipulated by the task, such as single objects, tuples, sets and lists. The *task structure* specifies the sub-tasks and their control dependencies in the form of a piece of pseudo-code. The arrows in the task structure describe the relation between input and output of the task or sub-task.

7.7.1 Plan assembly The question now arises whether it is possible to identify one inference that can generate the plan, or whether plan assembly should be considered a non-primitive task that requires further decomposition. To resolve this question we turn back to the protocol.

We noted the following characteristics of the way in which the expert orders the assignment of components:

1. The ordering is not based on individual components, but on *component types*: the expert does not talk about specific employees, but about the head of group, the secretaries etc. This means that during plan assembly it is necessary to classify components (the input of the assembly task) in terms of component *types*.

2. The head of group is assigned first, because this assignment "restricts the possibilities of subsequent assignments" (fragment 1 of the protocol). There is a similarity here with other models of design tasks, such as the model of the mixer configuration system [Wielemaker & Billault, 1988]: the component which is expected to impose the heaviest constraints on the final solution is tackled first. The allocation plan represents an implicit ordering of requirements: not the requirements themselves are ordered, but the component types to which they are related.

3. The other component types are ordered on the basis of the level of required access to and interaction with the head of group (fragments 2-4).

These observations led us to the formulation of three inferences that are needed to carry out the plan assembly task:

- *Classify* components as component types.
- *Select* the component type with the highest associated constraints.
- *Sort* the other components types relative to the one that imposes the highest constraints.

These three inferences are described in detail below. The inference structure in Fig. 7.4 shows the dependencies between the inferences for plan assembly. An important point of the specification of inferences is to indicate for each inference how its functional terms (meta-classes, domain view) relate to available domain knowledge. This will often reveal that some type of domain knowledge is lacking and can thus lead to new KE goals.[4]

[4]Here we will only describe inferences that use knowledge described in Fig. 7.2, but it is fair to say that this is an artefact of a post-hoc description.

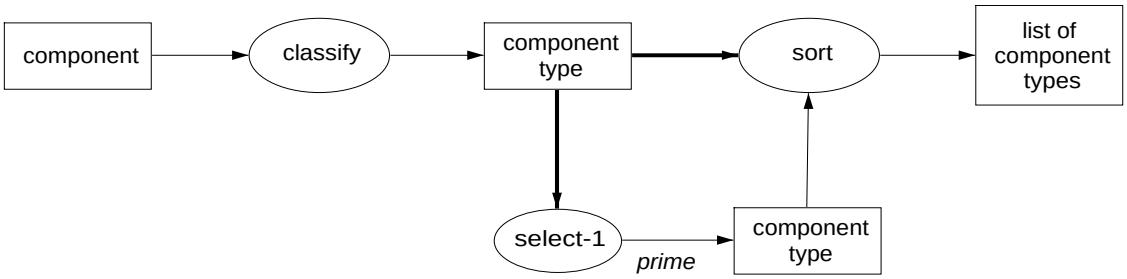

FIGURE 7.4: Inferences for plan assembly.

Classify The *classify* knowledge source uses the domain relation *employee-role* (see Fig. 7.2) for classifying a component (an employee instance) as a component type (i.e. a department role).

knowledge-source *classify*
input-meta-class
component → employee
output-meta-class
component-type → department-role
domain-view
type associations from component to component-type →
employee-role(employee, department-role)
description
knowledge-base look-up

The arrows in the description above show how names at the inference level map onto domain terms. The meta-classes can be seen as the data elements that are being manipulated by the knowledge source. The domain view describes the static knowledge that is used in this inference. The "description" slot gives an indication of how the output could be generated from the input and the domain view. This allows the knowledge engineer to make some remarks about possible computational methods. The actual selection of a computational method (which could turn out to be a different one) is part of the design process (see Sec. 7.8).

Select prime The *select-1* inference is used in the plan-assembly task to find the component type with the highest requirements. This knowledge source uses a domain relation *boss-of* to find the highest node in the component-type hierarchy[5]. This component (for which we will use the term "prime") is assumed to be the most critical one to assign (fragment 1 of the protocol).

knowledge-source *select-1* (select prime)
input-meta-class
component-types → set of department-role
output-meta-class
prime → department-role

[5]In retrospect, this is probably a suboptimal specification, because it makes unnecessary strong assumptions about the nature of the domain knowledge. It is conceivable that in other tasks other types of domain knowledge than hierarchies could be used to select the component with the highest associated constraints.

domain-view
hierarchy of component-types →
boss-of(department-role, department-role).
description
find the top node in the hierarchy of component types

Sort As remarked in Sec. 7.4, the other components are sorted on the basis of the amount of interaction that is required between certain types of components (see items 2-4 of the protocol).

knowledge-source *sort*
input-meta-class
prime → department-role
components-types → set of department-role
output-meta-class
component-types → list of department-role
domain-view
sort predicate →
value of the attribute "level" of the relation
role-interaction(department-role, department-role)
description
a component type is placed before another component type of the
level of required interaction with the prime is higher

Plan assembly tasks In the task-knowledge specification for the plan assembly task we have to indicate how the three inferences can be sequenced to achieve the goal of the task: the construction of a plan. The simplest solution would be to specify one task structure for plan assembly. However, the select and sort inference are so tightly connected, that we decided to view this as part of a separate sub-task *order*. A reason for this more detailed task decomposition is that one can envisage that in other domains this task could be realised with one inference.[6]

We thus end up with three tasks that specify the sequencing of inferencing in plan assembly: plan assembly and two sub-tasks: (i) a classification task, and (ii) an ordering task.

The plan-assembly task is specified as follows:

task *assemble-plan*
input: components
output: plan
control-terms:
component-types: set of components classes
task-structure
assemble-plan(components → plan) =
classify(components → component-types)
order(component-types → plan)

[6]Although this may sound a bit altruistic, the whole idea of "model construction for reusability" is so central to KADS approach that it tends to become a second nature for people involved in it.

The *classify* task requires a repeated invocation of the *classify* knowledge source plus a data operation (set unification).

task *classify*
  input: components
  output: component-types
  task-structure:
    classify(components → component-types) =
      FOREACH component ∈ components DO
        classify(component → component-type)
        component-types := component-type ∪ component-types

For readability purposes, the names of knowledge sources are italicised in the task structure.

The *order* task specifies a sequence of the select and sort inference and appends the output of both inferences to the resulting allocation plan.

task *order*
  input: component-types
  output: plan
  control-terms:
    prime: the component-type with the highest constraints
    other-components: the components minus the prime component
    ordered: the other components sorted with respect to constraints
        in relation to the prime
  task-structure
    order(component-types → plan) =
      select-1(component-types → prime)
      other-components := component-types/prime
      sort(other-components + prime → ordered)
      plan := prime , ordered

The "/" symbol represents a subtraction operator on a set; the "," symbol is used here to specify the order in a list. The resulting plan consists of an ordered list of component types.

7.7.2 Assign resources In *assign-resources* components of one particular type are allocated to a resource. Again, we turn to the protocol to study the inferences involved in assigning resources.

- As was noted in Sec. 7.4, if it concerns a multiple assignment (more than one component to one resource) the expert first groups these components into units of the right size using a special type of requirement concerning component interaction (avoiding conflicts and enhancing synergy, see protocol fragments 7-10). The type of assignment (single or shared) is fully determined by the component type (e.g. a head of a project should have a single room).
- The input for the actual *assign* task with respect to the components to be allocated can be of two types (see the remarks in Sec. 7.4):
  1. One single component (head of group, manager) or component group (secretaries).
  2. A *set* of components (heads of projects) or component groups (researchers).

If the input is a set, the assignment order of its elements should be random, as the expert indicates in the protocol explicitly that there is no particular reason for his sequencing of, for example, assignments of head of projects and pairs of researchers (fragments 4-6 and 8).

These observations lead us to a first refinement of *assign resources* by introducing an additional *group* step. This refined structure of the *assign* step in Fig. 7.3 is shown in Fig. 7.5. The corresponding task-knowledge specification is given below:

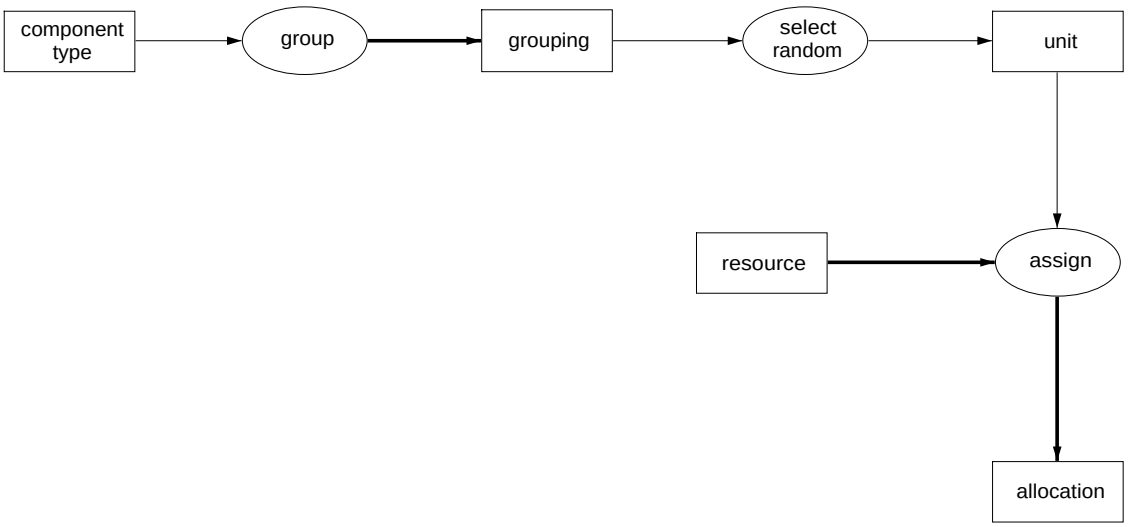

FIGURE 7.5: First refinement of the *assign* step of Fig. 7.3 by introducing a group step which generates possible groupings and a random selection of a unit (a component or set of components to which one resource will be assigned).

task *assign-resources*
  input:
    component-type: type of component allocated in this plan step
    resources: available resources
  output:
    allocations
  control-terms:
    unit: a component or set of components
    grouping: set of units
    suitable-groupings: groupings satisfying particular constraints
  task-structure
    assign-resources(component-type + resources → allocations) =
      group(component-type → suitable-groupings)
      select-random(suitable-groupings → grouping)
      REPEAT
        select-random(grouping → unit)
        assign(component-type + unit + resources → allocations)
      UNTIL grouping = ∅

The *group* step is only interesting for components that share resources. For other component types we assume it is a kind of no-op. The random selection of units in the REPEAT loop ensures that, for example, heads of projects are really assigned in a random

order. This also implies, that the specification differs here slightly from the assignment order in the protocol. There, a unit of two researchers is assigned directly after grouping. As the expert indicates that there is no special reason for this (except maybe mental hygiene) we have separated in our model the grouping of units from the actual assignment of units

It might be useful to note that the introduction of a separate *group* step implies a differentiation of allocation requirements into two major types: (i) requirements concerning interaction of components with respect to one resource (conflicts, etc.), and (ii) resource-specific requirements (room preferences, etc.). This is in fact a role differentiation at the level of the model of expertise that can make the resulting system more efficient (cf. Sec. 2.4). For example, a computational technique implementing one of these subtasks would need to handle less requirements and operate on a smaller set of components (because some of them are already grouped into units).

In the next sections, we study the group and the assign step in more detail.

7.7.3 Group When components are grouped together for joint assignments to one resource, a different kind of domain knowledge comes into play, namely knowledge about possible effects of the joint usage of the resource. The expert tries to minimise negative effects and support positive ones as much as possible. This grouping of components into appropriate units (fragments 7-10 in the protocol) appears to be the only part of the resource allocation process where the expert uses requirements based on properties of *individual* employees: e.g. whether they smoke or on which project they work, etc.

Generating suitable groupings is typically a task where one would specify another problem solving method for a machine than the one the expert employs. The expert generates in the protocol partial groupings based on the requirements. This partial grouping is in fact one of a set of possible partial groupings. Given the limited size of human short-term memory it is usually impossible to consider all possible solutions. For a machine however, this storage problem does not exist. On the other hand, the somewhat ad hoc, intuitive way in which the expert generates a grouping would be quite difficult to model for machine execution.

Thus, we decided to drop for this subtask the general guideline of modelling the expert as closely as possible and model the grouping task as consisting of two types of inferences:

- A *transform* inference, which generates all possible groupings.
- A *select* inference which selects a subset of groupings that satisfies particular requirements.

The transform inference is described below:

knowledge-source *transform*
input-meta-class:
component-type → department-role
output-meta-class:
possible groupings: set of employee structures
domain-view:
-
description:
generate all possible groupings of components of this type

For the select inference ("select-2", to distinguish it from the previous select inference) one has to decide what the requirements should be for suitable groupings. The following types of requirements are mentioned by the expert:

- *Conflicts* The expert tries to minimise conflicts. Putting a smoker and a non-smoker together is considered a major conflict (fragment 7) and should have a high impact. Putting a hacker and a non-hacker together is only a minor conflict that could be allowed if more important reasons exist for preferring such a grouping.
- *Synergy* The expert also tries to maximise synergy. Putting employees together that work on different projects is considered by the expert as an important type of synergy (fragment 8). Also, grouping researchers working on similar subjects is considered synergetic, although to a lesser degree (fragment 10).

The select inference specifies the selection of a subset of groupings given one particular criterion (some conflict or synergy). Based on the observations above, we distinguish four types of criteria: minimise major/minor conflicts and maximise major/minor synergy. This choice would have to be verified in future sessions with the expert (KE goal).

The dependencies between these two inferences, which constitute a refinement of the *group* step in Fig. 7.5, are shown in Fig. 7.6.

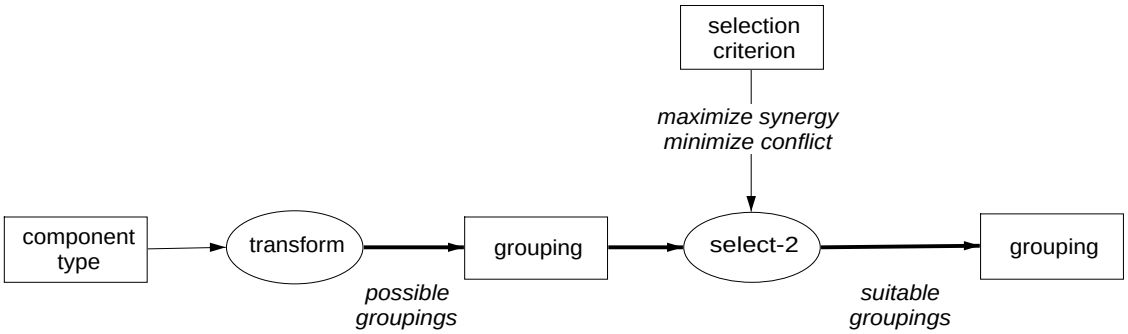

FIGURE 7.6: Inferences for generating suitable groupings of components.

knowledge-source *select-2 (select suitable groupings)*
input-meta-class:
groupings → set of employee structures
selection-criterion → a conflict- or synergy-type
output-meta-class:
suitable-groupings → set of employee structures
domain-view:
major-conflict → smoker-and-non-smoker relation
minor-conflict → hacker-and-non-hacker relation
major synergy → on-different-project relation
minor-synergy → works-with relation
description:
generate the subset of all possible groupings
that minimises some conflict or maximises some type of synergy.

This distinction between four different types of criteria can be considered as an example of *inference differentiation* (cf. Sec. 5.3 and Fig. 5.12): the *select-2* inference can be differentiated into four sub-types each using a different type of criteria.

In the task-knowledge specification of *group* we have to decide in which order these four possible instantiations of the *select-2* inferences should be executed. The order "avoid major conflict, increase major synergy, increase minor synergy, avoid minor conflict" seemed to conform most to the way the expert solves the grouping problem. Again, this hypothesis would need to be verified (KE goal).

task *group*
input:
component-type: the type of components being grouped
output:
preferred-groupings: the optimal sub-set of groupings given the selection criteria
control-terms
possible-groupings: all possible groupings of the components of this type
task-structure
group(component-type → suitable-groupings) =
transform(component-type → possible groupings)
select-2(possible-groupings + minimise(major-conflict) → preferred-groupings)
select-2(preferred-groupings + maximise(major-synergy) → preferred-groupings)
select-2(preferred-groupings + maximise(minor-synergy) → preferred-groupings)
select-2(preferred-groupings + minimise(minor-conflict) → preferred-groupings)

In Appendix B (Sec. B.5.2) a sample trace is listed of the execution of the group task for the researchers in the Sisyphus data set. The *transform* inference generates 105 possible groupings. Avoiding major conflicts reduces this set to 15. Maximising synergy by putting people on different projects together reduces this set further to 10 possible groupings. Maximising synergy by grouping people that work on similar subjects reduces the set of ten to two groupings. The last inference (reducing hacking conflicts) has no effect in this particular case.

The two groupings generated by the program differ slightly from the grouping generated by the expert. This is due to the fact that we assumed that the "works-with" relation in the sample data set represented the notion of working on similar subjects that the expert talks about, Probably, this was not a correct assumption and should be noted as a KE goal. However, this type of refinement does not affect the structure of the model and can be carried out in a later knowledge-refinement phase.

7.7.4 Assign In the assign task resources are allocated to components or groups of components on the basis of various requirements. We distinguished two types of such requirements:

1. *Resource specific requirements* Requirements about a resource independent of other allocations: required size, required location, etcetera.
2. *Positional requirements* Requirements about a resource that are *dependent on other allocations*: e.g. a room is required as close as possible to the head of group.

These requirements are the same for every component of a particular type.

Thus, we defined two select inferences *select-3* and *select-4* each selecting a subset of resources that respectively satisfy resource-specific and positional requirements. The *select-4* inference has as an additional input the current set of allocations.

Fig. 7.7 shows the dependencies between these two *select* inferences. This figure represents a further detailing of the *assign* step in Fig. 7.5.

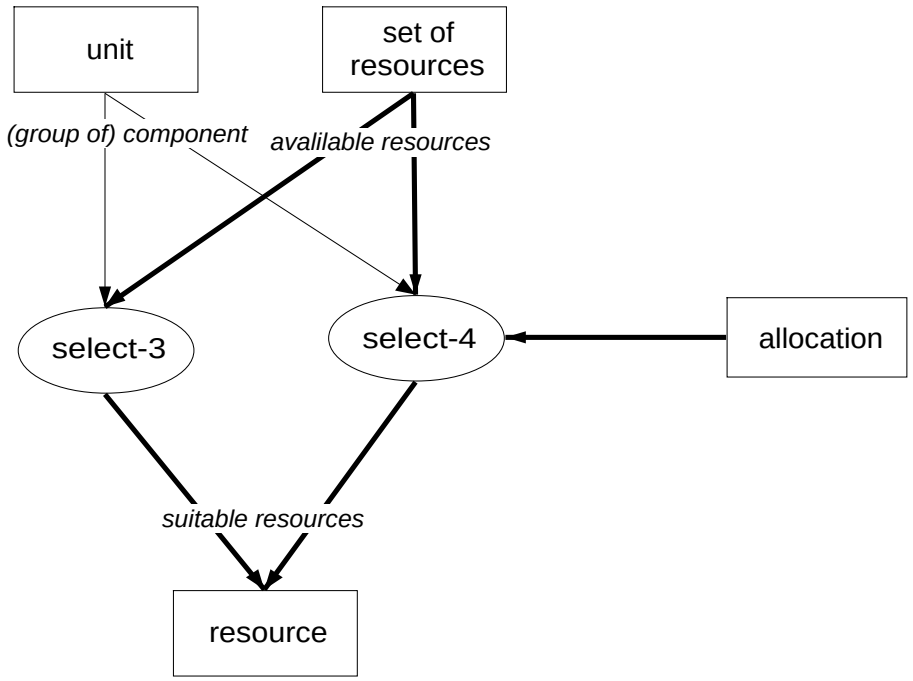

FIGURE 7.7: Inferences for resource selection.

knowledge-source *select-3* (select on resource requirements)
input-meta-class:
component-type → department-role
resources → set of rooms
output-meta-class:
suitable-resources → set of rooms
domain-view:
resource-requirement → room-preference relation
description:
select the subset of resources that satisfies
resource-specific requirements

knowledge-source *select-4* (select on positional requirements)
input-meta-class:
component-type → department-role
resources → set of rooms
output-meta-class:
suitable-resources → set of rooms
domain-view:
resource-requirement → near-to-preference relation
description:
select the subset of resources that satisfies
positional requirements

Note that the unit to which a resource will be assigned is input to neither of the two inferences. This is consistent with the fact that resources are only selected based on requirements connected to a component *type*. The main decision that has to be taken when defining control over these inferences, is which one should be executed before the other (or maybe in parallel). In the current task structure *select-3* is executed before the *select-4* inference. This implies that we give a higher priority to a resource-specific requirements. If, after execution of both inferences, more than one resource is considered suitable, one is selected at random.

task *assign*
  input
    component-type:
    unit: the component or group of components that to which a resource is assigned
    resources: available resources
    allocations: current allocations
  output
    resources: available resources
    allocations: current allocations
  control-terms: =
  task-structure
    assign(<component-type + unit + allocations + resources → allocations + resources)
      select-3(component-type + resources → suitable-resources)
      select-4(component-type + suitable-resources + allocations → suitable-resources)
      select-random(suitable-resources → resource)
      allocations := < unit, resource > ∪ allocations
      resources := resources/resource

In Appendix B (Sec. B.5.3) a sample trace is listed of the execution of the assign task for the manager. In the example, *select-3* generates four suitable rooms: the four small rooms. *Select-4* selects from this subset the room closest to the one allocated to the head of group.

The full inference structure that resulted from the model construction process for this model of resource allocation is shown in Fig. 7.8. Fig. 7.9 shows the resulting task decomposition.

7.8 Operationalising the Model of Expertise

In this section we describe some aspects of the design and implementation of a system that implements the behaviour specified in the model of expertise. In the design of the system we follow the structure-preserving principle as defined on Ch. 6: all relevant elements of the conceptual model should map onto clearly identifiable constructs in the system. The advantages of such a design approach are (see also Sec. 6.3):

- It simplifies the implementation of an explanation facility that enables the user and/or the expert to trace the system's execution in the vocabulary of the model of expertise. Although we have not build a graphical interface for this particular case, we have tried to ensure that all the necessary anchor points for such an extension are present.
- It provides clear routes for refining and/or extending the system, such as:

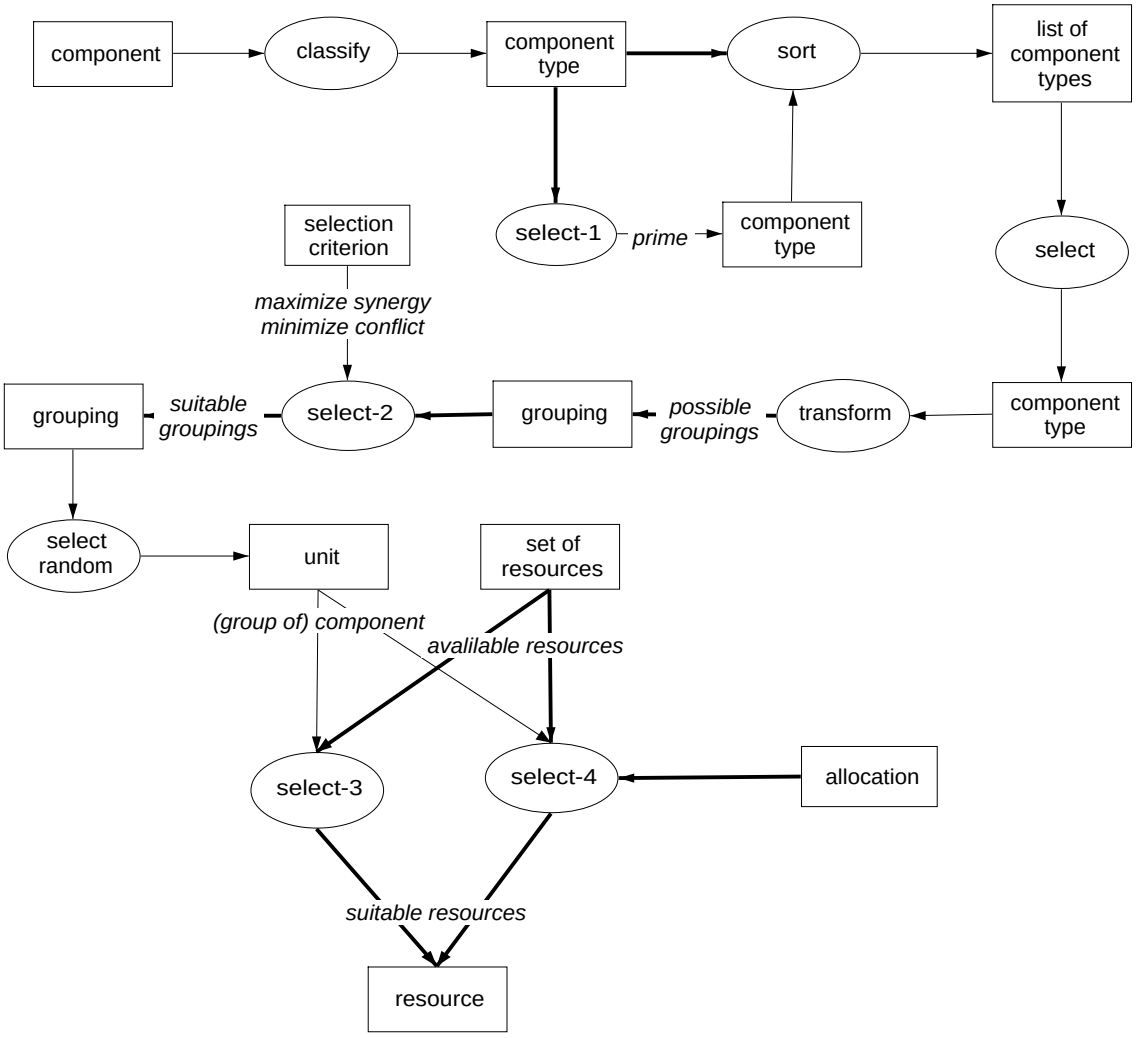

FIGURE 7.8: Inference structure for resource allocation in the office-assignment domain. The figure summarises the results of the various decompositions and refinements of the first model in Fig. 7.3.

1. adding/modifying domain knowledge such as other conflicts or room requirements;
2. changing the control of task execution;
3. replacing computational techniques;
4. introducing additional tasks and inferences such as for verification and revision.

No special-purpose tools were used in the development of this system. Also, the fact that no run-time interaction with external agents such as a user is required simplifies the system development. The chosen environment was the SWI-Prolog system [Wielemaker, 1991], mainly for pragmatic reasons. The system architecture is an instantiation of the skeletal architecture described in Sec. 6.4. Modules were used to support the separation of various elements of this architecture (see Fig. 6.5). Fig. 7.10 gives an overview of the various Prolog modules. The source code of the application plus some example traces can be found in Appendix B. A synopsis of the contents of each module is given in the rest of

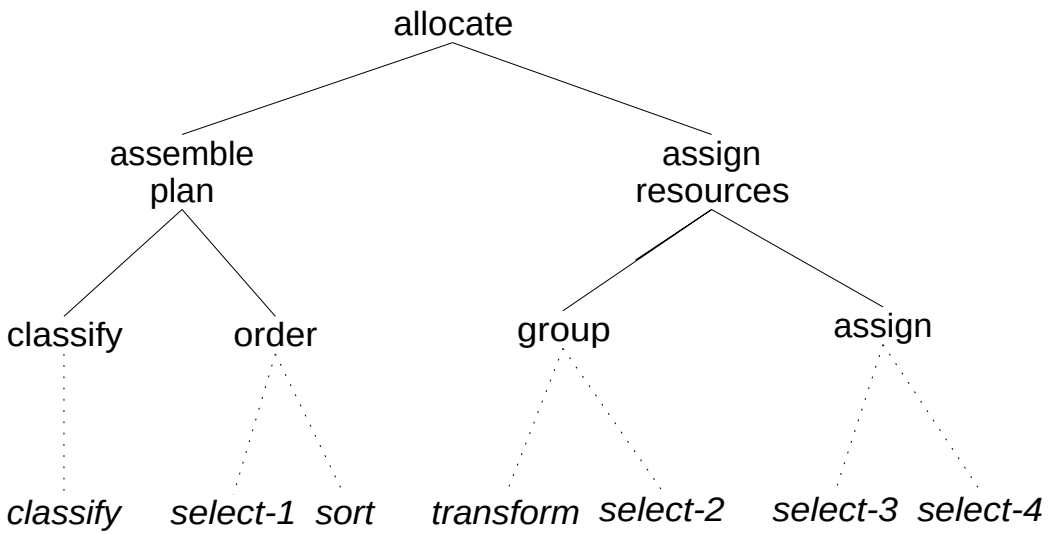

FIGURE 7.9: Task decomposition of the office-assignment problem. Italic names denote knowledge sources. Two trivial *select* inferences (*select-next* and *select-random*) have been left out

this section.

Task-declarations, inference-declarations and domain-theory These three modules contain an almost direct translation of the contents of the model of expertise task into a predefined format of Prolog clauses.

Below, the task declaration of the *classify* task is listed, It is an almost direct mapping of the description given in the previous section. The data operations in the original task structure (set membership, set addition) was translated into the format of the access functions defined in the module *task-working-memory* (see below).

```
task(          classify).
task_input(    classify,      'components').
task_output(   classify,      'component types').

task_structure(classify,
  ( forall( data_operation(member, components, C),
      ( exec_inference(classify, [C], CType)
      , data_operation(add, 'component types', CType)
      ))
  )).
```

An inference declaration is described in a similar fashion. The only difference is that all mappings from inference-level names to domain-specific constructs are specified in a separate module *domain index* (see below). The example below shows the Prolog facts associated with the *classify* inference:

```
% inference(Internal name, External name)
% metaclass(Inference,     Input/Output,      General name, Specialised name).
% domain_view(Inference, , Inference knowledge).

inference(  classify,   'Classify components').
```

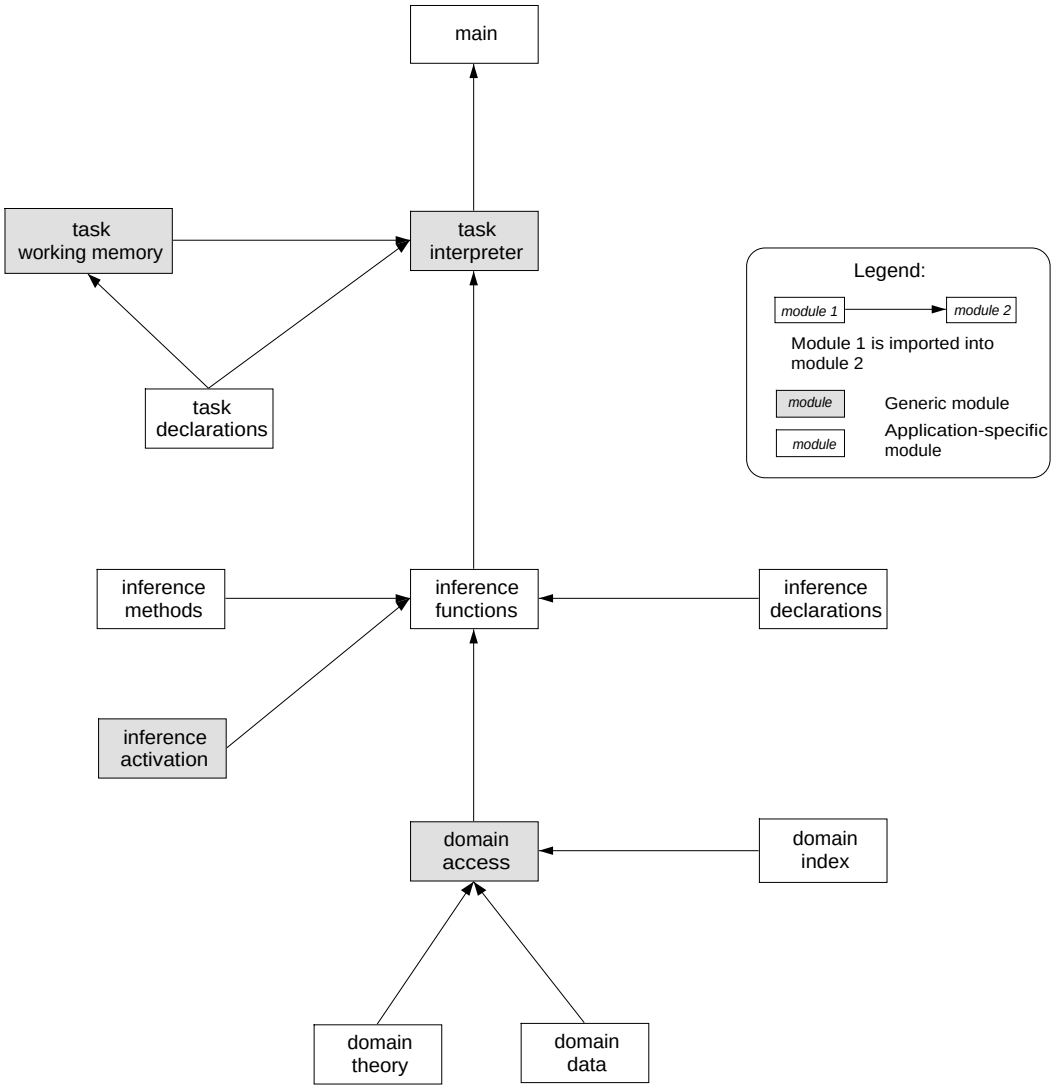

FIGURE 7.10: Import relations between Prolog modules of the Sisyphus system.

```
metaclass(  classify,   input(1),       component,      _).
metaclass(  classify,   output,         component_type, _).
domain_view(classify,   relation(type_association, component, component_type)).
```

The domain-theory module defines a language on top of Prolog which allows the declaration of concepts, instances, sets and various types of relations and also the possibility of defining properties for each of these constructs (not just for concepts). In addition, it is possible to specify semantic information about relations (associativity, transitivity) and sets (cardinality).

This language was used to describe the domain schema as presented in Fig. 7.2 and also the actual domain knowledge (concept hierarchies, relation tuples, definitional axioms). An example fragment of the representation of the domain knowledge used by the *classify* inference are listed:

```
% concept(Concept name,     Supertyes)
% property(Concept,         Property name,          Valueset)
% relation(Relation name,   Type first argument,    Type second argument)
% tuple(Relation name,      [First argument,        Second argument])

concept( employee).
property(employee,              hacker,                 bool).
property(employee,              smoker,                 bool).

concept(department_role).
concept(head_of_group,          [department_role]).
concept(manager,                [department_role]).
concept(secretary,              [department_role]).
concept(head_of_project,        [department_role]).
concept(researcher,             [department_role]).

relation(employee_role,         instance(employee),     department_role).

tuple(employee_role, [Employee, head_of_project]) :-
    get_instance(project, Project),
    get_value(Project, size, large),
    tuple(head_of, [Employee, Project]).
tuple(employee_role, [Employee, researcher]) :-
    tuple(works_on, [Employee, _SomeProject]),
    \+ tuple(employee_role, [Employee, head_of_project]),
    \+ tuple(employee_role, [Employee, head_of_group]).
```

Task-interpreter and task-working-memory The *task interpreter* module executes the task structures defined in the module *task-declarations* and stores intermediate results in data stores. These data stores and their access operations (cf. the *data-operation* clause in the Prolog task structure above) are defined in *task-working-memory*. The implementation supports three types of working-memory data structures: set, list (= sets with an ordering relation, used for example to represent the allocation plan) and a single object.

Inference-functions, inference-methods, and inference-activation The module *inference functions* defines for each knowledge source how inference methods should be activated to realise the inference. In addition, it retrieves the necessary domain knowledge by calling domain access functions. In fact, the *classify* inference function only retrieves domain knowledge:

```
inference_function(classify, [In], Out) :-
    domain_retrieval(find_one, type_association, {In, Out}),
```

The six other inferences in the office-plan model are realised through four inference methods (see Table 7.2). The *partition-set method* is used to realise three select inferences, all selecting a subset based on some criteria. All inference methods are defined in the module *inference-methods*. Ideally, one should have a large library available of such methods.

The module *inference-activation* contains the generic part of the *inference-functions module* (e.g. how to produce trace information).

Method	Method description	Used for
Hierarchy search	Supports search in hierarchical relations, in this case finding the root node of the hierarchy	*select-1*
Sorting	Supports sorting given a predicate that compares two members of the set being sorted (built-in SWI-Prolog predicate)	*sort*
Pair permutations	Generates all possible permutations of pairs of set elements	*transform*
Partition set	Partitions a set into an ordered list of subsets based on a predicate that assigns a rating number to each element of the set. The method can be used to either maximise or minimise this rating.	*select-2* select-3 *select-4*

TABLE 7.2: Inference methods used in the implementation of the Sisyphus model

Domain-access and domain-index The *domain-access* module defines a number of access functions for the domain knowledge base. The module uses the indexing information defined in *domain-index* to map inference-level names onto domain-specific ones. The access functions are used by the inference functions to retrieve domain knowledge (cf. *domain-retrieval* in the example inference function above). This is a typical part of KBS from which one abstracts in the model of expertise: it is specific for the representation chosen in design.

The domain index is a specialisation of the meta-class and domain view mappings defined in the model of expertise. These last ones are typically defined in a sloppy manner during analysis. Below the mappings used by the classify inference are listed:

```
domain_index(entity,     component,          [instance(employee)]).
domain_index(entity,     component_type,     [concept(department_role)]).
domain_index(relation,   type_association,   [relation(employee_role)]).
```

Domain data This module contains the example data set provided for the Sisyphus problem in the format of the knowledge base representation used in the module *domain theory*: (i) employee, room and project instances with their associated property values, and (ii) some relation tuples that are not part of the domain theory: employee-project tuples and some employee-role tuples. Some sample data:

```
instance(employee,    'Werner L.',   [smoker = false,    hacker = true]).
instance(project,     'RESPECT',     [size = medium]).

tuple(works_on,       ['Werner L.',  'RESPECT']).
tuple(employee_role,  ['Eva I.',     manager]).
```

Main This module is the central module that invokes the top-level task. It could contain in future versions some additional strategic knowledge.

7.9 Discussion

How general and/or reusable is the model? A major assumption in KADS is that the description of task and inference knowledge is sufficiently domain-independent to

have the potential of being reused in a similar task domain. With regard to the model that was constructed for the office-assignment domain, the following tentative observations can be made:

- The notion of plan representing an ordering of requirements seems to be a quite general one: it reoccurs in many constructive task-domains.
- The differentiation into various types of requirements can be useful. In some domain, e.g. allocating air planes to gates, the component-interaction requirements will not be relevant (only one plane per gate), thus leading to a simplified version of this part of the inference structure (the grouping inferences do not have to be included).
- The office-assignment domain contains a number of simplifications that could well not be present in other domains and thus may lead to more complex models, e.g.:
  - No time considerations come into play (no existing allocations, no planning of future allocations). This could be very important in a domain such as allocating air planes to gates.
  - Preferences of individual components are not considered in the selection of suitable resources.: only preferences of types of components.
- An obvious shortcoming of the model is that it covers only the propose task. In most domains, an iterative revision process is required.

  What would be needed to include the revisions in this model? The inclusion of a separate *revise* task and the additional control can easily be achieved by defining an additional task on top of *propose-allocations* which activates both the propose and the revise task. The main question is whether the revisions would require a different structure of the propose task. Some revisions can be achieved by relaxing the constraints, i.e. changing the domain theory and re-activating the propose task. For example, if not enough single rooms are available for all heads of projects, a revision might be to consider one of them (temporarily) as an ordinary employee. However, the nature of the revise task needs to be studied in more detail before a definite answer can be given,

Concerning the reusability of the domain knowledge, it can be said that the description of employees, rooms, projects, and department roles has a quite general flavour. On the other hand, some relations such as room preferences are rather specific for this task-domain.

Comparison with other approaches This exercise has made clear that there is quite some overlap between various approaches to modelling problem solving. As shown in this chapter, the problem solving methods described by Chandrasekaran (1988, 1990) and Marcus & McDermott (1989) could be used as input for a KADS modelling enterprise. We see two major differences between the Generic Task approach (as described in Chandrasekaran, 1990[7]) and KADS:

[7] The description given in this article is much more conceptual and therefore better comparable to KADS than other publications.

- The Generic Task approach makes the underlying problem solving method explicit (e.g. goal decomposition). In KADS this is implicit in the task knowledge description.
- In the Generic Task approach only the method description is domain-independent: its application to a task-domain is, unlike KADS, described in domain-specific terms [Allemang, 1991]. This limits the reusability of the resulting model.

In the computationally-oriented approaches, the underlying assumptions about the reasoning techniques supported by the approach tend to bias the problem solving model. For example, in a pure constraint-satisfaction approach the idea of grouping will usually not be considered and will also not be easy to include.

Weak points of the approach In this application of KADS, some weak points that have already been pointed at before (see Sec. 3.8), become very clear:

- If no interpretation model is available, the knowledge engineer has to construct a model almost from scratch.
- The typology of knowledge sources described in Breuker *et al.* (1987), and used quite rigorously in this chapter, does not provide always appropriate distinctions between inferences. For example, knowledge sources of type *select* appear in many places in the model presented and range from trivial selections to inferences involving complex knowledge structures (*select 2-4*).

Acknowledgement Werner Karbach provided valuable comments on an earlier version of this chapter.

Chapter 8

Comparing KADS to Conventional Software Engineering

In this chapter, a comparison is made between KADS and two leading software-engineering methodologies: Structured Analysis & Design and the Object Modelling Technique. In the comparison the emphasis lies on similarities and differences in analysis: the process of describing *what* the system should do. We compare the approaches with respect to three different perspectives on modelling a system: the data perspective, the functional perspective and the control (or: dynamic) perspective. The study shows that, though terminology is different at some points, there are quite a number of similarities between the approaches. We also study some important differences. We discuss some lessons that might be learned from these differences. A common topic that arises is that of reusability.

This chapter will be published in a collection of articles on KADS. It is co-authored by Bob Wielinga. Reference: Schreiber, A. T & Wielinga, B. J, (1993). Comparing KADS to conventional software engineering In Schreiber, A. T., Wielinga, B. J., & Breuker, J. A., editors, *KADS: A Principled Approach to Knowledge-Based System Development*. Academic Press, London.

8.1 Introduction

Knowledge engineering (KE) and conventional software engineering (CSE) are closely related fields. Although both fields emphasise different aspects of the system development process, there is no sharp boundary between conventional software systems and knowledge-based systems. The main features that distinguish a KBS from a conventional system are often said to be the nature of the task (problem solving) and the explicitness of knowledge (the knowledge base). But with the growing complexity of conventional systems, the borderline is at most vague. Also, the tendency is to use KBS applications not as stand-alone applications, but in combination with other, more conventional, applications.

The aim of this chapter is twofold:

1. To identify bridges between KADS and CSE methodologies: how do models, terms, techniques etc. map from one approach to another.
2. To identify common themes and suggest areas in which results achieved in one field could be of use for research in the other field.
   Both KE and CSE are still very much under development and sometimes the respective research communities seem to be more apart than ideally should be the case.

Given the context of this thesis, we limited the scope of comparison to the analysis stage and the role of the analysis results in system design. In Sec. 8.2 we discuss the type of analysis and design models that are distinguished in the approaches. In Sec. 8.3 three basic perspectives on describing a system are identified: the data, the functional and the control perspective. Sec. 8.4 discusses for each of the three perspectives the type of modelling techniques advocated by the approaches. Sec. 8.5 contains a brief discussion on the role of the analysis model in the design process. Sec. 8.6 summarises some conclusions with respect to the points raised above: bridges and cross-fertilisation.

The comparison is made with an emphasis on the content of the models and the modelling languages and methods, much less on the modelling *process*. The reason is that both in KADS and in CSE the descriptions of the actual modelling process are not prescriptive and often vague. Although Ch. 5 gives some indications about the modelling process in KADS and although there exist cognitive studies of the modelling process in CSE, a more thorough analysis is needed for a detailed comparison.

In this study we have limited the comparison to two examples of CSE approaches, each representing a leading development paradigm: the functional paradigm and the object-oriented paradigm. The example functional approach is Structured Analysis and Design which appears to be the leading methodology in this area. The major source of reference used in this comparison is the latest book on "Modern Structured Analysis" (MSA) [Yourdon, 1989b].

In the object-oriented area the choice was difficult. There is an abundance of books and articles on various flavours of "object-oriented". We have chosen the OMT (Object Modelling Technique) methodology [Rumbaugh *et al.*, 1991] as the example object-oriented approach for this study, because it assumes an object-oriented view point, but covers also many other aspects of the development process.

8.2 Models Distinguished

In this section we look at (i) what kind of models are being distinguished in KADS and CSE during analysis and design and (ii) what these models describe.

MSA In [Yourdon, 1989b; Chapter 17] two modelling approaches are sketched: the classical modelling approach and a more recent approach.

In the classical approach, four system models are important: two "physical" models, and two "logical" models. Physical models describe the detailed *implementation* of a system; logical models focus on the *essential requirements* of these systems. In other words, a logical model describes the *what* and a physical model describes the *how*. A further distinction is made between *current* and *new* models. The *current physical model* and the *current logical model* describe the current situation: the real world (organisation, enterprise, department, task) in which system development will be undertaken. The *new physical model* and the *new logical model* describe the target system to be developed, including its new[1] environment.

[1] This is consistent with observations that the introduction of new automated functions within an organisation often involves new distribution of tasks

Yourdon argues that this classical approach has failed, because in many cases it has proven difficult to build models of the current situation. Often, an enormous amount of time is spent on analysing the current situation. Especially, the construction of a current physical model in full detail is a time-consuming process, which only marginally pays off in later stages of the project.

Based on these observations, Yourdon argues that system development should focus more on modelling the new situation, and especially on the new logical model. He calls this model the *essential model*. He characterises the essential model as follows [Yourdon, 1989b; p. 323]:

> "The essential system model is a model of *what* the system must do in order to satisfy the user's requirements, with as little as possible (and ideally *nothing*) about *how* the system will be implemented."

Two sub-components are distinguished in an essential model:

- The *environmental model* defines the system boundary: what is the relation of the system with its environment.
- The *behavioural model* defines the required internal behaviour of the system necessary for performing its intended function in the environment,

The new physical model takes the role of the design model.

OMT OMT describes two models: the analysis model and the design model. The analysis model consists of three submodels: the object model, the dynamic model and the functional model. Whether this analysis model is a model of the current situation or the future situation is not completely clear. Some quotes from [Rumbaugh *et al.*, 1991]:

> "Starting from a statement of the problem, the analyst builds a model of the real-world situation showing its important properties." (p. 5).
>
> "During analysis, a model of the application domain is constructed ..." (p. 17)
>
> "The analysis model is a concise, precise abstraction of *what* the system should do, not *how* it will be done. The objects in the model should be application-domain concepts and no computer-implementation concepts such as data structures " (p. 17)

From the last quote, it can be concluded that the OMT analysis model is very similar to what Yourdon calls the essential model. The OMT analysis model should describe what the system should do, independent of how it will be realised in the artefact.

KADS Models distinguished in KADS are briefly described in Ch. 3. Fig. 3.2 in that chapter gives an overview on these models. We focus here on the main models relevant for a comparison with the models defined in CSE.

- The *task model* defines a top-level decomposition of tasks to be carried out in the application domain, together with their input-output relations. In addition, the task

model specifies the task distribution: an assignment of tasks to *agents*: (i) the system to be developed, (ii) users, and (iii) possibly other automated systems. The task model thus describes, from a high-level point of view what is called the *automation boundary* in CSE.

- The *model of expertise* describes for those tasks assigned solely to the system[2] the required internal behaviour of the system needed to carry out these tasks. The model of expertise is a so-called ' 'knowledge-level model" [Schreiber *et al.*, 1991a]: it describes the required problem solving behaviour in the vocabulary of users /experts and it abstracts from implementation details.

- The *model of cooperation* provides a description of those tasks that exchange information across the system boundary: e.g. require interaction with the user or an external system. It also abstracts from implementation details.

- Together, the model of expertise and the model of cooperation form the *conceptual model*. This conceptual model contains a complete specification of the functionality provided by the system, in implementation-independent terms.

- The *design model* specifies how this conceptual model will be realised in the artefact: what computational and representational techniques are needed to implement the requirements implied by the conceptual model.

Discussion It will be clear that, although terminology is different, there are many similarities between CSE and KE with respect to the models being distinguished. These similarities are summarised in Table 8.1

MSA	OMT	KADS
essential model	analysis model	task model + conceptual model
environmental model	part of analysis model	task model + model of cooperation
behavioural model	part of analysis model	model of expertise
new physical model	design model	design model

TABLE 8.1: Correspondences between models in the three approaches.

From this table it can be concluded that models distinguished in CSE and KADS are closely related. One question that comes up when inspecting this table is why a separate task model is considered necessary in KADS. The reason for this is that KADS distinguishes two stages in task description. The purpose of the first stage (described in the task model) is to decompose the application task down to the level where generic problem solving tasks can be identified. KADS provides a set of reusable templates for a number of those generic tasks, such as diagnosis, monitoring, assessment, repair, etc. The generic tasks provide

[2]When we use the term "system" we always mean the target system to be developed, unless explicitly stated otherwise. The simplifying assumption is made here that the system development process is aimed at one single application.

the starting point for the second stage: the detailed analysis of a problem solving task which is described in the model of expertise (and the model of cooperation).

Another question concerns the problem whether the analysis models describe the current or the future situation. The position taken in MSA is that one should concentrate on the future situation. Although this has been a point of discussion for many years in KADS (see Sec. 3.2) the current position is that the final conceptual model also is a model of the future situation. The position of OMT appears to be close to that of MSA as well.

8.3 Modelling Framework

One can take three basic perspectives when modelling a system [Yourdon, 1989a; p. 219] [Rumbaugh *et al.*, 1991; p. 17]:

Data perspective Modelling the essential information that one needs to represent in the system.

Functional perspective Modelling the functions and the flow of data between functions.

Control perspective Modelling the dynamic, time-dependent, behaviour of the system.

Yourdon remarks [Yourdon, 1989a; pp. 218-222][3] that many of the debates in software engineering have been about the "right" perspective. Traditionally, the information modellers argue that one must start with describing the data perspective because the data form the most stable part of the application. The data-flow adepts claim that data representation is so dependent on the way it is used that one should start with functional decomposition. People working on real-time systems claim that neither is the right approach for this type of system: a control-oriented modelling approach is required.

In Yourdon's view, these debates about the right perspective are fruitless in the sense that there does not exists a single perspective that is better than another for every application. The right perspective varies with the nature and the complexity of the application domain. In some domains, such as large database applications, the structure of the information is complex and the functions relatively simple. In real-time systems, the dynamics of the system are often the most complex part. Also, given the increasing complexity of systems being built, there now tend to be more and more applications in which all three perspective are (almost) equally important.

The OMT analysis model consists of three sub-models (object model, dynamic model and functional model) each representing one of the perspectives. The MSA analysis model does not contain an explicit representation of the three perspectives. The KADS model of expertise can be viewed as supporting an integrated description of the three perspectives (see also Ch. 6, Table 6.1), although this is not an articulate postulate of the description of the model components given in Ch. 3.

In the next section we compare the approaches with respect to the modelling techniques offered for modelling the three perspectives during analysis. In addition, we study the way in which the *connection* between perspectives is modelled.

[3] These remarks are made in a chapter of *Managing the structured techniques* and do not come from the book on MSA.

8.4 Modelling Techniques

8.4.1 Data perspective

OMT OMT offers an extensive (graphically-oriented) object-modelling language based on constructs developed in semantic database modelling. The main ingredients of this language are:

- *Object class & instance* The object is the central entity in the data model. An object class describes the structure of a set of object instances through the definition of attributes ("properties") and operations ("methods") for objects in this class.
- *Association & Link* A link is a connection between object instances. An association describes the structure of a set of links between object instances (one could think of it as a link class). For associations, attributes and cardinality constraints (in OMT "multiplicity") can be defined that apply to links in this class.
- *Aggregation* Aggregation provides a way of describing part-whole relations between objects, e.g. a mixer consists of a vessel and an agitator. It is in fact a special type of association with additional semantics.
- *Generalisation & Inheritance* Generalisation refers to a hierarchical organisation of object classes to capture similarities between objects. Inheritance refers to the fact that this hierarchy can be used to inherit object class definitions such as attributes and operations. OMT allows various types of generalisation/specialisation such as extension and restriction (similar to the notion of differentiation and value restriction in KL-ONE [Brachman & Schmolze, 1985]). It also supports multiple inheritance.
- *Module* A module provides a way of grouping object classes that naturally belong together (e.g. through various associations), without imposing the semantics of aggregation.
- *Constraints* Constraints are used to express features of one or more elements of the data model that cannot be represented with the constructs mentioned above. Simple constraints are annotated in the data model using semi-natural language (these strings could contain equations etc.). According to OMT, complex constraints should be placed in the functional model.

MSA MSA provides two modelling tools for describing the data perspective: the *data dictionary* and the *Entity-Relationship Diagram* (ERD).

The data dictionary consists of a set of structural descriptions of the basic data elements that are relevant in a particular application. For example, a data dictionary could contain an entry for *person-name* and a description of its internal structure (title + first name + optional middle name + last name, each conisting a some sequence of characters).

The ERD describes the general structure of entries in the data dictionary and their interrelationships. The ERD's used in MSA support a subset of the modelling constructs provided by OMT, notably entity classes with attributes (similar to object class definitions without operations), relationships (associations), and sub-supertype relations (generalisation & inheritance, but with less expressive power than provided by OMT).

KADS The KADS modelling tools for describing the data perspective were discussed extensively in Ch. 4. This data modelling language provides ER-type constructs with generalisation/specialisation and aggregation (part-of structures). In addition, KADS provides some constructs for modelling the structure of "rules" or 'axioms": relations between expressions about entities. An example of such a relation is a causal relation. A causal relation is not just a relation between two entities, but between *state values* of entities (e.g. "if the lumen of a coronary artery is obstructed, then this may lead to necrosis of the heart muscle"). A distinction is also made between relations between entity classes and entity instances.

Discussion From the data perspective there do not seem to be many differences between CSE and KE. This is also clear from the literature on for example "expert database systems" [Kerschberg, 1986]. There is a clear link between work on AI data modelling languages such as KL-ONE and its descendents, and research on extensions of the ER approach in semantic database modelling.

Still, the emphasis in KE is slightly different. An example of this can be found in the efforts in KADS to describe rule and axiom schemata. In OMT and MSA this type of information would either need to be described in the form of constraints (OMT), which are basically just textual annotations of the data model, or in the functional model. For example, Yourdon proposes *decision tables* as a possible technique for describing a process specification (the description of the internal process of a function, see below) [Yourdon, 1989b; p. 219]. Decision tables constitute in fact sets of rules (see the example decision table in Table 8.4.1). For a KBS application a schematic description in the data model of the structure of this type of knowledge is crucial. This explains also the need for additional vocabulary in the KADS data modelling language, such as relations between expressions. For example, the medication table can be modelled in the DDL (see Ch. 4) as a relation between expressions about certain properties of patients (age, sex and weight) and a medication.

	1	2	3	4	5	6	7	8
Age > 21	Y	Y	Y	Y	N	N	N	N
Sex	M	M	F	F	M	M	F	F
Weight > 150	Y	N	Y	N	Y	N	Y	N
Medication 1	X				X			X
Medication 2		X			X			
Medication 3			X			X		X
No medication				X			X	

TABLE 8.2: Example decision table (copied without permission from [Yourdon, 1989b]).

An important common theme is that of reusable data/knowledge bases. For example, many companies are developing company-wide data models, that should be used by each (new) application. In AI some efforts have started to develop large reusable knowledge bases. The best-known is the CYC project [Lenat & Guha, 1990]. Such large data or knowledge bases require at least an expressive data modeling language such as OMT and KADS offer. But this is not sufficient. The methodologies offer little guidance with respect to the ontologies of such knowledge bases. The definition of generally-shared ontologies in

currently an important research topic.

8.4.2 Functional perspective

MSA In MSA, the functional perspective is described using a "data-flow diagram" (DFD). A DFD consists of so-called ' 'bubbles", representing functions, and data stores. Functions and data stores are connected via directed links. These links represent data flows and describe the input/output of a function. The name of the data flow is written on the link. Fig. 8.1 shows an example DFD. In addition, a DFD can contain "control bubbles". These bubbles represent the link between the functional view and the control view (see Sec. 8.4.3).

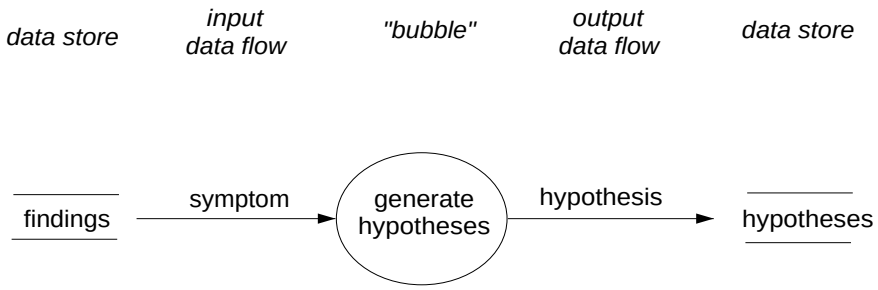

FIGURE 8.1: Example data-flow diagram

Each function itself can be described in more detail in a separate DFD, thus giving rise to a hierarchical DFD structure. A *process specification* describes the internal structure of a function that is not decomposed into sub-functions. MSA suggests three main techniques for writing process specifications:

1. *Structured English* This technique describes a procedure using a subset of English. The aim of Structured English is to balance the formal properties of (procedural) programming languages with the flexibility and readability of natural language. Structured English descriptions specify in an informal manner the algorithm that the function uses to compute its output from the input.
2. *Pre/post conditions* Pre- and post conditions describe logical relations that must hold between input data, output data and/or data stores. The main distinction with Structured English is that one does not describe the algorithm itself.
3. *Decision tables* A decision table specifies for each value of the input variables the value of the output variable. A decision table represents in KBS terms a rule set (see Table 8.4.1).

OMT OMT basically applies MSA techniques for describing the OMT functional model. The main difference lies in the fact that process specifications are described as operations (methods) on an object class in the data model. OMT advocates basically the same techniques for describing process specifications (i.e. operations) as MSA.

KADS The functional perspective in KADS can be found in the task model and in the model of expertise. The task model provides a top-level functional decomposition. The task knowledge in the model of expertise details this decomposition to describe problem-solving tasks (i.e. composite functions) and, at the lowest level of decomposition, knowledge sources and transfer tasks (i.e. primitive functions). DFD's are not systematically used in KADS to describe the functional decomposition. The input-output dependencies between the leaf functions are graphically described in an inference structure. An inference structure can be viewed as a data-dependency diagram at the lowest level of functional decomposition.

A first difference between the functional description in KADS and in CSE is that the data elements in inference structures do not refer directly to elements of the data model (entities, relations, etc.). Inferences in KADS define functional objects of which the name reflects the *role* that the object plays in the problem-solving process. The major reason for introducing separate functional objects in KADS is reusability. The functional objects give a characterisation of the data in terms of the roles that these data elements play in solving the application task. This creates the possibility of using functional descriptions such as inference structures as templates for a class of application tasks with similar characteristics. KADS calls these reusable templates "interpretation models". See for a more detailed discussion of interpretation models Ch. 3 and Ch. 5.

A second difference is that the KADS knowledge sources are not described via detailed process specifications. These leaf functions are described via the data-flows and the underlying domain knowledge used by the function. As pointed out in Sec. 6.4, the analyst takes an *automated deduction* view on the primitive functions: is it clear that it is possible to derive the input from the output plus the underlying domain knowledge? The specification of the actual algorithm for computing the output is left for the design phase. This approach is probably closest to the pre/post condition technique for writing process specifications. This technique only describes logical relations between data elements involved in the function.

A third distinction is that KADS provides a *typology* of basic functions (see Sec. 3.5.1). Each basic function should be an instance of one of these types. The current typology in KADS has however proved to be insufficient for handling every possible function.

Discussion: data-function interactions The interaction between the data perspective and the functional perspective is probably one of the most critical points in the system development process. Yourdon describes from his experience the frequently occurring situation where two groups, an "information modelling" group and a DFD group, start working in parallel on an application and end up with incompatible results. In KE this dependency problem between data and function has been called the *interaction hypothesis* [Chandrasekaran, 1988]: data cannot be described independent of its use.

In MSA the data flows and the data stores in the functional model refer directly to elements in the data model. The situation is similar for OMT. Direct links between the data model and the functional model hamper the reusability of functions (and data). The KADS approach to connecting the functional perspective and the data perspective is substantially different from the CSE approaches. In KADS functional role names are introduced in the functional model. The mapping of functional names onto elements of the data model is specified separately. The introduction of functional objects implies that

the functional description is not directly dependent on the data model. As remarked above, this opens the possibility of reusing functional descriptions such as inference structures. The tentative typology of primitive functions (knowledge sources) and the approach taken to define the internals of a primitive function (I/O plus static domain knowledge used) also facilitates a concise description of the functional perspective.

This type of reusability is not present in MSA. The simple case study of MSA in [Yourdon, 1989b; Appendix F] (the Yourdon Press application) contains 9 complex data-flow diagrams and 23 pages of process specifications in Structured English. It appears attractive to study whether KADS-type techniques can be used in CSE to reduce the effort spent on functional descriptions in individual applications.

OMT advocates reuse of functions by selecting (parts of hierarchical structures of) existing object-class definitions on which operations are defined. There is however no typology of primitive functions and no explicit relation between the type of task the system has to perform and the objects to be reused.

8.4.3 Control perspective

OMT & MSA Both OMT and MSA employ state-transition diagrams for describing the control (or "time-dependent behaviour") of the system. Sometimes, additional internal control is specified in process specifications, e.g. if these are described through structured English procedures.

OMT employs a comprehensive state-transition technique developed by Harel [Harel, 1987]. The MSA diagrams support a simpler version. The OMT state-transition diagrams consist of six types of elements: states, state-transition links, events, conditions, activities and actions. An (object) *state* is the set of attribute values and links held by an object or an abstraction of it. A *state-transition link* is a directed dependency between states: it indicates that one state can lead to another state. An *event* is something that happens at some point in time; *conditions* are valid over an interval of time. Events and conditions are used to indicate when a state change over a transition link takes place. For example, a state "no sound" of an audio system changes into the state "sound" when the event "play-button CD player pressed" occurs under the conditions that the power is on, a disc resides in the CD player etc. *Activities* and *actions* are functions (i.e. operations on an object). An activity is associated with a state: for example, in state S1, do activities A1 and A2 in sequence. An action is a function associated with a state transition. The OMT state-transition technique supports the specification of concurrency and the partitioning and levelling of diagrams.

In OMT the connection between the control perspective and the other perspectives is achieved through the activities and actions: these map onto operations of an object class in the data (object) model. MSA includes special "control bubbles" describing control relations in the data-flow diagram The process specification of a control process is provided by a state-transition diagram. The inclusion of control processes should ensure an adequate connection between functional and control perspective.

KADS The task knowledge in the KADS model of expertise uses a form of pseudo-code to describe the internal control of a problem-solving task. This control procedure (the

"task structure") defines control dependencies between functions (sub-tasks, inferences) involved in the task. Transfer tasks (functions that communicate with a user or another system, see Ch. 3 and [de Greef & Breuker, 1992]) are used to indicate events, for example those in which an external agent has the initiative (*receive* and *provide*, see Sec. 3.4.3). In some applications the structure diagrams proposed by JSD were used instead of pseudo code to describe task control [Readdie & Innes, 1987; de Greef *et al.*, 1987]. Additional control of the problem solving process can be specified in the strategic knowledge (to handle internal events such as impasses), but no particular formats are prescribed for these descriptions.

The model of cooperation describes aspects of the control of transfer tasks , in particular the *initiative* (who is responsible for starting communication).

The connection of control and function is achieved by describing control as a part of the task description.

Discussion In CSE the description of control is focused on the relation between the system and external agents (users, other systems). State-transition diagrams are especially useful if the system interacts heavily with its environment and external events (e.g. user actions, incoming data) strongly influence the activities to be performed by the system.

In contrast, the description of control in KADS focuses on the *internal* control of system behaviour. This is probably due to the particular characteristics of knowledge-based systems. Problem solving involves elaborate reasoning strategies which KADS tries to capture through the definition of task and strategic knowledge. In the model of expertise the emphasis lies on the internal control of reasoning. KADS assumes that it is possible (at least to some extent) to study the internal behaviour of the system (the model of expertise) and the interactions with the outside world (the model of cooperation) in parallel.

While the KADS approach may be sufficient for systems that work mainly in "batch-mode" (e.g. certain diagnostic or configuration systems), the vocabulary offered for describing control appears insufficient for real-time KBS applications such as process control systems. State-transition diagrams are more appropriate for these type of applications. Integrating state-transition diagrams with the existing KADS constructs can be achieved easily, e.g. by defining tasks as activities or actions or by allowing pseudo-code descriptions of the invocations of activities and/or actions. Given the growing complexity of conventional systems, these extensions can be useful for these systems as well.

8.5 The Role of the Analysis Model in Design

In this section some brief remarks are made about the role of the analysis model in the design process. As discussed in Ch. 6. KADS strongly advocates a "structure-preserving" approach to design: preserving the structure and the content of the information in the analysis model during design and implementation. It was argued that this approach to design facilitates code reusability, system maintenance, and explanation of the system's behaviour in a for humans intelligible way.

Similar ideas are put forward in the object-oriented approaches to analysis. Some quotes from OMT:

> "Optimization of the design should not be carried to excess, as ease of im-

> plementation, maintainability and extensibility are also important concerns." (p. 227)
>
> "Object-oriented design is primarily a process of refinement and adding detail." (p. 228)
>
> "Design decisions should be documented by extending the analysis model, by adding detail to the object, dynamic and functional models." (p. 253)

In OMT design is viewed as a process of adding detail to the three parts of the analysis model. For example, additional classes may be introduced for optimisation, but may not corrupt the original structure.

OMT is not the only object-oriented approach that takes this point of view on design. The object-oriented approach described by [Coad & Yourdon, 1991] views design along similar lines:

> "moving from OOA[4] to OOD[5] is a progressive expansion of the model. ...The expansion is in contrast with the radical movement from data flow diagrams to structure charts .... Such a movement is abrupt and forever disjoint: the designers get a hint from analysis and then go off to the "real" design. ...Moreover, meaningful traceability - one which supports the process itself – withers away." (p. 178)

In MSA, analysis and design are viewed in a more traditional way. The functions in the data-flow diagrams are mapped onto modules in "structure charts". These modules are reorganised on the basis of principles concerning coupling and cohesion of modules. No explicit attention is given in this transformation to the preservation of information.

8.6 Conclusions

The aim of this comparison was twofold: (i) to identify bridges between the two fields, and (ii) to identify common research areas in which results from one filed could be of use for the other field.

Bridges The common ground between CSE methods and KADS lies in the use of the three perspectives: data, function and control. Each of the methods studied constructs models along all three dimensions, albeit with a different emphasis. Fig. 8.2 summarises the various techniques that each method recommends for constructing models along each dimension. In OMT these perspectives are explicit sub-models of the modelling framework. In the KADS model of expertise, the three perspectives are integrated into one model. This similarity of perspectives between CSE and KADS is in our view a strong indication that KADS is not just an ideosyncratic method for constructing KBS's, but is based on the same foundations as conventional methods.

With respect to data modelling we have seen that KADS and CSE offer similar constructs, based on techniques from semantic database modelling. The main difference lies in the fact that in KADS (and in KE in general) more information is viewed as being part

[4]Object-oriented analysis
[5]Object-oriented design

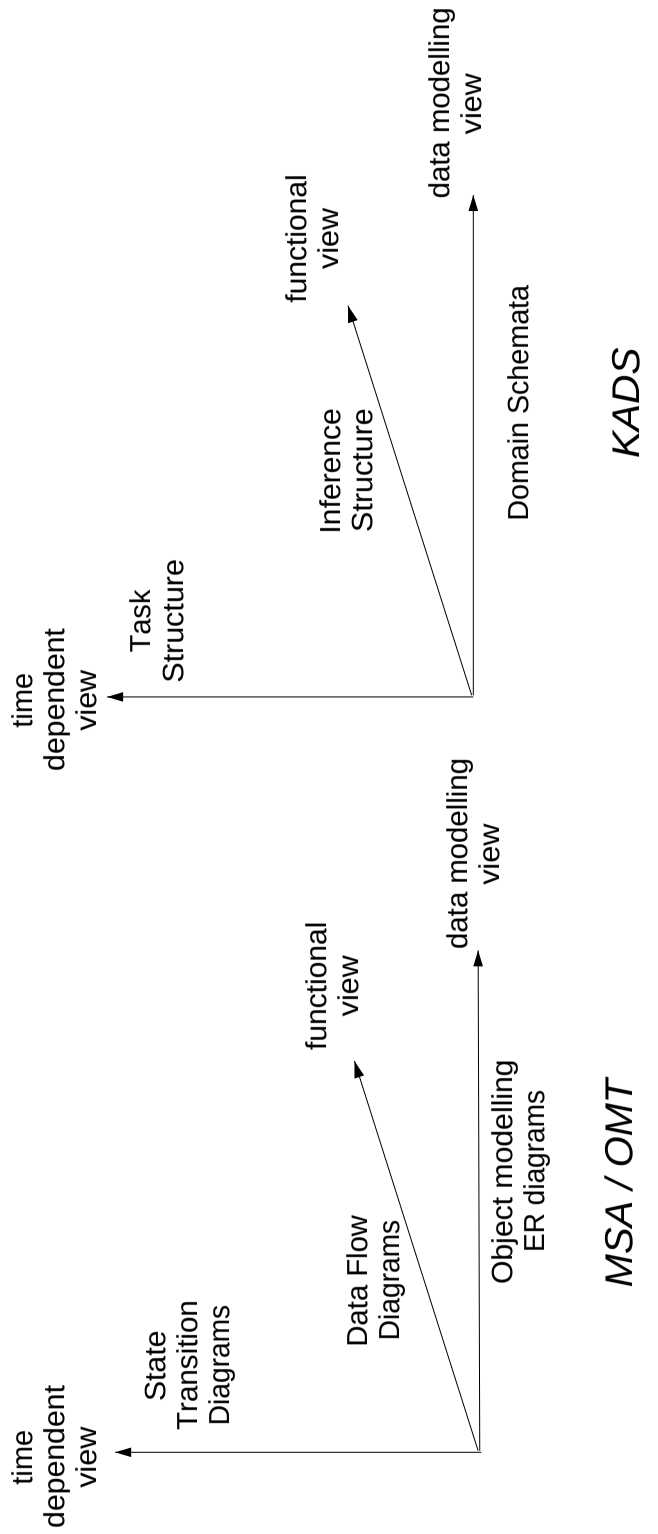

FIGURE 8.2: Synopsis of the main ingredients describing the three different perspectives in MSA/OMT and in KADS.

of the data perspective. For example, part of the information present in process specifications such as decision tables is included in the data model. This explains the need for additional vocabulary in the KADS data modelling language.

With respect to the functional perspective it can be concluded that the concept of tasks, inferences and inference structures is quite compatible with data-flow diagrams. The main difference is the extra layer of abstraction that KADS through the functional role names and the separate specification of data-function mappings.

With respect to the control perspective we have seen that there is a different emphasis in KE and CSE. In the first, the control description is focused on the internal control of reasoning; in the latter the control mainly describes the interaction between the system and the outside world.

Research topics The field of data modelling is clearly a common research area of KE and CSE. The development of data modelling languages which can be used for both types of systems would facilitate the integratrion of KBS and conventional applications. The study of reusable ontologies is a longer-term research goal. In principle, ontologies can provide powerful support for data modelling.

It appears worthwhile to study whether the KADS approach to functional modelling, in which one abstracts from the data model and tries to define types of functions, can also be used in CSE. This paves the way for reusing functional descriptions in a similar way as provided by interpretation models in KADS.

KADS should consider whether it is useful to include techniques such as state-transition diagrams to describe control, especially for real-time KBS applications.

Chapter 9

Differentiating Problem-Solving Methods

Problem solving methods (PSM's) are important in constructing modular and reusable knowledge-based systems, as they specify the different types of knowledge used in knowledge-based reasoning, as well as under what circumstances what knowledge is to be applied. We argue that there is a need for a more rigorous description of PSM's than the prevailing verbal and/or computational descriptions, because this facilitates clarifying, communicating and comparing problem-solving knowledge. In this chapter an attempt is made to describe the Cover-and-Differentiate method for diagnosis in a more formal way, and to compare this method to Heuristic Classification. We bring to light considerable differences with the heuristic classification method, although in the original literature Cover-and-Differentiate was said to be a specialised form of it. We are not claiming that our model is the only correct one. However, the account given in this chapter can be a starting point for a precise, knowledge-level, definition of what methods like Cover-and-Differentiate actually do.

This chapter is a revised version of a paper presented at EKAW'92. Reference: G. Schreiber, B. Wielinga, and H. Akkermans. Differentiating problem solving methods. In Th. Wetter, K-D. Althoff, J. Boose, B. Gaines, M. Linster, and F. Schmalhofer, editors, *Current Developments in Knowledge Acquisition - EKAW'92*, Berlin/Heidelberg, 1992. Springer Verlag.

9.1 Introduction

A generally accepted principle underlying Knowledge-Based Systems (KBS) is that they solve problems through the application of domain specific knowledge. On the basis of this principle many useful systems have been developed [McDermott, 1988], some of which are in operational use. However, the principle of *problem solving power through application of domain knowledge* does not specify what the *nature* of the knowledge is that these systems use and *under what circumstances what* knowledge should be applied, i.e. the *method* of solving a particular problem through application of knowledge still needs to be explicated. In recent work [McDermott, 1988; Clancey, 1983] several of such Problem Solving Methods (PSM's) have been described, but so far no comprehensive theory of PSM's has been put forward. The goal of this chapter is to investigate the nature of problem solving methods through an analysis of methods that were used in some well known AI programs. The application task is diagnostic reasoning. The example PSM this chapter focuses on is the Cover-and-Differentiate method [Eshelman *et al.*, 1988; Eshelman, 1988].

We also illustrate how several methods, which are seemingly alike at the level of informal description, can be compared and can be shown to be different, when a more rigorous PSM analysis method is applied.

What constitutes a PSM? It is progressively becoming clear [Clancey, 1985b; Wielinga & Breuker, 1986; Wielinga *et al.*, 1992a; Steels, 1990] that there are a number of basic ingredients that are needed in order to specify a problem solving method. These ingredients are *types* of knowledge which would be instantiated for each specific method. There are at least the following types of knowledge required in the specification of a PSM.

1. Knowledge describing which *inferences* are needed in an application. Inferences (in KADS: "knowledge sources") describe the elementary reasoning steps that one wants to make in some domain and the *roles* that pieces of domain knowledge that are manipulated by the inferences play in the overall reasoning process (e.g. *finding* or *hypothesis*; in KADS "meta classes"). The set of inferences is often graphically represented in a diagram showing the input-output dependencies between inferences: the so-called "inference structure".

2. Knowledge about the *structure of the domain-specific knowledge* required to perform inferences. For example, an inference in which quantitative *data* are abstracted into qualitative *findings* requires domain knowledge which relates pieces of domain knowledge that play the role of data and findings (e.g. definitions, generalisations or qualitative abstraction relations [Clancey, 1985b]). This type of knowledge corresponds to the notion of *domain view* in KADS [Wielinga *et al.*, 1992a].

3. *Control knowledge* which is used to determine how inferences are sequenced in a particular situation. The notion of a *task* is used to structure this control knowledge. A task defines a typical decomposition into inferences and/or sub-tasks together with internal sequencing information.

The different types of knowledge can be viewed as located in layers which have a object-meta-like relation. An inference *applies* domain knowledge with a particular structure; control knowledge *invokes* inferences.

Generally speaking, there are two ways in which PSM's are described in the literature: the *informal* description using either natural language or an informally defined graphical notation [Breuker *et al.*, 1987; Wielinga *et al.*, 1992a], and a *computational* description, which is formal and unambiguous, but difficult to interpret and dependent on implementation details. Both ways of describing PSM's make it hard to compare methods, let alone to develop a theory of problem solving in KBS. There is a clear need for an intermediate, formal but implementation-independent, description of PSM's.

In this chapter we show how formal methods can support the definition of the different knowledge types required for specification of PSM's. There are several reasons why a formal account of PSM's is useful. First, formal models are a means for concise and precise communication of PSM's. Second, formal models help to identify distinguishing properties of different PSM's and thus to compare them. A third reason is *re-usability*. When we specify different knowledge types in a modular way, modules can be re-used over different PSM's. Such re-usability is of great practical importance for building KBS's. Last but not least, formal approaches to modelling PSM's can provide first steps towards a theory of automated problem solving.

9.2 Framework for Specification of PSM's

In several recent papers [Akkermans *et al.*, 1992; van Harmelen *et al.*, 1990; van Harmelen & Balder, 1992], we have developed and applied a logic-based framework called ML^2, which is based upon the KADS approach and allows for a formal specification of PSM's. As pointed out in the Introduction, we propose to define a PSM in terms of different categories of knowledge, categories that are also distinguished in the KADS conceptual modelling framework. Our specification language ML^2 has been designed such that these various knowledge categories and types are expressed by means of different formal constructs. Basically, an ML^2 description is a structure of logical theories. The choice of both the logic and the structure has been derived from the nature of the various elements occurring in the KADS framework, as briefly indicated below.

Categories of knowledge Our description of PSM's will be in terms of different categories of knowledge: domain, inference, and task or control knowledge. This categorisation recurs in ML^2, each category being represented as a set of logical modules of a certain structure. Thus, in an ML^2 specification the various categories occur as separate components in a one-to-one relation to the 'layers' of a KADS conceptual model.

Domain knowledge Domain knowledge is modelled in ML^2 as a collection of logical theories. Each theory consists of a signature and a set of axioms. The logical language is usually first-order order-sorted predicate calculus (order-sorted FOPC), but it can be extended to include, for example, modal operators. Domain knowledge can be structured, since ML^2 allows for the modular combination of separate subtheories. Typically, such a subtheory or module corresponds to a specific domain model. This modular structuring and recombination is done by means of simple meta-theoretic operators, such as the *import* operator which generates the union of two theory modules.

Inference knowledge Also the knowledge at the inference layer is specified in terms of a modular order-sorted FOPC. The modular structure is such that the well-known inference structure diagrams in KADS and in ML^2 are identical. Inference steps (knowledge sources) and their inputs and outputs (metaclasses) are thus visible in ML^2 as separate components. Knowledge sources are specified as full-fledged logical theories, whereas metaclasses are mainly given by signature only (defining the language used at the inference layer).

Task knowledge Task knowledge contains control and procedural aspects which cannot be naturally modelled in terms of FOPC. Therefore we use for these aspects a different logical language, viz. quantified dynamic logic (QDL, a multi-modal logic) that is able to speak about the execution of inference steps and has built-in notions of sequence, iteration and selection. This QDL is a superset of the language used at the inference layer such that task decompositions, structures and procedures can be written down formally. Thus, it is possible to formally express procedural knowledge and have a declarative semantics. However, in the present chapter we will mainly deal with the domain and inference knowledge, and hardly touch upon the control aspects.

Links between categories In ML^2 the relation between the domain and inference knowledge is a meta-relation. This is a natural solution since the domain layer describes the content of expert knowledge, while the inference layer speaks *about* the inferential use of the domain knowledge. The link between these layers is specified in by a so-called *lift definition*. It gives domain model statements a name at the inference layer by establishing a naming relation. The mapping is achieved via sets of rewrite rules. Such a user-definable naming makes it possible to give *meaningful* names to domain knowledge elements that express the *role* that they play in the inference process. In addition to meaningful naming, ML^2 employs so-called reflection rules. For example, in the context of a specific inference step one can ask by means of a reflection rule whether a certain domain statement is present as an axiom or can be derived. In the following we will see several examples of these specification elements. For a further technical discussion the reader is referred to the above-cited papers on ML^2.

This specification framework will be used in the sequel for the description of PSM's, and in particular for our analysis of the Cover-and-Differentiate method.

9.3 A Model of Cover-and-Differentiate

In this section parts of a formal description in ML^2 of the problem-solving method *cover-and-differentiate* (C&D) are presented. The full formal description can be found in [van Harmelen *et al.*, 1990]. The information sources for this description [Eshelman *et al.*, 1988; Eshelman, 1988] do not supply a complete description of all the details of C&D. Parts of the underlying specification are thus a more or less "educated guess" about the workings of C&D. The idea behind this description is to create a platform to discuss what PSM's like C&D actually do and to be able to compare them.

Note All free variables in the theories given below are implicitly assumed to be universally quantified.

9.3.1 Conceptual description of C&D Cover-and-differentiate [Eshelman, 1988] is a problem solving method for diagnostic tasks. The main knowledge structure on which C&D operates is a causal network. The nodes in this network are expressions about the state of the system being diagnosed. Reasoning basically comprises two types of inferences: *cover* inference steps in which the causal network is used in an abductive manner to generate potential explanations for nodes that need to be explained, and *differentiate* inference steps in which these potential explanations are confirmed or disconfirmed by applying additional knowledge in the network. C&D uses in its reasoning two general principles: *exhaustivity* (every symptom should be explained), and *exclusivity* (a form of Occam's razor: all things being equal, parsimonious explanations are preferred). The solution that C&D comes up with is an explanation path from symptoms to (potentially multiple) causes. The solution can be a partial one.

One of the points that triggered the work presented in this chapter is the following quote from [Eshelman, 1988; p. 37]:

> "MOLE is an expert system shell that can be used in building systems that use a specialised form of heuristic classification to solve diagnostic problems."

In the comparison between Cover-and-Differentiate and Heuristic Classification in Sec. 9.4 we will come back to this statement and show that there are a number of fundamental differences between these two problem solving methods.

9.3.2 Structure of domain-specific knowledge in C&D

Concepts Two types of concepts are distinguished: *states* and *qualifiers*. States are the nodes in the causal network. Fig. 9.1 shows the example causal network defined in this section. The start-nodes in the causal network are the initial causes, the end-nodes are the complaints or symptoms. and the intermediate ones are internal states. Qualifiers are observations, that do not play the role of symptoms. These are used to qualify (or disqualify) the "truth" of a state or of a causal relation between states.

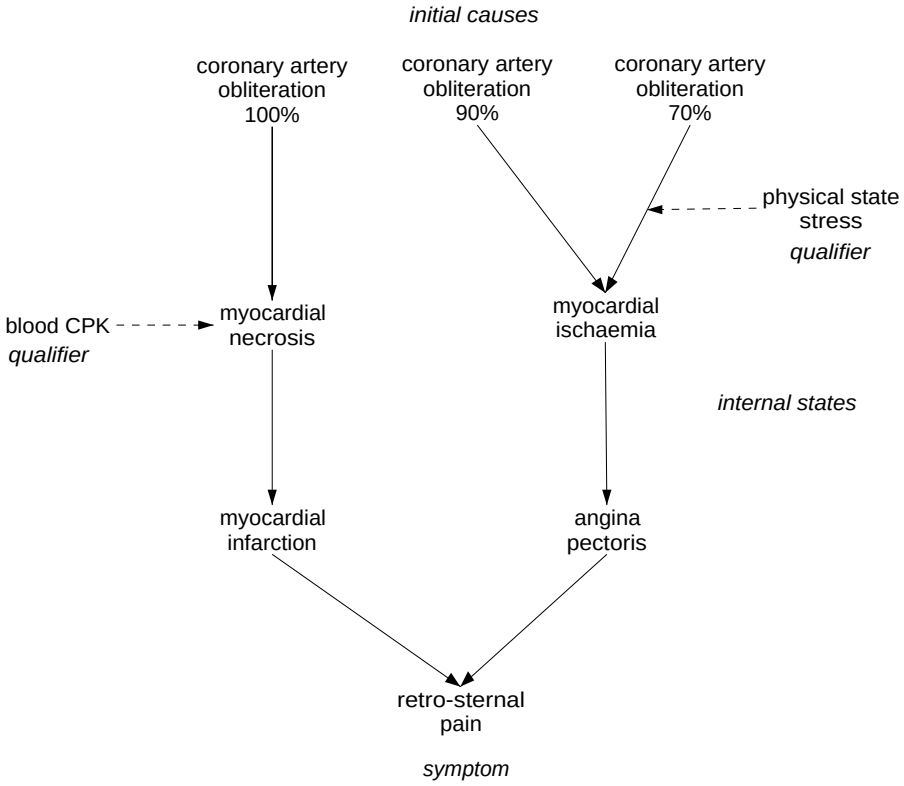

FIGURE 9.1: Example causal network for Cover-and-Differentiate

Relations In C&D, two types of relations are distinguished between states and/or qualifiers. These relations are represented in the domain theories as axioms.

Causal relations These relations define a causal network from causes to symptoms, potentially via intermediate states.

The theory *causes* shows some axioms representing causal relations in a domain of ischaemic heart diseases[1]. The axioms can be read as "some state (may) cause some other state". We use a modal operator $\Diamond$ ("it is possible that") to indicate that a state transition is possible, but not necessary.[2].

theory *causes*
  axioms
    $\Diamond$ (coronary-artery-obliteration(70-percent)
      $\rightarrow$ myocardial-ischaemia(present)) ;
    $\Diamond$ (coronary-artery-obliteration(90-percent)
      $\rightarrow$ myocardial-ischaemia(present)) ;
    $\Diamond$ (myocardial-ischaemia(present)
      $\rightarrow$ angina-pectoris(present)) ;
    $\Diamond$ (angina-pectoris(present)
      $\rightarrow$ retro-sternal-pain(present)) ;
    coronary-artery-obliteration(100-percent)
      $\rightarrow$ myocardial-necrosis(present) ;
    $\Diamond$ (myocardial-necrosis(present)
      $\rightarrow$ myocardial-infarction(present)) ;
    $\Diamond$ (myocardial-infarction(present)
      $\rightarrow$ retro-sternal-pain(present)) ;

Qualification relations States can be qualified through certain observations. The effect of such a qualifier can be positive (a state becomes more likely) or negative (a state becomes less likely). In a similar spirit, *causal relations* between states can be qualified through observations. The effect of such a qualifier can be positive (a causal relation becomes more likely) or negative (a causal relation becomes less likely).

theory *manifestations*
  axioms
    blood-CPK(high)
      $\rightarrow$ myocardial-necrosis(present) ;

    physical-state(stress)
      $\rightarrow$ $\Diamond$
        (coronary-artery-obliteration(70-percent)
        $\rightarrow$ myocardial-ischaemia(present)) ;

Lift definition The required domain structure for C&D is specified by providing *meaningful names* (see Sec. 9.2) for the domain specific axioms shown above. This is done in a so-called *lift definition*. The connection between axioms and their names is realised through a set of rewrite rules, that specify the relation between object-level (= domain-specific) and meta-level (= PSM-specific) knowledge structures.

In the lift definition *domain-schemata* below we show how the axioms of the theories presented above can be mapped onto names on a meta-level. The names are in this case

[1] To save space, we have left out the declaration of the signature (sorts, constants, functions, and predicates) and also of some import operations. For more details on these issues, see [Akkermans *et al.*, 1992]

[2] This use of modal logic presents no problems, as we do not deduce new theorems in this theory. See also Sec. 9.3.3

uninterpreted function terms such as *cover-relation(S1, S2)* and correspond to what we call a meaningful name. The first argument of the mapping function *lift* in the rewrite rules is the name of some object-level theory; the second argument specifies an axiom schema in this theory. The right-hand side of the rewrite rule maps instances of such a schema onto names in the meta-theory, such as a complex term of type *cover-relation*.

lift-definition *domain-schemata*
  from causes, manifestations
  to cover-theory, anticipate-theory, prefer-theory, ...
  signature
    sorts:
      % event has two sub-sorts
      (event (state qualifier))
    functions:
      cover-relation: state × state → ...
      anticipate-relation: state × state → ..
      prefer-state: qualifier × state → ..
      rule-out-state: qualifier × state → ..
      prefer-connection: qualifier × state × state → ..
      rule-out-connection: qualifier × state × state → ..
  lift-variables: P_1, P_2, P_3: atom
  mapping
      lift(causes, $\Diamond$ ($P_1 \rightarrow P_2$))
        $\mapsto$ cover-relation(⌈P_1⌉:state, ⌈P_2⌉:state) ;
      lift(causes, $P_1 \rightarrow P_2$)
        $\mapsto$ cover-relation(⌈P_1⌉:state, ⌈P_2⌉:state) ;
      lift(causes, $P_1 \rightarrow P_2$)
        $\mapsto$ anticipate-relation(⌈P_1⌉:state, ⌈P_2⌉:state) ;
      lift(manifestations, $P_1 \rightarrow P_2$)
        $\mapsto$ prefer-state(⌈P_1⌉:qualifier, ⌈P_2⌉:state) ;
      lift(manifestations, $P_1 \rightarrow \neg P_2$)
        $\mapsto$ rule-out-state(⌈P_1⌉:qualifier, ⌈P_2⌉:state) ;
      lift(manifestations, $P_1 \rightarrow (P_2 \rightarrow P_3)$)
        $\mapsto$ prefer-connection(⌈P_1⌉:qualifier, ⌈P_2⌉:state, ⌈P_2⌉:state) ;
      lift(manifestations, $P_1 \rightarrow \neg (P_2 \rightarrow P_3)$)
        $\mapsto$ rule-out-connection(⌈P_1⌉:qualifier, ⌈P_2⌉:state, ⌈P_2⌉:state) ;

The approach of separating the two views of knowledge structures (domain-specific and PSM-specific) has important advantages. Domain-specific theories could be re-used in other PSM's. Multiple mappings can facilitate multiple use of essentially the same knowledge. For example, the non-modal implications in the *causes* theory are mapped onto two different names: $P_1 \rightarrow P_2$ maps to both a *cover-relation* symbol and a *anticipate-relation* symbol (see the second and third lift rule). In C&D , this separation also keeps intact two distinct views on nodes in the causal network, namely the node as an expression about a value of an attribute of the system (at the object level) and the node as an object in its own right (at the meta level).

9.3.3 Inference knowledge in C&D Cover-and-differentiate operates on a causal network of states. This network is actually present in two forms:

1. The *causal network* itself as defined by the *cover* relations. These relations describe possible explanations.

2. The *explanation network* that is built during problem solving. The explanation network is a subset of the causal network and can be viewed as its instantiation for a particular problem.

The explanation network consists of three subsets, namely considered explanations, accepted (= preferred) explanations and rejected explanations.

The inference theories operate on the following data elements:

1. Element of the three subsets of the explanation network: (i) considered explanations, (ii) accepted explanations, or (iii) rejected explanations.
2. A focus: a state in the considered or accepted explanation network that is not explained by a another state.
3. A finding: some observed state or qualifier.

These data elements correspond to what were called *roles* earlier.

theory *role-defs*
  use domain-schemata
  signature
    predicates
      considered-explanation: state × state
      accepted-explanation: state × state
      rejected-explanation: state × state
      focus: state
      finding: state ∨ qualifier

The phrase "explanation network" as used in the rest of this text refers to the considered solutions. Note that inference theories specify elementary inference steps. Updates of the sets of considered, accepted and rejected explanations are handled in the control knowledge. Through the **use** clause one declares that the theory needs access to the object-level terms provided by the lift definition *domain-schemata*.

The elementary inference steps are described as a set of first-order theories, that use the domain schemata described in Sec. 9.3.2. Fig. 9.2 depicts the inference steps (the ovals) we have specified for C&D and their input-output (the boxes).[3] The inferences can be divided into two groups:

1. Inferences that use the domain knowledge defined by the lift definition *domain-schemata*. Examples of these inferences are *cover*, *anticipate*, *prefer* and *rule-out*.

2. Inferences that reason only about the current state of working memory, e.g. the current contents of the explanation network. Examples of such inferences are the theories describing the principles of *exhaustivity* and *exclusivity*.

Cover Inference The *cover* inference generates considered explanations. It uses the *cover-relation* to find a potential explanation of a state that is not yet explained (the focus). The cover inference step builds the explanation network by going backwards through the causal network. *Cover* models one aspect of the *exhaustivity* principle of C&D: all symptoms should be explained, whenever possible.

[3] Note that this diagram does not prescribe a particular order in which the actual problem-solving should be carried out. This is specified as separate control knowledge.

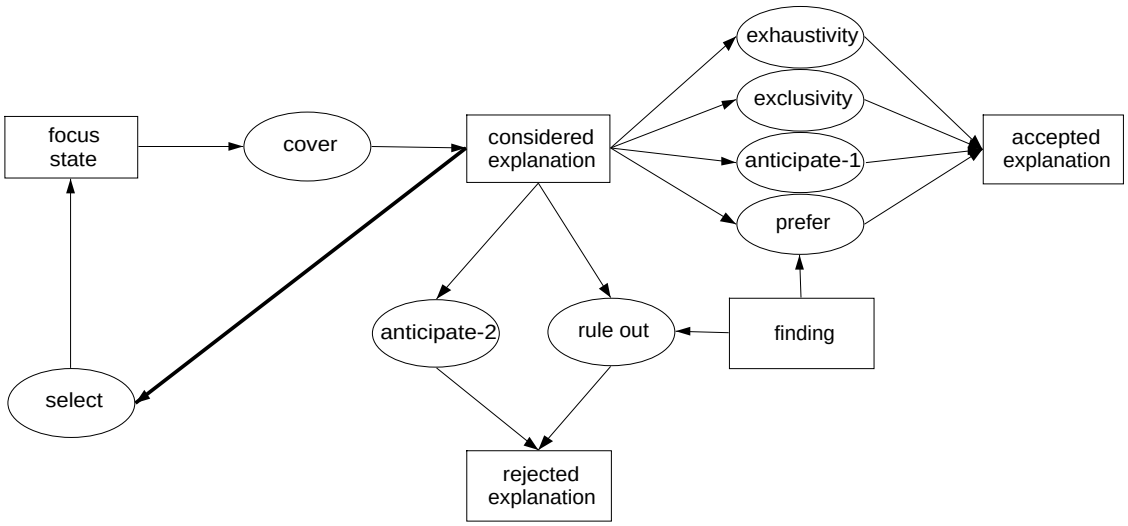

FIGURE 9.2: Inferences in Cover-and-Differentiate: their input-output dependencies.

theory *cover*
  use domain-schemata
  import role-defs
  axioms
    focus(S_1) $\wedge$ $ask_{\in}$(causes, cover-relation(S_2, S_1))
      $\rightarrow$ considered-explanation(S_2, S_1)

$Ask_{\in}$ is one of the reflective predicates. It requests the lift definition *domain-schemata* to find out whether the complex *cover-relation* term can be mapped onto an axiom of the "potential-causes" theory. Note that the predicate *considered-explanation* is an example of a description of the *role* that an object (or in this case, a tuple of objects) plays in the inference process. These role predicates are defined in the *role-defs* theory specified earlier. This theory is *imported* into the inference theory.

The structure of the defining axiom of inferences that make use of domain knowledge is typically:

$$< inputs > \wedge\ ask_{\in}(domain\ knowledge) \rightarrow < output >$$

where the inputs and outputs are role-names of objects in the reasoning process.

Anticipate Inference The differentiation part of C&D is more complicated than covering and consists of a number of elementary inferences (see also below). The *anticipate* inference is part of this differentiation process, in which the considered explanations generated by *cover* are pruned. The *anticipate* theory defines that if a state S1, that is considered as an explanation for a state S2, should always cause some other state S3, then S3 should be true. If this is the case, then S1 should be accepted as an explanation of S2 (and S3), else it should be rejected.

As inference in KADS have only one output type (cf. Ch. 5), the anticipate inference is defined in two separate theories. *Anticipate-1* produces accepted explanations; *anticipate-2* produces rejected explanations.

theory *anticipate-1*
  use domain-schemata
  import role-defs
  axioms
    considered-explanation(S_1, S_2)
    $\wedge$ $ask_{\in}$(causes, anticipate-relation(S_1, S_3))
    $\wedge$ finding(S_3)
      $\rightarrow$ accepted-explanation(S_1, S_2) $\wedge$ accepted-explanation(S_1, S_3)

theory *anticipate-2*
  use domain-schemata
  import role-defs
  axioms
    considered-explanation(S_1, S_2)
    $\wedge$ $ask_{\in}$(causes, anticipate-relation(S_1, S_3))
    $\wedge$ $\neg$ finding(S_3)
      $\rightarrow$ rejected-explanation(S_1, S_2) $\wedge$ rejected-explanation(S_1, S_3)

We do not define *finding* here. It is assumed to find out whether a state is part of the explanation network or to query the user for a value, whatever is appropriate.

Prefer & Rule-out Inferences *Prefer* and *rule-out* are also part of the differentiation step of C&D.

The *prefer* theory uses two prefer relations (*prefer-state* and *prefer-connection*) to prefer a particular state as the explanation of a state over other states that are not explicitly preferred. The preference is established by the presence of qualifiers for this state or causal relation. E.g., in the example causal network of Fig. 9.1 the finding that a patient is physically stressed would give rise to a preference for the state with a smaller degree of coronary artery obliteration as the explanation of myocardial ischaemia (i.e. oxygen shortage in the heart muscle).

theory *prefer*
  use domain-schemata
  import role-defs
  axioms
    considered-explanation(S_1, S_2)
    $\wedge$ $ask_{\in}$(manifestations, prefer-state(S_1, Q))
    $\wedge$ finding(Q) $\rightarrow$
      accepted-explanation(S_1, S_2)) ;

    considered-explanation(S_1, S_2)
    $\wedge$ $ask_{\in}$(manifestations, prefer-connection(S_1, S_2, Q))
    $\wedge$ finding(Q) $\rightarrow$
      accepted-explanation(S_1, S_2) ;

The structure of the *rule-out* theory is similar to the *prefer* theory. This inference produces rejected instead of accepted explanations using another partition of the domain knowledge (*rule-out-state* and *rule-out-connection*).

theory *rule-out*
  use domain-schemata
  import role-defs

axioms
considered-explanation(S_1, S_2)
$\wedge$ $ask_{\in}$(manifestations, rule-out-state(S_1, Q))
$\wedge$ finding(Q) $\rightarrow$
rejected-explanation(S_1, S_2)) ;

considered-explanation(S_1, S_2)
$\wedge$ $ask_{\in}$(manifestations, rule-out-connection(S_1, S_2, Q))
$\wedge$ finding(Q) $\rightarrow$
rejected-explanation(S_1, S_2) ;

Exhaustivity C&D assumes that every state that can be explained, must be explained. This is called the *exhaustivity principle*. This principle can be used to accept an explanation by ruling out the candidate explanations.

The theory below specifies this use of the exhaustivity principle. The axiom states that a potential explanation of a state can be accepted, if there are no other potential explanations of the state. This is the case if all other explanations were ruled out or if the explaining state was the only explaining state in the causal network.

theory *exhaustivity*
import role-defs
axioms
considered-explanation(S_1, S_2) AND
$\neg$ $\exists$ S_3 considered-explanation(S_3, S_2)
$\rightarrow$
accepted-explanation(S_1, S_2)

N.B. We assume that the task knowledge specification ensures that explanations that are rejected are no longer member of the set of considered explanations.

Exclusivity *Exclusivity* models the exclusivity principle of cover-and-differentiate. Exclusivity is a form of Occam's razor: all things being equal, parsimonious explanations are preferred.

The axiom below says that if a state S_1 explains a state S_3 and also some other state S_4, then this explanation should be preferred above a competing explanation S2 for S_3 where the explaining state explains only S_2.

theory *exclusivity*
import role-defs
axioms
considered-explanation(S_1, S_3) $\wedge$ considered-explanation(S_2, S_3)
$\wedge$ ($\exists$ S_4 considered-explanation(S_1, S_4) $\wedge$ $S_3 \neq S_4$)
$\wedge$ $\neg$ ($\exists$ S_5 considered-explanation(S_2, S_5) $\wedge$ $S_3 \neq S_5$)
$\rightarrow$ accepted-explanation(S_1, S_3)

An interesting feature of these last two theories is that, unlike the other theories, these do not make use of domain knowledge (i .e. there is no *ask* statement). This is fully in accordance with the generality of the principles. To re-use the theories in another PSM it would be sufficient to rename the predicate-symbols.

Establish focus The *establish-focus* inference searches for states which are considered (or accepted) as an explanation for another state, but need themselves to be explained as well.

theory *establish-focus*
  import role-defs
  axioms
    ($\exists$ S_2 considered-explanation(S_1, S_2))
      AND
    ($\neg$ $\exists$ S_3 considered-explanation(S_3, S_1))
      $\rightarrow$ focus-state(S_1)

This last theory also does not apply domain-specific knowledge.

9.4 Analysing Cover-and-Differentiate

Given a formal account of cover-and-differentiate as a problem solving method for a diagnosis task, we are now in a position to use the formalisation for analysing the relation of C&D to other methods for diagnosis. Eshelman [Eshelman, 1988] states that C&D is a form of heuristic classification (HC) [Clancey, 1985b]. A formal description of parts of HC is presented in [Akkermans *et al.*, 1992]. When we compare C&D and HC there appear to be a number of fundamental differences.

1. A crucial elementary inference step in HC is the *abstraction* inference: the left part of Clancey's "horseshoe" [Clancey, 1985b] (see Fig. 5.16). This abstraction step in the HC problem solving method is used to abstract specific findings (e.g. patient is alcoholic) to more general ones such as "compromised host". These general findings are then used in an *association* step to generate hypotheses. It is clear from the formal definition of C&D that there is no equivalent of such abstraction steps in the C&D method. Findings are either symptoms or qualifiers and are directly associated with hypotheses (states that explain other states). Of course abstraction could be added to C&D, but this would require an additional domain theory describing the relations to be used in the abstraction inferences. In addition it would require the definition of an *abstract* problem solving step, changes to the cover-theory would be needed and a new role would have to be defined: *abstracted-data*. Although these changes are not very difficult to make in the formal model, they would yield a different structure of the knowledge at several levels.

2. A second difference concerns the way in which hypotheses (*considered-explanations*) are generated. In the *cover-theory* these hypotheses are generated through a query of the potential-causes theory concerning cover-relations. This means that only those hypotheses are generated which are directly linked to the symptom being focussed on. In HC hypotheses can be generated from an etiological hierarchy through trigger relations. A trigger relation can relate one or more symptoms to a hypothesis anywhere in the hierarchy. So, the method for generating hypotheses in HC is more flexible and more of a heuristic nature than the one in C&D. Changing the C&D model to incorporate such heuristic associations would require significant changes. In order to maintain the exhaustivity principle (all symptoms are explained) a new

inference would be needed. This inference would establish an explanation path between hypotheses somewhere in the causal network and the symptoms. Moreover the simple control structure of C&D would need to be replaced by a more complex one, since the set of covered symptoms would have to be derived.

3. A third difference between C&D and HC concerns the way in which the differential is reduced. In C&D the set of hypotheses is reduced by applying rule-out anticipation and preference inferences. Each of these inferences applies to a *single hypothesis*. The HC method differentiates *between* competing hypotheses by searching for discriminating evidence. For example, the equivalent in HC of the *anticipate* inference would look like the theory below.

theory *hc-anticipate*
axioms
considered-explanation(S1, S3) $\wedge$ considered-explanation(S2, S3) AND
$ask_{\in}$(causes, anticipate-relation(S1, S4)) AND
$ask_{\in}$(causes, anticipate-relation(S2, $\neg$ S4))
$\rightarrow$
(finding(S4) $\wedge$ rejected-explanation(S2, S3)) OR
($\neg$ finding(S4) $\wedge$ rejected-explanation(S1, S3))

The premise of the theory mentions two *considered-explanation* atoms. The reason why in C&D differentiation can be performed on single hypotheses is that the exhaustivity principle allows C&D to prefer hypotheses by ruling out alternatives. There are thus essential differences between what is called *differentiation* in C&D and HC.

4. In HC hypotheses (internal states) are structured in a hierarchy which is used to generalise or specialise a hypothesis. No such hierarchical relations are present in the C&D domain theories, nor are they used in the inferences. Again such knowledge could be incorporated in a new domain theory and inferences and tasks could be updated accordingly.

In fact, there are many more differences. The solution in HC contains in principle one cause; in C&D the solution can consist of multiple causes and includes the causal pathways to these causes.

If we step back and take a global view on both C&D and HC, we observe some similarities. Both PSM's are specialisations of a general *generate and test* schema. In C&D the generate process is simply represented by the cover-theory. In HC this process is represented by a more complex combination of abstraction and heuristic association. The test process in C&D is realised by the differentiate task, which in turn applies the rule-out, prefer and anticipate inferences. In HC the test is performed through a different *differentiate* task using the hierarchy of hypotheses. Concerning the principles that underlie HC and C&D we see that both PSM's are based on abductive generation of hypotheses, but that C&D requires the symptoms to be fully explained by the solution, and that HC only requires that a solution is consistent with the symptoms.

All in all, we can conclude that C&D and HC have some similarities when viewed at a sufficiently high level of abstraction, but that the differences at a more detailed level turn

out to be considerable. The given formal account has thus shown that is not warranted to view C&D as a special form of HC. In addition we can see how new PSM's can be constructed by combining the ingredients of both models.

9.5 Conclusions

We have sketched in this chapter a framework for the analysis of problem solving methods. This has been illustrated by constructing a specification of the Cover-and-Differentiate method that can be used for certain types of diagnosis tasks. On this basis, we have brought to light considerable differences with the heuristic classification method, although in the original literature Cover-and-Differentiate was claimed to be a variant of it. Different specifications of the Cover-and-Differentiate method are conceivable. We are not claiming that our model is the only correct one, nor that it fully reflects the actual implementation in MOLE. However, the account given in this chapter can be a starting point for a precise definition of what Cover-and-Differentiate is.

The main conclusion is that analysis and specification of problem solving methods of a type as exercised in this chapter, is a useful means for clarifying and communicating the precise nature of problem-solving knowledge that underlies reasoning in knowledge-based systems. Informal statements in the literature about methods such as Cover-and-Differentiate appear to be imprecise, and sometimes misguided, if not incorrect. Thus, we need specification methods for PSM's that are better than saying that the computer program is the ultimate specification. This chapter has suggested some ways how this may be achieved.

A long-term goal behind the present work is the idea *to combine components of various problem-solving methods into new* PSM*'s* that are tailored to the domain application. For example, one could add to a Cover-and-Differentiate PSM an abstraction inference (an element of Heuristic Classification not present in Cover-and-Differentiate) if it appears that the domain data space is too large. This is a constructive use of PSM's and their components which however requires specifications of PSM's that are well-structured into components and clearly define the nature of these components, as well as the assumptions under which they may be used. Ultimately, this should result in knowledge-engineering libraries of reusable and combinable PSM components. Although mainly used for analysis purposes here, we believe that the methods discussed in this chapter provide a useful step in this direction.

Chapter 10

Conclusions

In this thesis we studied the application of Newell's idea of a "knowledge level" in the process of constructing knowledge-based systems. In our view a knowledge-level description is an essential ingredient in any principled approach to building intelligent systems. This is surely not to say that the approach outlined here is the only one. It should just be considered as a attempt to put the the knowledge-level idea to work in a consistent manner to the level of detail necessary for practical usage in knowledge engineering.

10.1 Explication of Assumptions behind KADS

In Ch. 2 and Ch. 3 assumptions behind KADS were explicated. A central assumption is that the KADS models of expertise can be seen as an attempt to reify the knowledge-level hypothesis for practical use in knowledge engineering.

A second assumption of KADS is that a knowledge-level description is *necessarily* underspecified for the purpose of constructing a symbol-level system that implements this description. Just like a physical law is no recipe for building a numerical simulation program, a knowledge-level model is no blue print for the artefact. In each case, a number of detailed "symbol-level" decisions still have to be made. It is possible to construct for some subset of knowledge-level descriptions automated transformation procedures which predefine symbol-level decisions and translate the descriptions into a working system, but this does not corrupt the premise that a knowledge level model (the KADS model of expertise) and its symbol-level realisation are fundamentally different levels of descriptions.

A third assumption of KADS is that a knowledge-level typology of the elements necessary for the required problem-solving behaviour provides important safe-guards against computationally intractable systems. This is fully in line with McCarthy's point that epistemological adequacy is directly related to computational adequacy. Finer grained epistemological distinctions prevent unrestricted application of domain knowledge and thus lead to more efficient and tractable systems.

An assumption is also that the knowledge-level provides the possibility of both explaining *and* predicting the behaviour of the resulting system. These predications are of a particular type, namely at the level of reasoning steps (which type, under what conditions) that we can expect an intelligent to perform.

10.2 Principles and Techniques Developed

With respect to the nature and role of KADS model of expertise we studied a number of more detailed methodological topics, notably (i) the specification of domain knowledge, (ii) the construction of inference structures, and (iii) the design process: mapping a knowledge-level description onto a symbol-level representation.

Domain-knowledge specification Data modelling methods in conventional software engineering and those used in knowledge engineering are converging. The DDL developed in this thesis already is a synthesis between conventional data modelling and AI representation languages. The DDL is based on an analysis of KBS-specific requirements for data modelling. It provides a generalisation over various symbolic representation languages. This is exactly what one would want from a knowledge-level description of domain knowledge, because it provides the key to reusability. What is still needed is a more formal theory of the constructs in languages such as this DDL, comparable to Chen's theory of ER models and its extensions.

Construction of inference structures Any methodology for KBS development needs to provide structured techniques for model construction in order to turn this process from an art (which it currently often basically is) into structured engineering. Ch. 5 presents a structured analysis of the process of constructing inference structures: a crucial ingredient of KADS models of expertise. The notion of top-down construction of inference models and the identification of generic components that can support this approach is a promising approach for this knowledge engineering activity.

The methodology underlying model construction needs to be further developed. The operations, methods and criteria discussed need to be worked out in more detail. For example, it is worthwhile to study various sub-types of the knowledge-differentiation method in more depth and try to link these to the type of operations that need to be performed on the inference structure. Also, criteria used in this process are an important subject for further research. Work on this last topic is being done in the SKBS-A2 project [Benjamins *et al.*, 1992a] and in the KADS-II project.

Operationalisation: from knowledge level to symbol level Operationalisation of a knowledge-level model should be guided by the "structure-preserving principle": both the *content* and the *structure* of the information in the model should be preserved in the final artefact. The skeletal architecture which follows from this principle and from the structure of the KADS model gives strong guidance for the design and implementation process and can be the basis for powerful support tools.

Currently, there are a number of research programs that attempt to bridge the gap between informal knowledge-level models and operational systems, namely (see also Fig. 10.1):

- *Language approach* Informal knowledge-level descriptions are mapped onto dedicated operational languages (OMOS, Model-K, etc.), possibly via intermediate formal knowledge-level descriptions. The mapping is a partial one, because the operational languages limit the expressivity due to executability requirements.

- *Reusable-architecture approach* The structure of the knowledge-level model is fixed to a large extent. This skeletal model is operationalised through a predefined mapping onto a reusable, task-specific architecture. This architecture can be instantiated by the knowledge engineer and/or the expert to derive the actual application.

- *Configuration of reusable components and methods* The knowledge-level model is configured from a library of reusable, generic model components (tasks, inference components, domain schemata). This model is mapped onto operational constructs in an environment that is based on a structure-preserving skeletal architecture and which contains a library of reusable pieces of code for implementing particular model components.

This last approach is the most promising one because it combines the "best of both worlds": it facilitates flexibility, reusability as well as executability. The systematic-diagnosis system (see Ch. 6) is a first step in this direction.

10.3 Applicability

In a methodological study such as this one it is difficult to provide sufficient empirical evidence for the claims being made. We have made an effort to use as much as possible real-life examples to illustrate applications of the ideas, principles and techniques presented.

The applications in the audio domain illustrate the general principles underlying KADS, the proposed data modelling language, the refinement of inference structures and the realisation of a dedicated support environment based on the structure-preserving principle.

The Sisyphus application illustrates the use of the DDL. It provides a detailed example of a principled process of model construction. It also shows clearly how the structure-preserving principle can be applied to build a working system that meets the requirements set out in Ch. 6. The resulting system is a clear example of an instantiation of the skeletal architecture defined in Sec. 6.4.

The model of heuristic classification, as realised in the NEOMYCIN, was used in many places as a reference point: particularly in the description of model construction, but also in the example operationalisation of the "abstract & specify" example.

The Fraudwatch application [Killin, 1992; Porter, 1992] illustrates the advantages of the combination of knowledge-level modelling and structure-preserving design with respect to the maintenance and refinement of the operational system.

The StatCons application [de Greef *et al.*, 1987], although not described in detail in this thesis, has served as a source of inspiration for many ideas presented.

10.4 Evidence for Newell's Claim

In the introduction it was pointed out that Newell formulated his knowledge-level hypothesis in response to confusion in the knowledge-representation community. The knowledge level was his proposal for a platform on which one could discuss and compare the merits of the various languages and applications.

We view this thesis as an attempt to put Newell's hypothesis to work in the (more restricted) domain of knowledge engineering. It makes clear that the knowledge-level

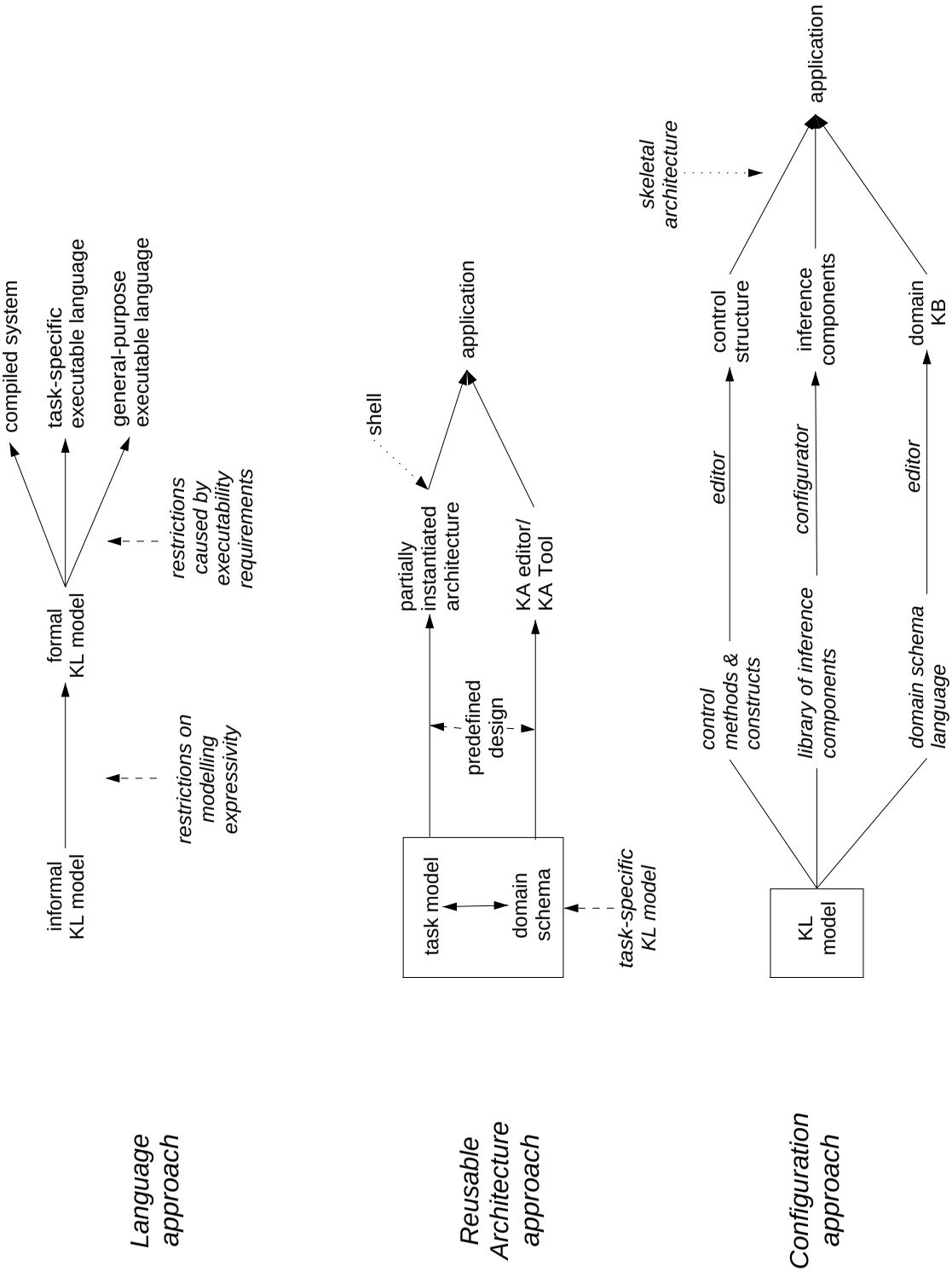

FIGURE 10.1: Schematic overview of characteristics of the three approaches to supporting the operationalisation of knowledge-level (KL) models.

view point is indeed useful in this context. Up till now, most KE theories were either described informally or through example pieces of code with their inherent computational biases. In our opinion, this study shows that the knowledge-level is indeed a medium for discussing and comparing problem solving methods employed by knowledge-based systems (cf. Ch. 9). It also provides the necessary anchor points for describing a methodological approach to knowledge engineering.

10.5 Perspectives for Knowledge Engineering

It is necessary and essential for the field to start working on comparing and unifying the various approaches. Some initial efforts in this direction have been undertaken. At the 1990 European Knowledge Acquisition Workshop the Sisyphus project was initiated, aiming (amongst other things) at a comparison of approaches to modelling problem solving by applying these to a standard data-set. Similar activities are also planned for the 1992 Banff Knowledge Acquisition Workshop. In the KADS-II project the University of Amsterdam and the Free University of Brussels are investigating a unification of KADS as presented in this thesis and the Components of Expertise framework developed in Brussels, while incorporating also ideas from other approaches.

In the short term, cross-validation studies could help in defining a common vocabulary for describing models of problem solving for knowledge-based systems. This could pave the way for an exchange of models between research groups and a standardisation of ingredients of such models. In the longer term, we envisage a research effort aimed at defining a comprehensive set of commonly shared problem solving methods and generic domain structures. This is an important (if not the only) route towards improving the state of the art in KBS development.

In our view, the knowledge-level is the appropriate level for *theory development* about knowledge-based systems in general and thus should be a major focus of attention in KE research. Such theories can be of various nature:

- The definition of the problem solving strategy for carrying out a task (the "problem solving method", see e.g. Ch. 9)
- The definition of the relation between input and output of a problem-solving task (as for diagnosis, e.g. [Console & Torasso, 1990; Konolige, 1992]).
- Task-independent theories about problem solving (e.g. SOAR [Laird *et al.*, 1987]).

Knowledge-level theories also provide a starting point for theory unification. Some work on this last point has been carried out in the REFLECT project [van Harmelen *et al.*, 1992].

In summary, knowledge-level modelling in combination with structure-preserving operationalisation offers both a principled approach to the engineering of knowledge-based systems, as well as a theoretical platform for theory formation about knowledge engineering.

Appendix A

DDL Definition of the Sisyphus Domain Knowledge

concept *employee;*
  properties:
    smoker: [true, false];
    hacker: [true, false];

relation *works-with;*
  argument-1: instance(employee);
  argument-2: instance(employee);
  semantics: symmetric;

relation *smoker-and-non-smoker*
  argument-1: instance(employee);
  argument-2: instance(employee);
  semantics: symmetric;
  axioms:
    ∀ E1,E2:employee
      smoker-and-non-smoker(E1,E2)
        ↔
      smoker(E1) = true AND
      smoker(E2) = false;

relation *on-different-projects*
  argument-1: instance(employee);
  argument-2: instance(employee);
  semantics: symmetric;
  axioms:
    ∀ E1,E2:employee, P1,P2:project
      on-different-projects(E1,E2)
        ↔
      works-on(E1, P1) AND
      works-on(E1, P1)
      ∧ P1 ≠ P2;

relation *hacker-and-non-hacker*
  argument-1: instance(employee);
  argument-2: instance(employee);
  semantics: symmetric;
  axioms:
    ∀ E1,E2:employee
      hacker-and-non-hacker(E1,E2)
        ↔
      hacker(E1) = true AND
      hacker(E2) = false;

concept *project;*
  properties:
    size: [small, medium, large]

relation *works-on;*
  argument-1: instance(employee);
  argument-2: instance(project);

relation *head-of;*
  argument-1: instance(employee);
  argument-2: instance(project);
    cardinality: min 0 max 1;

concept *department-role;*
  properties:
    occupancy: single, shared ;

concept *head-of-group;*
  sub-type-of: department-role;
  axioms:
    occupancy(head-of-group) = single;

concept *head-of-project;*
  sub-type-of: department-role;
    occupancy(head-of-project) = single;

concept *researcher;*
  sub-type-of: department-role;
  axioms:
    occupancy(researcher) = shared ;

concept *manager;*
  sub-type-of: department-role;
    occupancy(manager) = single;

concept *secretary;*
  sub-type-of: department-role;
  axioms:
      occupancy(secretary) = shared ;

relation *role-interaction;*
  argument-1: department-role;
  argument-2: department-role;
  properties:
      level: universal;
  semantics: symmetric;
  axioms:
      Level of interaction with
      head of group in descending order:
      secretary, manager, head of project

relation *boss-of;*
  argument-1: department-role;
  argiment-2: department-role;
  semantics: transitive;
  tuples:
      < head-of-group, manager >
      < head-of-group, head-of-project >
      < manager, secretary >
      < head-of-project, researcher >;

relation *employee-role;*
  argument-1: instance(employee);
  argument-2: department-role;
  axioms:
      ∀ E;employee
        employee-role(E, head-of-project)
          ↔
        ∃ P:project
          boss-of(E, P) ∧ size(P) = large

      ∀ E: employee
        employee-role(E, researcher)
          ↔
        ∃ P:project works-on(E, P) AND
          ¬ employee-role(E, head-of-group) AND
          ¬ employee-role(E, head-of-project);

concept *room;*
  properties:
      floor: string;
      number: nat;
      size: [small, medium, large];
      type: [office, other];
      location: [central, peripheral];

relation *distance;*
  argument-1: instance(room);
  argument-2: instance(room);
  properties:
    value: nat;
  semantics: symmetric;

relation *next-to;*
  argument-1: instance(room);
  argument-2: instance(room);
  semantics: symmetric;

relation *room-preference;*
  argument-1: department-role;
  argument-2: expression(room);
  tuples:
    < department-role, type(room) = office >
    < head-of-group, location(room) = central >
    < head-of-group, size(room) = large >
    < head-of-project, size(room) = small >
    < researcher, size(room) = large >
    < manager, size(room) = small >
    < secretary, size(room) = large >

relation *near-to-preference;*
  argument-1: department-role;
  argument-2: department-role;
  properties:
    strength: universal;
  semantics: symmetric;
  tuples:
    < head-of-group, head-of-project >
    < head-of-group, manager >
    < head-of-group, secretary > ;
  axioms:
    ∀ R1,R2,R3:department-role
      level(role-interaction(R1, R2) >
      level(role-interaction(R1, R3)
        →
      strength(near-to-pref(R1, R2)) >
      strength(near-to-pref(R1, R3));

structure *floor-plan;*
  parts:
    rooms:
      set(instance(room));
    room-relations:
      set(tuple(distance));
    set(tuple(next-to);
  properties:
    floor: string;
  axioms:
    ∀ R1,R2:room ∈ rooms

```
        value(distance(R1, R2)) =
        |number(R1) - number(R2)|

      ∀ R1,R2:room ∈ rooms
        next-to(R1, R2) ↔
        value(distance(R1, R2)) = 1;


structure department;
  parts:
      employees:
        set(instance(employee));
      projects:
        set(instance(project));
      assignments:
        set(tuple(works-on));
        set(tuple(head-of));
      employee-relations:
        set(tuple(works-well-with));
        set(tuple(head-of));
        set(tuple(employee-role));
  properties:
      name: string;


structure requirements;
  parts:
      room-related:
        set(tuple(room-pref));
      positional:
        set(tuple(near-to-pref));
      interactions:
        set(tuple(allocation-interaction));
```

Appendix B

Source Code Sisyphus Application

This appendix contains the source code for the example Sisyphus application. It constitutes an operationalisation of the model of expertise of the office-planning problem described in Ch. 7. The implementation follows the structure-preserving principles defined in Ch. 6.

B.1 Top-level module

Main

```
:- module(main, [off_plan/1]).

:- use_module(
  [ 'task-interpreter.pl'
  , 'domain-access.pl'
  ]).

:- dynamic
    trace/1,
    trace/2.

:- assert(main:trace(task)).
:- assert(main:trace(inference)).


off_plan(Allocations) :-
    domain_retrieval(find_all, component, Employees),
    domain_retrieval(find_all, resource,  Rooms),
    exec_task('propose allocations',
            [ 'components' = Employees
            , 'resources'  = Rooms
            ],
            'allocations' = Allocations).
```

B.2 Task-level modules

B.2.1 Generic modules

Task interpreter

```
:- module('task-interpreter.pl', [exec_task/3]).

:- use_module(
  [ 'task-declarations.pl'
  , 'task-working-memory.pl'
  , 'inference-functions.pl'
  ]).

%
% Task execution primitives
%

exec_task(Task, Input, Output) :-
    ignore(write_task(start, Task, Input)),
    task_interpreter(Task, Input, Output),
    ignore(write_task(end, Task, Output)).

task_interpreter(Task, Input, Output) :-
    init_task(Task, Input, Output),
    init_task_procedure(Task),
    once(task_procedure(Task)),
    retrieve_output(Task, Output).

init_task(Task, Input, Output) :-
    init(input, Task, Input),
    init(output, Task, Output),
    init(control_term, Task), !.

init(Type, Task, Arg) :-
    is_list(Arg), !,
    checklist(init(Type, Task), Arg).
init(input, Task, Name = Value) :-
    task_input(Task, Name),
    data_type(Name, DataType),
    data_operation(create, DataType, Name, Value).
init(input, Task, Name) :-
    task_input(Task, Name).
init(output, Task, Name = _) :-
    task_output(Task, Name),
    data_type(Name, DataType),
    data_operation(create, DataType, Name, []).
init(output, Task, Name) :-
    task_output(Task, Name).
init(control_term, Task) :-
    forall( control_term(Task, Name),
      ( data_type(Name, DataType)
      , data_operation(create, DataType, Name, [])
      )).

init_task_procedure(Task) :-
    (   clause(task_procedure(Task), _)
    ;   task_structure(Task, TaskStructure)
    ,   assert((task_procedure(Task) :- TaskStructure))
    ).

retrieve_output(Task, Output) :-
```

```
    is_list(Output), !,
    checklist(retrieve_output(Task), Output).
retrieve_output(Task, Name = Value) :-
    task_output(Task, Name),
    data_operation(retrieve, Name, Value).
retrieve_output(Task, Name) :-
    task_output(Task, Name).

%
% Inference execution
%

exec_inference(Inference, Input, Output) :-
    maplist(map(input), Input, In),
    invoke_inference(Inference, In, Out),
    map(output, Output, Out).

map(input, Input, In) :-
    nonvar(Input),
    data_type(Input, _), !,
    data_operation(retrieve, Input, In).
map(output, Output, Out) :-
    nonvar(Output),
    data_type(Output, _), !,
    data_operation(store, Output, Out).
map(_, Arg, Arg).

%
% Control primitives
%

repeat(Body, until(Condition)) :-
    repeat,
    once(Body),
    once(Condition), !.

transfer_task(obtain, Attribute, Attribute = Value) :-
    writef('\n Please enter the value of %w: ', [Attribute]),
    read(Value),
    nl.

%
% Trace info
%

write_task(start, Task, Input) :-
    ( main:trace(task) ; main:trace(task, Task) ) ->
    (   writef('\nActivating task "%w" \n', [Task])
    ,   write_arg('input ', Input)
    ).
write_task(end, Task, Output) :-
    ( main:trace(task) ; main:trace(task, Task) ) ->
    (   writef('\nTask "%w" terminated \n', [Task])
    ,   write_arg(output, Output)
    ).
```

```
write_arg(Type, Arg) :-
    is_list(Arg), !,
    checklist(write_arg(Type), Arg).
write_arg(Type, Name = Value) :-
    writef('  %w: %w = %w\n', [Type, Name, Value]).
write_arg(Type, Name) :-
    data_operation(retrieve, Name, Value),
    writef('  %w: %w = %w\n', [Type, Name, Value]).
```

Working memory

```
:- module('task-working-memory.pl',
  [ data_operation/3
  , data_operation/4
  ]).

:- use_module(['task-declarations.pl']).

:- dynamic
   data_store/3.

%
% Working memory primitives
%

data_operation(create, set, SetName, InitialValue) :-
    retractall(data_store(set, SetName, _)),
    assert(data_store(set, SetName, InitialValue)).
data_operation(create, list, ListName, InitialValue) :-
    retractall(data_store(list, ListName, _)),
    assert(data_store(list, ListName, InitialValue)).
data_operation(create, element, ElName, InitialValue) :-
    retractall(data_store(element, ElName, _)),
    assert(data_store(element, ElName, [InitialValue])).

data_operation(Op, Name, Store) :-
    nonvar(Store),
    data_type(Store, _), !,
    operation(retrieve, Store, Value),
    operation(Op, Name, Value),
    ignore(write_operation(Op, Name, Value)).
data_operation(Op, Name, Value) :-
    operation(Op, Name, Value),
    ignore(write_operation(Op, Name, Value)).

%
% Operations on all types (set, list, element)
%    store
%    retrieve
%

operation(store, SetName, Arg) :-
    check_arg(Arg, Set),
    retract(data_store(set, SetName, _)),
    assert(data_store(set, SetName, Set)).
operation(store, ListName, Arg) :-
    check_arg(Arg, List),
```

```
    retract(data_store(list, ListName, _)),
    assert(data_store(list, ListName, List)).
operation(store, ElName, Element) :-
    retract(data_store(element, ElName, _)),
    assert(data_store(element, ElName, [Element])).
operation(retrieve, SetName, Set) :-
    data_store(set, SetName, Set), !.
operation(retrieve, ListName, List) :-
    data_store(list, ListName, List).
operation(retrieve, ElName, Element) :-
    data_store(element, ElName, [Element]).

% Set operations:
%
%     member (returns a member randomly)
%     select (idem)
%     add
%     subtract
%     empty

operation(member, SetName, Member) :-
    data_store(set, SetName, Set),
    rnd_member(Member, Set).
operation(select, SetName, Member) :-
    retract(data_store(set, SetName, Set)),
    rnd_member(Member, Set),
    select(Set, Member, NewSet),
    assert(data_store(set, SetName, NewSet)), !.
operation(add, SetName, Arg) :-
    check_arg(Arg, Additions),
    retract(data_store(set, SetName, Set)),
    union(Set, Additions, NewSet),
    assert(data_store(set, SetName, NewSet)), !.
operation(subtract, SetName, Arg) :-
    check_arg(Arg, Deletions),
    retract(data_store(set, SetName, Set)),
    subtract(Set, Deletions, NewSet),
    assert(data_store(set, SetName, NewSet)), !.
operation(empty, Store, Bool) :-
    data_operation(retrieve, Store, []) -> Bool = true ; Bool = false.

% List operations:
%
%    member   (from first to last)
%    select   (idem)
%    select random
%    append
%    delete
%    empty    (similar to set operation)

operation(member, ListName, Member) :-
    data_store(list, ListName, List),
    member(Member, List).
operation(select, ListName, Selection) :-
    retract(data_store(list, ListName, List)),
    select(List, Selection, NewList),
```

```
    assert(data_store(list, ListName, NewList)), !.
operation('select random', ListName, Member) :-
    retract(data_store(list, ListName, List)),
    rnd_member(Member, List),
    select(List, Member, NewList),
    assert(data_store(list, ListName, NewList)), !.
operation(append, ListName, Arg) :-
    check_arg(Arg, Additions),
    retract(data_store(list, ListName, List)),
    append(List, Additions, NewList),
    assert(data_store(list, ListName, NewList)), !.
operation(delete, ListName, Element) :-
    retract(data_store(list, ListName, List)),
    delete(Element, List, NewList),
    assert(data_store(list, ListName, NewList)), !.

check_arg(Arg, Arg) :-
    is_list(Arg), !.
check_arg(Arg, [Arg]) :- !.

% Trace output

write_operation(Op, Store, Value) :-
    main:trace(working_memory, Ops),
    member(Op, Ops)
    -> write_op(Op, Store, Value).

write_op(Op, Store, Value) :-
    (   (Op==select ; Op=='select random' ; Op==member ; Op==empty)
    ,   writef('\nWorking memory operation "%w" on "%w"\n', [Op, Store])
    ,   writef('  with result: "%w"\n', [Value])
    ;   writef('\nWorking memory operation "%w %w" on "%w"\n', [Op, Value, Store])
    ).
```

B.2.2 Application-specific modules

Task declarations This module contains the model-of-expertise information about task knowledge.

```
:- module('task-declarations.pl',
  [ task/1
  , task_input/2
  , task_output/2
  , control_term/2
  , task_structure/2
  , data_type/2
  ]).

:- discontiguous
    task/1,
    task_input/2,
    task_output/2,
    control_term/2,
    task_structure/2,
    data_type/2.
```

```
%
% Task knowledge
%

task(          'propose allocations').
task_input(    'propose allocations',    'components').
task_input(    'propose allocations',    'resources').
task_output(   'propose allocations',    'allocations').
control_term(  'propose allocations',    'allocation plan').
control_term(  'propose allocations',    'plan element').

task_structure('propose allocations',
  ( exec_task('assemble plan', 'components', 'allocation plan')
  , forall(data_operation(member, 'allocation plan', Element),
      ( data_operation(store, 'plan element', Element)
      , exec_task('assign resources', ['plan element', 'resources'], 'allocations')
      ))
  )).

task(          'assemble plan').
task_input(    'assemble plan',     'components').
task_output(   'assemble plan',     'allocation plan').
control_term(  'assemble plan',     'component types').

task_structure('assemble plan',
  ( exec_task(classify, 'components',        'component types')
  , exec_task(order,    'component types',   'allocation plan')
  )).

task(          classify).
task_input(    classify,      'components').
task_output(   classify,      'component types').

task_structure(classify,
  ( forall( data_operation(member, components, C),
      ( exec_inference(classify, [C], CType)
      , data_operation(add, 'component types', CType)
      ))
  )).

task(          order).
task_input(    order,    'component types').
task_output(   order,    'allocation plan').
control_term(  order,    'sorted types').
control_term(  order,    'prime').

task_structure(order,
  ( exec_inference(select_1, ['component types'],           'prime')
  , exec_inference(sort,    ['component types', 'prime'], 'sorted types')
  , data_operation(append, 'allocation plan', 'prime')
  , data_operation(append, 'allocation plan', 'sorted types')
  )).

task(          'assign resources').
task_input(    'assign resources',   'plan element').
task_input(    'assign resources',   'resources').
```

```
task_output(   'assign resources',   'allocations').
control_term(  'assign resources',   'groupings').
control_term(  'assign resources',   'grouping').

task_structure('assign resources',
  ( exec_task(group, 'plan element', groupings)
  , data_operation(select, 'groupings', Grouping)
  , data_operation(store, 'grouping', Grouping)
  , forall( data_operation(member, 'grouping', Unit),
      exec_task(assign,
                  ['plan element'
                  , unit = Unit
                  , resources
                  , allocations
                  ],
                  allocations)
      )
  )).

task(          group).
task_input(    group,     'plan element').
task_output(   group,     'groupings').
control_term(  group,     'possible groupings').
control_term(  group,     'selected groupings').

task_structure(group,
  ( exec_inference(transform, ['plan element'], 'possible groupings')
  , exec_inference(select_2, ['possible groupings'
                             , criterion(minimise, major_conflict)
                             ],
                             'selected groupings')
  , exec_inference(select_2, ['selected groupings'
                             , criterion(maximise, major_synergy)
                             ],
                             'selected groupings')
  , exec_inference(select_2, ['selected groupings'
                             , criterion(maximise, minor_synergy)
                             ],
                             'selected groupings')
  , exec_inference(select_2, ['selected groupings'
                             , criterion(minimise, minor_conflict)
                             ],
                             'groupings')
  )).

task(          assign).
task_input(    assign,     'plan element').
task_input(    assign,     'unit').
task_input(    assign,     'allocations').
task_input(    assign,     'resources').
task_output(   assign,     'allocations').
task_output(   assign,     'resources').
control_term(  assign,     'suitable resources').

task_structure(assign,
  ( exec_inference(select_3,
```

```
                        [ 'plan element'
                        , 'resources'
                        ],
                        'suitable resources')
  , exec_inference(select_4,
                        [ 'plan element'
                        , 'suitable resources'
                        , 'allocations'
                        ],
                        'suitable resources')
  , data_operation(select, 'suitable resources', Resource)
  , data_operation(retrieve, 'unit', Unit)
  , data_operation(add, 'allocations', [[Resource, Unit]])
  , data_operation(subtract, 'resources', Resource)
  )).

data_type('components',                set).
data_type('resources',                 set).
data_type('allocations',               set).
data_type('allocation plan',           list).
data_type('plan element',              element).
data_type('component types',           set).
data_type('sorted types',              list).
data_type('prime',                     element).
data_type('groupings',                 set).
data_type('selected groupings',        set).
data_type('possible groupings',        set).
data_type('grouping',                  set).
data_type('unit',                      element).
data_type('suitable resources',        set).
```

B.3 Inference-level modules

B.3.1 Generic modules

Generic inference activation primititves

```
invoke_inference(Inference, Input, Output) :-
    (   once(inference_function(Inference, Input, Output))
    ,   ignore(write_trace(Inference, Input, Output))
    ;   Output = []
    ).

write_trace(Inference, Input, Output) :-
    (   main:trace(inference)
    ;   main:trace(inference, Inference)
    ) ->
    write_inf(Inference),
    write_mc(Inference, Input),
    write_mc(Inference, Output, output).

write_inf(Inf) :-
    inference(Inf, String),
    (  String == [] -> Name = Inf ; Name = String),
    writef('\nInvoking inference %w\n', [Name]).
write_mc(Inference, Input) :-
```

```
    forall( nth1(Num, Input, MC), write_mc(Inference, MC, input(Num))).
write_mc(Inference, Value, input(Num)) :-
    metaclass(Inference, input(Num), GenName, SpecName),
    domain_retrieval(find_type, GenName, DomainEntity),
    (   var(SpecName) -> Name = GenName ; Name = SpecName),
    writef('  input :  %w "%w" (a %w)\n', [Name, Value, DomainEntity]).
write_mc(Inference, Value, output) :-
    metaclass(Inference, output, GenName, SpecName),
    domain_retrieval(find_type, GenName, DomainEntity),
    (   var(SpecName) -> Name = GenName ; Name = SpecName),
    writef('  output:  %w "%w" (a %w)\n', [Name, Value, DomainEntity]).
```

B.3.2 Application-specific modules

Inference functions

```
:- module('inference-functions.pl', [ invoke_inference/3 ]).

:- ensure_loaded(['inference-activation.pl']).

:- use_module(
  [ 'inference-methods.pl'
  , 'inference-declarations.pl'
  , 'domain-access.pl'
  ]).

:- discontiguous
    inference_function/3.

inference_function(classify, [In], Out) :-
    domain_retrieval(find_one, type_association, [In, Out]).

inference_function(select_1, _, Out) :-
    domain_retrieval(find_all, hierarchy, Hierarchy),
    hierarchy_search(Out, Hierarchy, is_root_node).

inference_function(sort, [In, Prime], Out) :-
    delete(In, Prime, Rest),
    predsort(compare_interaction(Prime), Rest, Out).

compare_interaction(Prime, C1, C2) :-
    domain_retrieval(find_one, interaction_level, [V1, Prime, C1]),
    domain_retrieval(find_one, interaction_level, [V2, Prime, C2]),
    domain_retrieval(find_one, level, Values),
    nth1(N1, Values, V1),
    nth1(N2, Values, V2),
    N1 > N2.

inference_function(transform, [Ctype], Structures) :-
    domain_retrieval(find_all, type_association, Ass),
    findall(C, member([C, Ctype], Ass), Cset),
    domain_retrieval(find_one, structure, [StrucType, Ctype]),
    (   StrucType = single
    ,   Structures = [Cset]
    ;   StrucType =  shared
    ,   pair_permutations(Cset, Structures)
```

```
    ).

inference_function(select_2, [Set, criterion(MinMax, CritType)], SubSet) :-
    partition_set(Set, SubSet, MinMax, _, grouping_criterion(CritType),
       (grouping_criterion(Criterion, Structure, Num) :-
           findall(Pair,
             ( member(Pair, Structure)
             , 'inference-functions.pl':domain_retrieval(find_one, Criterion, Pair)
             ),
             Pairs),
           length(Pairs, Num)
       )
    ).

inference_function(select_3, [Type, Resources], SubSet) :-
    partition_set(Resources, SubSet, minimise, 0, resource_criterion,
       (resource_criterion(Res, Num) :-
           'inference-functions.pl':domain_retrieval(find_all,
                                           resource_requirement,
                                           Reqs),
           assert(number_of_conflicts(0)),
           forall(member([Type, Expression], Reqs),
               (   Expression =.. [equal, Prop, Value]
               ,   'inference-functions.pl':domain_retrieval(find_one,
                                                   resource_expr,
                                                   [Res, Prop, Value])
               ;   Expression =.. [not_equal, Prop, Value]
               ,   'inference-functions.pl':domain_retrieval(find_one,
                                                   resource_expr,
                                                   [Res, Prop, Value])
               ;   retract(number_of_conflicts(OldNum))
               ,   succ(OldNum, NewNum)
               ,   assert(number_of_conflicts(NewNum))
               )
             ),
           retract(number_of_conflicts(Num))
        )
    ).

inference_function(select_4, [Type, Resources, Allocations], SubSet) :-
    partition_set(Resources, SubSet, minimise, _, positional_criterion(Allocations),
       (positional_criterion(Allocations, Res, Num) :-
         ( 'inference-functions.pl':domain_retrieval(find_one,
                                           positional_requirement,
                                           [Type, C])
         , 'inference-functions.pl':domain_retrieval(find_all,
                                           type_association,
                                           Ass)
         , member([I, C], Ass)
         , (   member([IRes, I], Allocations)
           ;   member([IRes, Is], Allocations)
           ,   member(I, Is)
           )
         , 'inference-functions.pl':domain_retrieval(find_one,
                                           position,
                                           [Num, Res, IRes])
```

```
        ; Num = 0
        )
      )
    ).
```

Inference declarations This module contains the model-of-expertise information about inference knowledge.

```
:- module('inference-declarations.pl',
  [ inference/2
  , metaclass/4
  , domain_view/2
  ]).

:- discontiguous
    inference/2,
    metaclass/4,
    domain_view/2.

inference(  classify,   'Classify components').
metaclass(  classify,   input(1),       component,      _).
metaclass(  classify,   output,         component_type, _).
domain_view(classify,   relation(type_association, component, component_type)).

inference(  select_1,   'Select prime').
metaclass(  select_1,   input(1),       set(component_type),    all_types).
metaclass(  select_1,   output,         component_type,         prime).
domain_view(select_1,   relation(hierarchy, component_type, component_type)).

inference(  sort,       'Sort types').
metaclass(  sort,       input(1),       set(component_type),  unsorted_types).
metaclass(  sort,       input(2),       component,       sort_criterion).
metaclass(  sort,       output,         list(component_type), sorted_types).
domain_view(sort,       relation(interaction_level,
                                         level,
                                         component_type,
                                         component_type)).

inference(  transform, 'Transform into possible groupings').
metaclass(  transform, input(1),        component_type, _).
metaclass(  transform, output,          set(structure(component)), groupings).
domain_view(transform, relation(structure, structure_type, component_type)).
domain_view(transform, relation(type_association, component, component_type)).

inference(  select_2, 'Select suitable groupings').
metaclass(  select_2, input(1), set(structure(component)),
                                'current groupings').
metaclass(  select_2, input(2), criterion,
                                'selection criterion').
metaclass(  select_2, output,   set(structure(component)),
                                'subset of groupings').
domain_view(select_2, relation(major_conflict, component, component)).
domain_view(select_2, relation(minor_conflict, component, component)).
domain_view(select_2, relation(major_synergy,  component, component)).
domain_view(select_2, relation(major_synergy,  component, component)).
```

```
inference(  select_3,   'Select on resource requirements').
metaclass(  select_3,   input(1),       component_type, _).
metaclass(  select_3,   input(2),       set(resource),
                                        'available resources').
metaclass(  select_3,   output,         set(resource),
                                        'suitable resources').
domain_view(select_3,   relation(resource_requirement,
                                        component_type, expression(resource))).

inference(  select_4,   'Select on positional requirements').
metaclass(  select_4,   input(1),       component_type, _).
metaclass(  select_4,   input(2),       set(resource),
                                        'available resources').
metaclass(  select_4,   input(3),       structure(resource, list(component)),
                                        'current positions').
metaclass(  select_4,   output,         set(resource),
                                        'suitable resources').
domain_view(select_4,   relation(positional_requirement,
                                        component_type, compoenent_type)).
domain_view(select_4,   relation(position, integer,
                                        instance(resource),
                                        instance(resource))).
```

Inference methods This module defines the inference methods for realising particluar inferences. The *partition*$_s$*et* method is used by three inference functions, but for different purposes.

```
:- module('inference-methods.pl',
  [ hierarchy_search/3
  , partition_set/6
  , pair_permutations/2
  ]).


% Computational techniques useed by inferences
% (NB. the sort method is a built-in SWI-Prolog method}

hierarchy_search(Root, Hierarchy, is_root_node) :-
    member([Root, _], Hierarchy),
    \+ member([_, Root], Hierarchy).

partition_set(Set, SubSet, MinMax, Rating, CritHead, CritClause) :-
    assert(CritClause),
    findall(Num/E,
      ( member(E, Set)
      , CritHead =.. Head
      , append(Head, [E, Num], TermList)
      , Pred =.. TermList
      , once(Pred)
      ),
      Rated),
      keysort(Rated, Sorted),
      (   MinMax == minimise
      ,   first(Rating/Element, Sorted)
      ;   MinMax == maximise
      ,   last(Rating/Element, Sorted)
      ),
```

```
    retract(CritClause),
    findall(E, member(Rating/E, Sorted), SubSet).

first(E, L) :-
    member(E, L), !.

pair_permutations(Set, Perms) :-
    findall(Perm, pair_perm(Set, Perm), Perms).

pair_perm(Set, [E | Rest] ):-
    length(Set, L),
    odd(L), !,
    select(Set, E, SubSet),
    pair_perm2(SubSet, Rest).
pair_perm(Set, Perm) :-
    pair_perm2(Set, Perm).

pair_perm2([], []).
pair_perm2(Set, [[E1, E2] | Rest]) :-
    select(Set, E1, Tmp), !,
    select(Tmp, E2, SubSet),
    pair_perm2(SubSet, Rest).

odd(N) :-
    Rem is N mod 2,
    Rem == 1.
even(N) :-
    \+ odd(N).
```

B.4 Domain-level modules

B.4.1 Generic modules

Domain access This module defines the access functions that can be used by the inference functions to retrieve domain knowledge from the knowledge base. The module uses indices defined in the module *domain-index* to to map the inference-level names onto domain-specific queries.

```
:- module('domain-access.pl', [domain_retrieval/3]).

:- discontiguous
    concept/1, concept/2, set/2, structure/2, relation/3, property/3,
    semantics/2, instance/3, value/3, tuple/2.

:- dynamic
    concept/1, concept/2, set/2, structure/2, relation/3, property/3,
    semantics/2, instance/3, value/3, tuple/2.

:- multifile
    value/3, tuple/2.

:- use_module([ 'domain-index.pl' ]).

:- ensure_loaded( [ 'domain-data.pl', 'domain-theory.pl']).

%
```

```
% Domain-knowledge access primitives
%

domain_retrieval(find_one, InfRelation, Tuple) :-
    domain_index(relation, InfRelation, DomainKnowTypes),
    member(Type, DomainKnowTypes),
    get_relation(Type, Tuple), !.
domain_retrieval(find_one, InfObject, Obj) :-
    domain_index(entity, InfObject, DomainKnowTypes),
    member(Type, DomainKnowTypes),
    get_entity(Type, Obj), !.
domain_retrieval(find_one, InfExpr, Expr) :-
    domain_index(expression, InfExpr, DomainKnowTypes),
    member(Type, DomainKnowTypes),
    get_expression(Type, Expr), !.

domain_retrieval(find_all, InfRelation, Knowledge) :-
    domain_index(relation, InfRelation, DomainKnowTypes), !,
    findall(KnowElement,
      ( member(Type, DomainKnowTypes)
      , get_relation(Type, KnowElement)
      ),
      Knowledge).
domain_retrieval(find_all, InfObject, Knowledge) :-
    domain_index(entity, InfObject, DomainKnowTypes), !,
    findall(KnowElement,
      ( member(Type, DomainKnowTypes)
      , get_entity(Type, KnowElement)
      ),
      Knowledge).

domain_retrieval(find_type, structure(I1, I2), structure(D1, D2)) :-
    domain_retrieval(find_type, I1, D1),
    domain_retrieval(find_type, I2, D2).
domain_retrieval(find_type, structure(InfName), structure(DomainEntity)) :-
    domain_retrieval(find_type, InfName, DomainEntity).
domain_retrieval(find_type, set(InfName), set(DomainEntity)) :-
    domain_retrieval(find_type, InfName, DomainEntity).
domain_retrieval(find_type, list(InfName), list(DomainEntity)) :-
    domain_retrieval(find_type, InfName, DomainEntity).
domain_retrieval(find_type, InfName, one_of(Defs)) :-
    domain_index(def, InfName, Defs).
domain_retrieval(find_type, InfName, DomainEntity) :-
    domain_index(_, InfName, DomainEntity).

get_relation(relation(RelationName), Tuple) :-
    get_tuple(RelationName, Tuple).
get_relation(property(Prop, ObjectType), [Value, Object]) :-
    get_object(ObjectType, Object),
    get_value(Object, Prop, Value).
get_relation(property(Prop, relation(RelType)), [Value, Obj1, Obj2]) :-
    get_value(tuple(RelType, [Obj1, Obj2]), Prop, Value).

get_entity(instance(C), I) :-
    get_instance(C, I).
get_entity(concept(C), SubC) :-
```

```
    get_concept(C, SubC).
get_entity(value_set(Prop, Type), Values) :-
    property(Type, Prop, Values).
get_entity(A, A) :-
    atom(A).

get_expression(expression(ObjType), [Obj, Prop, Value]) :-
    get_object(ObjType, Obj),
    get_value(Obj, Prop, Value).

get_object(Type, Object) :-
    get_instance(Type, Object).
get_object(Type, Object) :-
    get_concept(Type, Object).

get_instance(C, I) :-
    instance(C, I, _).
get_concept(C1, C2) :-
    sub_concept(C2, C1).

sub_concept(C1, C2) :-
    concept(C1, Supers),
    memberchk(C2, Supers).
sub_concept(C1, C2) :-
    concept(C1, Supers),
    member(C, Supers),
    sub_concept(C, C2).

get_value(Object, Property, Value) :-
    value(Object, Property, Value).
get_value(Object, Property, Value) :-
    instance(_, Object, PropValues),
    memberchk(Property = Value, PropValues).

get_tuple(Relation, Tuple) :-
    tuple(Relation, Tuple).
get_tuple(Relation, [Arg1, Arg2]) :-
    semantics(relation(Relation), symmetric),
    tuple(Relation, [Arg2, Arg1]).
get_tuple(Relation, [Arg1, Arg2]) :-
    semantics(relation(Relation), transitive),
    tuple(Relation, [Arg1, Tmp]),
    get_tuple(Relation, [Tmp, Arg2]).
```

B.4.2 Application-specific modules

Domain index The domain index defines the mapping from inference-level terms (meta-classes, domain views) onto domain-specific terminology.

```
:- module('domain-index.pl', [domain_index/3]).

domain_index(entity,     component,             [instance(employee)]).
domain_index(entity,     component_type,        [concept(department_role)]).
domain_index(relation,   type_association,      [relation(employee_role)]).
domain_index(relation,   hierarchy,             [relation(boss_of)]).
domain_index(relation,   interaction_level,     [property(
```

```
                                                level,
                                                relation(interaction))]).
domain_index(entity,     level,                 [value_set(level, interaction)]).
domain_index(entity,     resource,              [instance(room) ]).
domain_index(relation,   resource_requirement,  [relation(room_preference)]).
domain_index(expression, resource_expr,         [expression(room)]).
domain_index(relation,   positional_requirement, [relation(nearto_preference)]).
domain_index(relation,   position,              [property(value, relation(distance))]).
domain_index(entity,     structure_type,        [value_set(occupancy, department_role)]).
domain_index(relation,   structure,             [property(occupancy, department_role)]).
domain_index(relation,   major_conflict,        [relation(smoker_and_non_smoker)]).
domain_index(relation,   major_synergy,         [relation(on_different_projects)]).
domain_index(relation,   minor_conflict,        [relation(hacker_and_non_hacker)]).
domain_index(relation,   minor_synergy,         [relation(works_with) ]).
domain_index(def,        criterion,             [ minimise(major_conflict)
                                                , minimise(minor_conflict)
                                                , maximise(major_synergy)
                                                , maximise(minor_synergy)
                                                ]).
```

Domain theory This file contains the operationaled version of the Sisyphus domain knowledge defined in Appendix A.

```
concept(employee).
property(employee,             hacker,                  bool).
property(employee,             smoker,                  bool).
relation(employee_role,        instance(employee),      department_role).
relation(works_on,             instance(employee),      instance(project)).
relation(head_of,              instance(employee),      instance(project)).
relation(allocation,           set(instance(employee)), instance(room)).

relation(works_with,           instance(employee),      instance(employee)).
relation(smoker_and_non_smoker, instance(employee),     instance(employee)).
relation(on_different_projects, instance(employee),     instance(employee)).
relation(hacker_and_non_hacker, instance(employee),     instance(employee)).

semantics(relation(works_with),                symmetric).
semantics(relation(smoker_and_non_smoker),     symmetric).
semantics(relation(on_different_projects),     symmetric).
semantics(relation(hacker_and_non_hacker),     symmetric).

tuple(smoker_and_non_smoker, [Emp1, Emp2]) :-
    get_value(Emp1, smoker, false),
    get_value(Emp2, smoker, true).
tuple(hacker_and_non_hacker, [Emp1, Emp2]) :-
    get_value(Emp1, hacker, false),
    get_value(Emp2, hacker, true).
tuple(on_different_projects, [Emp1, Emp2]) :-
    get_tuple(works_on, [Emp1, Project1]),
    get_tuple(works_on, [Emp2, Project2]),
    Project1 \== Project2.

concept(department_role).
concept(head_of_group,    [department_role]).
concept(manager,          [department_role]).
concept(secretary,        [department_role]).
```

```
concept(head_of_project,  [department_role]).
concept(researcher,       [department_role]).
property(department_role, occupancy, [single, shared]).
relation(boss_of,       department_role, department_role).
semantics(relation(boss_of), transitive).

tuple(boss_of, [head_of_group, manager]).
tuple(boss_of, [head_of_group, head_of_project]).
tuple(boss_of, [manager, secretary]).
tuple(boss_of, [head_of_project, researcher]).

tuple(employee_role, [Employee, head_of_project]) :-
    get_instance(project, Project),
    get_value(Project, size, large),
    tuple(head_of, [Employee, Project]).
tuple(employee_role, [Employee, researcher]) :-
    tuple(works_on, [Employee, _SomeProject]),
    \+ tuple(employee_role, [Employee, head_of_project]),
    \+ tuple(employee_role, [Employee, head_of_group]).

value(head_of_group,    occupancy, single).
value(head_of_project,  occupancy, single).
value(manager,          occupancy, single).
value(secretary,        occupancy, shared).
value(researcher,       occupancy, shared).

relation(interaction, department_role, department_role).
property(interaction, level, [normal, above_normal, high, very_high]).

value(tuple(interaction, [head_of_group, secretary]),     level, very_high).
value(tuple(interaction, [head_of_group, manager]),       level, high).
value(tuple(interaction, [head_of_group,head_of_project]),level, above_normal).
value(tuple(interaction, [head_of_group, researcher]),    level, normal).

concept(project).
property(project,       size,           [small, medium, large]).

concept(room).
property(room, floor,           string).
property(room, number,          nat).
property(room, size,            [small, medium, large]).
property(room, type,            [office, other]).
property(room, location,        [central, peripheral]).
relation(next_to,               instance(room),         instance(room)).
relation(distance,              instance(room),         instance(room)).
property(distance,              value,                  nat).
semantics(relation(next_to),    symmetric).
semantics(relation(distance),   symmetric).

tuple(next_to, [Room1, Room2]) :-
    get_value(Room1, number, Num1),
    get_value(Room2, number, Num2),
    Diff = Num1 - Num2,
    abs(Diff) == 1.
value(tuple(distance, [Room1, Room2]), value, Distance) :-
    get_value(Room1, number, Num1),
```

```
    get_value(Room2, number, Num2),
    Diff = Num1 - Num2,
    Distance is abs(Diff).

relation(room_preference, department_role, expr(room)).

tuple(room_preference, [head_of_group,   equal(location,     central)]).
tuple(room_preference, [head_of_group,   equal(type,         office)]).
tuple(room_preference, [head_of_group,   equal(size,         large)]).

tuple(room_preference, [manager,         equal(size,         small)]).
tuple(room_preference, [manager,         equal(type,         office)]).

tuple(room_preference, [secretary,       equal(size,         large)]).
tuple(room_preference, [secretary,       equal(type,         office)]).

tuple(room_preference, [head_of_project, equal(size,         small)]).
tuple(room_preference, [head_of_project, equal(type,         office)]).

tuple(room_preference, [researcher,      equal(size,         large)]).
tuple(room_preference, [researcher,      equal(type,         office)]).

relation(nearto_preference, department_role, department_role).
semantics(relation(nearto_preference), symmetric).
property(nearto_preference, strength, [normal, above_normal, high, very_high]).
tuple(nearto_preference, [head_of_group, secretary]).
tuple(nearto_preference, [head_of_group, manager]).
tuple(nearto_preference, [head_of_group, head_of_project]).
value(nearto_preference(Role1, Role2), strength, Value) :-
   get_value(role_interaction(Role1, Role2), level, Value).
```

Domain data This file contains the Sisyphus example data set.

```
instance(employee, 'Werner L.',     [smoker = false,    hacker = true]).
instance(employee, 'Marc M.',       [smoker = false,    hacker = true]).
instance(employee, 'Angi W.',       [smoker = false,    hacker = false]).
instance(employee, 'Juergen L.',    [smoker = false,    hacker = true]).
instance(employee, 'Andy L.',       [smoker = true,     hacker = false]).
instance(employee, 'Michael T.',    [smoker = false,    hacker = true]).
instance(employee, 'Harry C.',      [smoker = false,    hacker = true]).
instance(employee, 'Uwe T.',        [smoker = true,     hacker = true]).
instance(employee, 'Thomas D.',     [smoker = false,    hacker = false]).
instance(employee, 'Monika X.',     [smoker = false,    hacker = false]).
instance(employee, 'Ulrike U.',     [smoker = false,    hacker = false]).
instance(employee, 'Hans W.',       [smoker = true,     hacker = false]).
instance(employee, 'Eva I.',        [smoker = false,    hacker = false]).
instance(employee, 'Joachim I.',    [smoker = false,    hacker = false]).
instance(employee, 'Katharina N.',  [smoker = true,     hacker = true]).

tuple(employee_role, ['Thomas D.',       head_of_group]).
tuple(employee_role, ['Eva I.',          manager]).
tuple(employee_role, ['Monika X.',       secretary]).
tuple(employee_role, ['Ulrike U.',       secretary]).

instance(project, 'BABYLON Product',    [size = large]).
instance(project, 'ASERTI',             [size = large]).
```

```
instance(project, 'MLT',                [size = large]).
instance(project, 'RESPECT',            [size = medium]).
instance(project, 'EULISP',             [size = medium]).
instance(project, 'KRITON',             [size = small]).
instance(project, 'TUTOR2000',          [size = small]).
instance(project, 'Autonomous systems', [size = small]).

tuple(head_of, ['Hans W.',       'BABYLON Product']).
tuple(head_of, ['Joachim I.',    'ASERTI']).
tuple(head_of, ['Katharina N.',  'MLT']).
tuple(head_of, ['Angi W.',       'RESPECT']).
tuple(head_of, ['Thomas D.',     'EULISP']).

tuple(works_on, ['Werner L.',     'RESPECT']).
tuple(works_on, ['Marc M.',       'KRITON']).
tuple(works_on, ['Angi W.',       'RESPECT']).
tuple(works_on, ['Juergen L.',    'EULISP']).
tuple(works_on, ['Harry C.',      'BABYLON Product']).
tuple(works_on, ['Thomas D.',     'EULISP']).
tuple(works_on, ['Michael T.',    'BABYLON Product']).
tuple(works_on, ['Andy L.',       'TUTOR2000']).
tuple(works_on, ['Uwe T.',        'Autonomous systems']).
tuple(works_on, ['Hans W.',       'BABYLON Product']).
tuple(works_on, ['Joachim I.',    'ASERTI']).
tuple(works_on, ['Katharina N.',  'MLT']).

tuple(works_with, ['Werner L.',   'Angi W.']).
tuple(works_with, ['Werner L.',   'Marc M.']).
tuple(works_with, ['Angi W.',     'Marc M.']).
tuple(works_with, ['Angi W.',     'Werner L.']).
tuple(works_with, ['Michael T.',  'Hans W.']).
tuple(works_with, ['Thomas D.',   'Harry C.']).
tuple(works_with, ['Thomas D.',   'Juergen L.']).
tuple(works_with, ['Harry C.',    'Juergen L.']).
tuple(works_with, ['Eva I.',      'Thomas D.']).
tuple(works_with, ['Eva I.',      'Monika X.']).
tuple(works_with, ['Eva I.',      'Ulrike U.']).
tuple(works_with, ['Monika X.',   'Thomas D.']).
tuple(works_with, ['Monika X.',   'Ulrike U.']).
tuple(works_with, ['Thomas D.',   'Ulrike U.']).

instance(room, 'C5-113', [number = 113,  size = small,   type = office   ]).
instance(room, 'C5-114', [number = 114,  size = small,   type = office   ]).
instance(room, 'C5-115', [number = 115,  size = small,   type = office   ]).
instance(room, 'C5-116', [number = 116,  size = small,   type = office   ]).
instance(room, 'C5-117', [number = 117,  size = large,   type = office   ]).
instance(room, 'C5-118', [number = 118,  size = large,   type = other    ]).
instance(room, 'C5-119', [number = 119,  size = large,   type = office   ]).
instance(room, 'C5-120', [number = 120,  size = large,   type = office   ]).
instance(room, 'C5-121', [number = 121,  size = large,   type = office   ]).
instance(room, 'C5-122', [number = 122,  size = large,   type = office   ]).
instance(room, 'C5-123', [number = 123,  size = large,   type = office   ]).

value('C5-117', location, central).
value('C5-118', location, central).
value('C5-119', location, central).
```

B.5 Example output

B.5.1 Full task-level trace

```
4 ?- off_plan(_).

Activating task "propose allocations"
  input : components = [Werner L.,Marc M.,Angi W.,Juergen L.,Andy
                        L.,Michael T.,Harry C.,Uwe T.,Thomas D.,Monika X.
                        ,Ulrike U. ,Hans W. ,Eva I.,Joachim I.,Katharina N.]
  input : resources = [C5-113,C5-114,C5-115,C5-116,C5-117,C5-118
                       ,C5-119,C5-120,C5-121,C5-122,C5-123]

Activating task "assemble plan"
  input : components = [Werner L.,Marc M.,Angi W.,Juergen L.,Andy
                        L.,Michael T.,Harry C.,Uwe T.,Thomas D.,Monika
                        X. ,Ulrike U. ,Hans W. ,Eva I.,Joachim I.,Katharina N.]

Activating task "classify"
  input : components = [Werner L.,Marc M.,Angi W.,Juergen L.,Andy
                        L.,Michael T.,Harry C.,Uwe T.,Thomas D.,Monika
                        X. ,Ulrike U. ,Hans W. ,Eva I.,Joachim I.,Katharina N.]

Task "classify" terminated
  output: component types =
          [manager,head_of_group,head_of_project,secretary,researcher]

Activating task "order"
  input : component types =
          [manager,head_of_group,head_of_project,secretary,researcher]

Task "order" terminated
  output: allocation plan =
          [head_of_group,secretary,manager,head_of_project,researcher]

Task "assemble plan" terminated
  output: allocation plan =
          [head_of_group,secretary,manager,head_of_project,researcher]

Activating task "assign resources"
  input : plan element = head_of_group
  input : resources = [C5-113,C5-114,C5-115,C5-116,C5-117,C5-118
                       ,C5-119,C5-120,C5-121,C5-122,C5-123]

Activating task "group"
  input : plan element = head_of_group

Task "group" terminated
  output: groupings = [[Thomas D.]]

Activating task "assign"
  input : plan element = head_of_group
  input : unit = Thomas D.
  input : resources = [C5-113,C5-114,C5-115,C5-116,C5-117,C5-118
                       ,C5-119,C5-120,C5-121,C5-122,C5-123]
  input : allocations = []
```

```
Task "assign" terminated
  output: allocations = [[C5-117,Thomas D.]]

Task "assign resources" terminated
  output: allocations = [[C5-117,Thomas D.]]

Activating task "assign resources"
  input : plan element = secretary
  input : resources = [C5-113,C5-114,C5-115,C5-116,C5-118,C5-119
                      ,C5-120,C5-121,C5-122,C5-123]

Activating task "group"
  input : plan element = secretary

Task "group" terminated
  output: groupings = [[[Monika X.,Ulrike U.]]]

Activating task "assign"
  input : plan element = secretary
  input : unit = [Monika X.,Ulrike U.]
  input : resources = [C5-113,C5-114,C5-115,C5-116,C5-118,C5-119
                      ,C5-120,C5-121,C5-122,C5-123]
  input : allocations = [[C5-117,Thomas D.]]

Task "assign" terminated
  output: allocations = [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]]]

Task "assign resources" terminated
  output: allocations = [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]]]

Activating task "assign resources"
  input : plan element = manager
  input : resources = [C5-113,C5-114,C5-115,C5-116,C5-118,C5-120
                      ,C5-121,C5-122,C5-123]

Activating task "group"
  input : plan element = manager

Task "group" terminated
  output: groupings = [[Eva I.]]

Activating task "assign"
  input : plan element = manager
  input : unit = Eva I.
  input : resources = [C5-113,C5-114,C5-115,C5-116,C5-118,C5-120
                      ,C5-121,C5-122,C5-123]
  input : allocations = [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]]]

Task "assign" terminated
  output: allocations =
  [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]],[C5-116,Eva I.]]

Task "assign resources" terminated
  output: allocations =
  [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]],[C5-116,Eva I.]]
```

```
Activating task "assign resources"
  input : plan element = head_of_project
  input : resources = [C5-113,C5-114,C5-115,C5-118,C5-120,C5-121,C5-122,C5-123]

Activating task "group"
  input : plan element = head_of_project

Task "group" terminated
  output: groupings = [[Hans W.,Joachim I.,Katharina N.]]

Activating task "assign"
  input : plan element = head_of_project
  input : unit = Joachim I.
  input : resources = [C5-113,C5-114,C5-115,C5-118,C5-120,C5-121,C5-122,C5-123]
  input : allocations =[[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]],[C5-116,Eva I.]]

Task "assign" terminated
  output: allocations =
  [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]],[C5-116,Eva I.],
  [C5-115,Joachim I.]]

Activating task "assign"
  input : plan element = head_of_project
  input : unit = Katharina N.
  input : resources = [C5-113,C5-114,C5-118,C5-120,C5-121,C5-122,C5-123]
  input : allocations =
  [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]],[C5-116,Eva I.],
  [C5-115,Joachim I.]]

Task "assign" terminated
  output: allocations =
  [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]],[C5-116,Eva I.],
  [C5-115,Joachim I.],[C5-114,Katharina N.]]

Activating task "assign"
  input : plan element = head_of_project
  input : unit = Hans W.
  input : resources = [C5-113,C5-118,C5-120,C5-121,C5-122,C5-123]
  input : allocations =
  [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]],[C5-116,Eva I.],
  [C5-115,Joachim I.],[C5-114,Katharina N.]]

Task "assign" terminated
  output: allocations =
  [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]],[C5-116,Eva I.],
  [C5-115,Joachim I.],[C5-114,Katharina N.],[C5-113,Hans W.]]

Task "assign resources" terminated
  output: allocations =
  [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]],[C5-116,Eva I.],
  [C5-115,Joachim I.],[C5-114,Katharina N.],[C5-113,Hans W.]]

Activating task "assign resources"
  input : plan element = researcher
  input : resources = [C5-118,C5-120,C5-121,C5-122,C5-123]
```

```
Activating task "group"
  input : plan element = researcher

Task "group" terminated
  output: groupings = [
  [[Werner L.,Marc M.],[Angi W.,Michael T.],[Juergen L.,Harry C.],[Andy L.,Uwe T.]],
  [[Werner L.,Michael T.],[Marc M.,Angi W.],[Juergen L.,Harry C.],[Andy L.,Uwe T.]]]

Working memory operation "select" on "groupings"
  with result: "
  [[Werner L.,Michael T.],[Marc M.,Angi W.],[Juergen L.,Harry C.],[Andy L.,Uwe T.]]"

Activating task "assign"
  input : plan element = researcher
  input : unit = [Andy L.,Uwe T.]
  input : resources = [C5-118,C5-120,C5-121,C5-122,C5-123]
  input : allocations =
  [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]],[C5-116,Eva I.],
  [C5-115,Joachim I.],[C5-114,Katharina N.],[C5-113,Hans W.]]

Task "assign" terminated
  output: allocations =
  [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]],[C5-116,Eva I.],
  [C5-115,Joachim I.],[C5-114,Katharina N.],[C5-113,Hans W.],
  [C5-120,[Andy L.,Uwe T.]]]

Activating task "assign"
  input : plan element = researcher
  input : unit = [Marc M.,Angi W.]
  input : resources = [C5-118,C5-121,C5-122,C5-123]
  input : allocations =
  [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]],[C5-116,Eva I.],
  [C5-115,Joachim I.],[C5-114,Katharina N.],[C5-113,Hans W.],
  [C5-120,[Andy L.,Uwe T.]]]

Task "assign" terminated
  output: allocations =
  [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]],[C5-116,Eva I.],
  [C5-115,Joachim I.],[C5-114,Katharina N.],[C5-113,Hans W.],
  [C5-120,[Andy L.,Uwe T.]],[C5-122,[Marc M.,Angi W.]]]

Activating task "assign"
  input : plan element = researcher
  input : unit = [Werner L.,Michael T.]
  input : resources = [C5-118,C5-121,C5-123]
  input : allocations =
  [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]],[C5-116,Eva I.],
  [C5-115,Joachim I.],[C5-114,Katharina N.],[C5-113,Hans W.],
  [C5-120,[Andy L.,Uwe T.]],[C5-122,[Marc M.,Angi W.]]]

Task "assign" terminated
  output: allocations =
  [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]],[C5-116,Eva I.],
  [C5-115,Joachim I.],[C5-114,Katharina N.],[C5-113,Hans W.],
  [C5-120,[Andy L.,Uwe T.]],[C5-122,[Marc M.,Angi W.]],
  [C5-121,[Werner L.,Michael T.]]]
```

```
Activating task "assign"
  input : plan element = researcher
  input : unit = [Juergen L.,Harry C.]
  input : resources = [C5-118,C5-123]
  input : allocations =
  [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]],[C5-116,Eva I.],
  [C5-115,Joachim I.],[C5-114,Katharina N.],[C5-113,Hans W.],
  [C5-120,[Andy L.,Uwe T.]],[C5-122,[Marc M.,Angi W.]],
  [C5-121,[Werner L.,Michael T.]]]

Task "assign" terminated
  output: allocations =
   [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]],[C5-116,Eva I.],
   [C5-115,Joachim I.],[C5-114,Katharina N.],[C5-113,Hans W.],
   [C5-120,[Andy L.,Uwe T.]],[C5-122,[Marc M.,Angi W.]],
   [C5-121,[Werner L.,Michael T.]],[C5-123,[Juergen L.,Harry C.]]]

Task "assign resources" terminated
  output: allocations =
   [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]],[C5-116,Eva I.],
   [C5-115,Joachim I.],[C5-114,Katharina N.],[C5-113,Hans W.],
   [C5-120,[Andy L.,Uwe T.]],[C5-122,[Marc M.,Angi W.]],
   [C5-121,[Werner L.,Michael T.]],[C5-123,[Juergen L.,Harry C.]]]

Task "propose allocations" terminated
  output: allocations =
   [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]],[C5-116,Eva I.],
   [C5-115,Joachim I.],[C5-114,Katharina N.],[C5-113,Hans W.],
   [C5-120,[Andy L.,Uwe T.]],[C5-122,[Marc M.,Angi W.]],
   [C5-121,[Werner L.,Michael T.]],[C5-123,[Juergen L.,Harry C.]]]
```

B.5.2 Tracing the grouping of researchers

```
Activating task "group"
  input : plan element = researcher

Invoking inference Transform into possible groupings
  input : component_type "researcher" (a [concept(department_role)])
  output: groupings  < 105 groupings, not listed to save space (GS) >
  (a set(structure([instance(employee)])))

Invoking inference Select suitable groupings
  input : current groupings < 105 groupings, not listed to save space (GS) >
          (a set(structure([instance(employee)])))
  input : selection criterion "criterion(minimise, major_conflict)"
  (a one_of([major_conflict,major_synergy,minor_conflict,minor_synergy]))
  output: subset of groupings
"[[[Werner L.,Angi W.],[Marc M.,Harry C.],[Juergen L.,Michael T.],[Andy L.,Uwe T.]],
  [[Werner L.,Angi W.],[Marc M.,Juergen L.],[Harry C.,Michael T.],[Andy L.,Uwe T.]],
  [[Werner L.,Angi W.],[Marc M.,Michael T.],[Juergen L.,Harry C.],[Andy L.,Uwe T.]],
  [[Werner L.,Harry C.],[Marc M.,Angi W.],[Juergen L.,Michael T.],[Andy L.,Uwe T.]],
  [[Werner L.,Harry C.],[Marc M.,Juergen L.],[Angi W.,Michael T.],[Andy L.,Uwe T.]],
  [[Werner L.,Harry C.],[Marc M.,Michael T.],[Angi W.,Juergen L.],[Andy L.,Uwe T.]],
  [[Werner L.,Juergen L.],[Marc M.,Angi W.],[Harry C.,Michael T.],[Andy L.,Uwe T.]],
  [[Werner L.,Juergen L.],[Marc M.,Harry C.],[Angi W.,Michael T.],[Andy L.,Uwe T.]],
  [[Werner L.,Juergen L.],[Marc M.,Michael T.],[Angi W.,Harry C.],[Andy L.,Uwe T.]],
```

```
  [[Werner L.,Marc M.],[Angi W.,Harry C.],[Juergen L.,Michael T.],[Andy L.,Uwe T.]],
  [[Werner L.,Marc M.],[Angi W.,Juergen L.],[Harry C.,Michael T.],[Andy L.,Uwe T.]],
  [[Werner L.,Marc M.],[Angi W.,Michael T.],[Juergen L.,Harry C.],[Andy L.,Uwe T.]],
  [[Werner L.,Michael T.],[Marc M.,Angi W.],[Juergen L.,Harry C.],[Andy L.,Uwe T.]],
  [[Werner L.,Michael T.],[Marc M.,Harry C.],[Angi W.,Juergen L.],[Andy L.,Uwe T.]],
  [[Werner L.,Michael T.],[Marc M.,Juergen L.],[Angi W.,Harry C.],[Andy L.,Uwe T.]]]"
  (a set(structure([instance(employee)])))

Invoking inference Select suitable groupings
  input : current groupings < see output previous inference (GS) >
                            (a set(structure([instance(employee)])))
  input : selection criterion "criterion(maximise, major_synergy)"
  (a one_of([major_conflict,major_synergy,minor_conflict,minor_synergy]))
  output: subset of groupings
" [[[Werner L.,Harry C.],[Marc M.,Angi W.],[Juergen L.,Michael T.],[Andy L.,Uwe T.]],
  [[Werner L.,Harry C.],[Marc M.,Juergen L.],[Angi W.,Michael T.],[Andy L.,Uwe T.]],
  [[Werner L.,Harry C.],[Marc M.,Michael T.],[Angi W.,Juergen L.],[Andy L.,Uwe T.]],
  [[Werner L.,Juergen L.],[Marc M.,Harry C.],[Angi W.,Michael T.],[Andy L.,Uwe T.]],
  [[Werner L.,Juergen L.],[Marc M.,Michael T.],[Angi W.,Harry C.],[Andy L.,Uwe T.]],
  [[Werner L.,Marc M.],[Angi W.,Harry C.],[Juergen L.,Michael T.],[Andy L.,Uwe T.]],
  [[Werner L.,Marc M.],[Angi W.,Michael T.],[Juergen L.,Harry C.],[Andy L.,Uwe T.]],
  [[Werner L.,Michael T.],[Marc M.,Angi W.],[Juergen L.,Harry C.],[Andy L.,Uwe T.]],
  [[Werner L.,Michael T.],[Marc M.,Harry C.],[Angi W.,Juergen L.],[Andy L.,Uwe T.]],
  [[Werner L.,Michael T.],[Marc M.,Juergen L.],[Angi W.,Harry C.],[Andy L.,Uwe T.]]]"
  (a set(structure([instance(employee)])))

Invoking inference Select suitable groupings
  input : current groupings < see output previous inference (GS) >
                            (a set(structure([instance(employee)])))
  input : selection criterion "criterion(maximise, minor_synergy)"
   (a one_of([major_conflict,major_synergy,minor_conflict,minor_synergy]))
  output: subset of groupings
  [[Werner L.,Marc M.],[Angi W.,Michael T.],[Juergen L.,Harry C.],[Andy L.,Uwe T.]],
  [[Werner L.,Michael T.],[Marc M.,Angi W.],[Juergen L.,Harry C.],[Andy L.,Uwe T.]]]"
  (a set(structure([instance(employee)])))

Invoking inference Select suitable groupings
  input : current groupings < see output previous inference (GS) >
                            (a set(structure([instance(employee)])))
  input : selection criterion "criterion(minimise, minor_conflict)"
   (a one_of([major_conflict,major_synergy,minor_conflict,minor_synergy]))
  output: subset of groupings
  [[Werner L.,Marc M.],[Angi W.,Michael T.],[Juergen L.,Harry C.],[Andy L.,Uwe T.]],
  [[Werner L.,Michael T.],[Marc M.,Angi W.],[Juergen L.,Harry C.],[Andy L.,Uwe T.]]]"
  (a set(structure([instance(employee)])))

Task "group" terminated
  output: groupings =
  [[Werner L.,Marc M.],[Angi W.,Michael T.],[Juergen L.,Harry C.],[Andy L.,Uwe T.]],
  [[Werner L.,Michael T.],[Marc M.,Angi W.],[Juergen L.,Harry C.],[Andy L.,Uwe T.]]]"
```

B.5.3 Tracing the room selection inferencing

```
Activating task "assign"
  input : plan element = manager
  input : unit = Eva I.
```

```
  input : resources = [C5-113,C5-114,C5-115,C5-116,C5-118,
                       C5-120,C5-121,C5-122,C5-123]
  input : allocations = [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]]]

Invoking inference Select on resource requirements
  input :  component_type "manager" (a [concept(department_role)])
  input :  available resources "[C5-113,C5-114,C5-115,C5-116,C5-118,
                                 C5-120,C5-121,C5-122,C5-123]"
                               (a set([instance(room)]))
  output:  suitable resources "[C5-113,C5-114,C5-115,C5-116]"
                               (a set([instance(room)]))

Invoking inference Select on positional requirements
  input :  component_type "manager" (a [concept(department_role)])
  input :  available resources "[C5-113,C5-114,C5-115,C5-116]"
                               (a set([instance(room)]))
  input :  current positions "[[C5-117,Thomas D.],
                              [C5-119,[Monika X.,Ulrike U.]]]"
               (a structure( [instance (room)], list([instance(employee)])))
  output:  suitable resources " [C5-116]" (a set([instance(room)]))

Working memory operation "select" on "suitable resources"
  with result: "C5-116"
Working memory operation "add [[C5-116,Eva I.]]" on "allocations"
Working memory operation "subtract C5-116" on "resources"

Task "assign" terminated
  output: allocations = [[C5-117,Thomas D.],
                         [C5-119,[Monika X.,Ulrike U.]],
                         [C5-116,Eva I.]]
```

Appendix C

Example Implementation: Abstract & Specify

In this appendix only the application-specifc modules are listed. The generic modules can be found in Appendix B.

C.1 Application-specific modules

Task declarations

```
:- module('task-declarations.pl',
  [ task/1
  , task_input/2
  , task_output/2
  , control_term/2
  , task_structure/2
  , data_type/2
  ]).

:- discontiguous
    task/1,
    task_input/2,
    task_output/2,
    control_term/2,
    task_structure/2.

%
% Task knowledge
%
% task(        Task name).
% task_input(  Task name, Input name).
% task_output( Task name, Output name).
% control_term(Task name, Term name).
%
% task_structure(Task name, Procedure).

task(          'Abstract new evidence').
task_input(    'Abstract new evidence',    'new evidence').
task_output(   'Abstract new evidence',    'new findings').
```

```
control_term(  'Abstract new evidence',    'focus set').

task_structure('Abstract new evidence',
  ( data_operation(store, 'focus set', 'new evidence'),
    repeat((
      data_operation(select, 'focus set', Focus),
      exec_inference(abstract, [Focus], NewFinding),
      data_operation(add, 'focus set', NewFinding),
      data_operation(add, 'new findings', NewFinding)),
    until(data_operation(empty, 'focus set', true)))
  )).

task(          'Clarify finding').
task_input(    'Clarify finding',  'finding').
task_output(   'Clarify finding',  'new observation').
task_structure('Clarify finding',
  ( exec_inference(specify, ['finding'], Observable)
  , transfer_task(obtain, Observable, Observation)
  , data_operation(store, 'new observation', Observation)
  )).

data_type('new evidence',       set).
data_type('new findings',       set).
data_type('focus set',          set).
data_type('finding',            element).
data_type('new observation',    element).
```

Inference declarations

```
:- module('inference-declarations.pl',
  [ inference/2
  , metaclass/4
  , domain_view/2
  ]).

:- discontiguous
    inference/2,
    metaclass/4,
    domain_view/2.

% inference(Internal name, External name)
% metaclass(Inference,     Input/Output,        General name, Specialised name).
% domain_view(Inference, , Inference knowledge).

inference(  abstract,  'Abstract').
metaclass(  abstract,  input(1),        finding,     'Specific finding').
metaclass(  abstract,  output,                finding,     'General finding').
domain_view(abstract,  relation(abstraction, finding, finding)).

inference(  specify,   'Specify').
metaclass(  specify,   input(1),  finding,     'Finding to be clarified').
metaclass(  specify,   output,    observable,  'Dependent observable').
domain_view(specify,   relation(specification, finding, finding)).
```

Inference functions

```
:- module('inference-functions.pl', [ invoke_inference/3]).
```

```
:- ensure_loaded( ['inference-activation.pl'] ).

:- use_module(
  [ 'inference-methods.pl'
  , 'inference-declarations.pl'
  , 'domain-access.pl'
  ]).

inference_function(abstract, [In], Out) :-
    domain_retrieval(find_all, abstraction, Rules),
    rule_interpreter(Rules, In, Out, forward, single_pass, find_one).

inference_function(specify, [In], Out) :-
    domain_retrieval(find_all, specification, Rules),
    rule_interpreter(Rules, In, Out, backward, multi_pass, find_one).
```

Inference methods

```
:- module('inference-methods.pl',
  [ rule_interpreter/6
  ]).

%
% Computational techniques useed by inferences
%

rule_interpreter(Rules, In, Out, forward, single_pass, find_one) :-
    member([Premise, Conclusion], Rules),
    consistent(In, Premise),
    Out = Conclusion.
rule_interpreter(Rules, In, Out, backward, multi_pass, find_one) :-
    member([Premise, Conclusion], Rules),
    consistent(In, Conclusion),
    operand_of(Premise, Op),
    (   rule_interpreter(Rules, Premise, Out, backward, multi_pass, find_one)
    ;   Out = Op
    ).

consistent(X = Y, X = Y).
consistent(X = Y, X > Z) :-
    Y > Z.
consistent(X = Y, X >= Z) :-
    Y >= Z.
consistent(X = Y, X < Z) :-
    Y < Z.
consistent(X = Y, X =< Z) :-
    Y =< Z.

operand_of(X = _, X).
operand_of(X > _, X).
operand_of(X >= _, X).
operand_of(X < _, X).
operand_of(X =< _, X).
```

Domain index

```
:- module('domain-index.pl', [domain_index/3]).

% Inference -> domain mappings
%

domain_index(expression, finding,       [expr(patient_data)]).
domain_index(entity,     observable,    [property(patient_data)]).
domain_index(relation,   abstraction,   [relation(qual_abstraction),
                                         relation(definition)]).
domain_index(relation,   specification, [relation(qual_abstraction),
                                         relation(definition)]).
```

Domain theory

```
% Domain knowledge format
%
% concept(Concept name,     Supertyes)
% property(Concept,         Property name,           Valueset)
% relation(Relation name,   Type first argument,     Type second argument)
% tuple(Relation name,      [First argument,         Second argument])

concept(patient_data,       []).
concept(quantitative_data,  [patient_data]).
concept(qualitative_data,   [patient_data]).

property(quantitative_data, temperature,         numberrange(35.0, 42.0)).
property(quantitative_data, diastolic_pressure,  numberrange(0, 300)).
property(qualitative_data,  fever,               [present, absent]).
property(qualitative_data,  blood_pressure,      [normal, elevated]).
property(qualitative_data,  hypertension,        [present, absent]).

relation(qual_abstraction,  expr(quantitative_data), expr(qualitative_data)).
relation(definition,        expr(qualitative_data),  expr(qualitative_data)).

tuple(qual_abstraction, [temperature >= 38.0,      fever = present]).
tuple(qual_abstraction, [temperature < 38.0,       fever = absent]).
tuple(qual_abstraction, [diastolic_pressure >= 95, blood_pressure = elevated]).
tuple(qual_abstraction, [diastolic_pressure < 95,  blood_pressure = normal]).
tuple(definition,       [blood_pressure =elevated, hypertension = present]).
tuple(definition,       [blood_pressure = normal,  hypertension = absent]).
```

C.2 Example output

```
3 ?-  abstract_example(_).

Activating task "Abstract new evidence"
  input : new evidence = [temperature = 40,diastolic_pressure = 100]

Invoking inference Abstract
  input :  Specific finding "diastolic_pressure = 100" (a [expr(patient_data)])
  output:  General finding "blood_pressure = elevated" (a [expr(patient_data)])

Invoking inference Abstract
  input :  Specific finding "blood_pressure = elevated" (a [expr(patient_data)])
  output:  General finding "hypertension = present" (a [expr(patient_data)])
```

```
Invoking inference Abstract
  input :  Specific finding "temperature = 40" (a [expr(patient_data)])
  output:  General finding "fever = present" (a [expr(patient_data)])

Task "Abstract new evidence" terminated
  output: new findings = [blood_pressure = elevated,hypertension = present,fever = present]

Yes
4 ?- specify_example(_).

Activating task "Clarify finding"
  input : finding = hypertension = present

Invoking inference Specify
  input :  Finding to be clarified "hypertension = present" (a [expr(patient_data)])
  output:  Dependent observable "diastolic_pressure" (a [property(patient_data)])

 Please enter the value of diastolic_pressure: 100.

Task "Clarify finding" terminated
  output: new observation = diastolic_pressure = 100

Yes
```

Bibliography

Abiteboul, S. & Hull, R. (1987). IFO: A formal semantic dabase model. *ACM Trans. on Database Systems*, 12:525–565.

Akkermans, J. M., van Harmelen, F., Schreiber, A. T., & Wielinga, B. J. (1992). A formalisation of knowledge-level models for knowledge acquistion. *International Journal of Intelligent Systems*. forthcoming.

Alexander, J. H., Freiling, M. J., Shulman, S. J., Rehfuss, S., & Messick, S. L. (1988). Ontological analysis: an ongoing experiment. In Boose, J. & Gaines, B., editors, *Knowledge-Based Systems, Volume 2: Knowledge Acquisition Tools for Expert Systems*, pages 25–37. Academic Press, London.

Allemang, D. (1991). Sisyphus part 2: Generic tasks. In *Proceedings EKAW'91, Sisyphus Working Papers*.

Angele, J., Fensel, D., Landes, D., & Studer, R. (1991). KARL: An executable language for the conceptual model. In *Proceedings of the 6th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff*, pages 1.1–20, Canada. SRDG Publications, University of Calgary.

Anjewierden, A., editor (1991). *KEW Infrastructure Documentation*. Deliverable ESPRIT Project 2576 ACKnowledge. University of Amsterdam, SWI, Roetersstraat 15, 1018 WB Amsterdam. Available from: University of Amsterdam, Social Science Informatics, Roetersstraat 15, 1018 WB, The Netherlands.

Anjewierden, A., Wielemaker, J., & Toussaint, C. (1992). Shelley - computer aided knowledge engineering. *Knowledge Acquisition*, 4(1). Special issue:"The KADS approach to knowledge engineering".

Barthèlemy, S., Edin, G., Toutain, E., & Becker, S. (1987). Requirements Analysis in KBS Development. ESPRIT Project P1098 Deliverable D3 (task A2), Cap Sogeti Innovation.

Barthèlemy, S., Frot, P., & Simonin, N. (1988). Analysis document experiment F4. ESPRIT Project P1098, Deliverable E4.1, Cap Sogeti Innovation.

Bauer, C. & Karbach, W., editors (1992). *Proceedings Second KADS User Meeting*, ZFE BT SE 21, Otto-Hahn Ring 6, D-8000 Munich 83. Siemens AG.

Benjamins, R., Abu-Hanna, A., & Jansweijer, W. (1992a). Suitability criteria for model based diagnostic methods. In *In Proceedings of ECAI-workshop on Model Based Reasoning*, Vienna. SKBS/A2/92-13.

Benjamins, V. R., Jansweijer, W. N. H., & Abu-Hanna, A. (1992b). Integrating problem solving methods into KADS. In Bauer, C. & Karbach, W., editors, *Proccedings 2nd KADS User Meeting*, Munich. SKBS/A2/92-02.

Bennet, J. S. (1985). ROGET : A knowledge-based system for acquiring the conceptual structure of a diagnostic expert system. *Journal of Automated Reasoning*, 1:49–74.

Billault, J. P. (1989). Design and implementation of a configuration task. ESPRIT Project P1098, Deliverable E5.2, University of Amsterdam. Available from: University of Amsterdam, Social Science Informatics, Roetersstraat 15, 1018 WB, The Netherlands.

Boden, M. A. (1977). *Artificial Intelligence and Natural Man*. Basic Books, New York.

Boehm, B. W. (1988). A spiral model of software development and enhancement. *IEEE Computer*, pages 61–72.

Brachman, R. J. (1979). On the epistemological status of semantic networks. In Findler, N. V., editor, *Associative Networks*, New York. Academic Press.

Brachman, R. J. & Schmolze, J. G. (1985). An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9:171–216.

Brachman, R. J. & Smith, B. C. (1980). Special issue on knowledge representation. *SIGART Newsletter*, 70:1–138.

Bredeweg, B. & Wielinga, B. J. (1988). Integrating qualitative reasoning approaches. In *Proceedings of ECAI–88, Munich*, pages 195–201.

Breuker, J. A. & Wielinga, B. J. (1984). Techniques for knowledge elicitation and analysis. ESPRIT Project P12 Report 1.5, University of Amsterdam.

Breuker, J. A. & Wielinga, B. J. (1989). Model Driven Knowledge Acquisition. In Guida, P. & Tasso, G., editors, *Topics in the Design of Expert Systems*, pages 265–296, Amsterdam. North Holland.

Breuker, J. A., Wielinga, B. J., van Someren, M., de Hoog, R., Schreiber, A. T., de Greef, P., Bredeweg, B., Wielemaker, J., Billault, J. P., Davoodi, M., & Hayward, S. A. (1987). Model Driven Knowledge Acquisition: Interpretation Models. ESPRIT Project P1098 Deliverable D1 (task A1), University of Amsterdam and STL Ltd.

Brunet, E. & Toussaint, C. (1990). A KADS application in insurance. ESPRIT Project P1098, Deliverable E9.1, Cap Sesa Innovation.

Bylander, T. & Chandrasekaran, B. (1988). Generic tasks in knowledge-based reasoning: The right level of abstraction for knowledge acquisition. In Gaines, B. & Boose, J., editors, *Knowledge Acquisition for Knowledge Based Systems*, volume 1, pages 65–77. Academic Press, London.

Bylander, T. & Mittal, S. (1986). CSRL: A language for classificatory problem solving and uncertainty handling. *AI Magazine*, 7.

Chandrasekaran, B. (1988). Generic tasks as building blocks for knowledge-based systems: The diagnosis and routine design examples. *The Knowledge Engineering Review*, 3(3):183–210.

Chandrasekaran, B. (1990). Design problem solving: A task analysis. *AI Magazine*, 11:59–71.

Chen, P. P. S. (1976). The entity relationship model – toward a unified view of data. *ACM Transactions on Database Systems*, 1:9–36.

Clancey, W. J. (1983). The epistemology of a rule based system -a framework for explanation. *Artificial Intelligence*, 20:215–251.

Clancey, W. J. (1985a). Acquiring, representing and evaluating a competence model of diagnostic strategy. In Chi, Glaser, & Far, editors, *Contributions to the Nature of Expertise*.

Clancey, W. J. (1985b). Heuristic classification. *Artificial Intelligence*, 27:289–350.

Clancey, W. J. (1992). Model construction operators. *Artificial Intelligence*, 53(1):1–115.

Clancey, W. J. & Letsinger, R. (1984). NEOMYCIN: Reconfiguring a rulebased expert system for application to teaching. In Clancey, W. J. & Shortliffe, E. H., editors, *Readings in Medical Artificial Intelligence: the first decade*, pages 361–381. Addison-Wesley, Reading.

Coad, P. & Yourdon, E. (1991). *Object-Oriented Analysis*. Prentice Hall, Englewood Cliffs, New Jersey.

Console, L. & Torasso, P. (1990). Integrating models of the correct behaviour into abductive diagnosis. In Aiello, L. C., editor, *Proceedings ECAI–90, Stockholm*, pages 160–166, London. ECCAI, Pitman Publishing.

David, J. M. & Krivine, J. P. (1990). Explaining reasoning from knowledge level models. In Aiello, L., editor, *Proceedings ECAI'90, Stockholm*, pages 186–188, London. Pitman.

Davis, J. P. & Bonnel, R. P. (1990). Producing visually-based knowledge specifications for acquiring organizational knowledge. In Wielinga, B. J., Boose, J. H., Gaines, B. R., Schreiber, A. T., & van Someren, M. W., editors, *Current Trends in Knowledge Acquisition*, pages 105–122, Amsterdam. IOS Press.

Davis, R. (1980). Metarules: Reasoning about control. *Artificial Intelligence*, 15:179–222.

Davis, R. (1984). Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24:347–410.

de Greef, P. (1989). Cooperative statistical problem solving. In *Proceedings of the Second International Workshop on AI and Statistics, Fort Lauderdale, Florida.*

de Greef, P. & Breuker, J. A. (1985). A case study in structured knowledge acquisition. In *Proceedings of the 9th IJCAI*, pages 390–392, Los Angeles.

de Greef, P. & Breuker, J. A. (1989). A methodology for analysing modalities of system/user cooperation for KBS. In Boose, J., Gaines, B., & Ganascia, J. G., editors, *Proceedings European Knowledge Acquisiton Workshop, EKAW–89*, pages 462–473, Paris, France.

de Greef, P. & Breuker, J. A. (1992). Analysing system-user cooperation. *Knowledge Acquisition*, 4(1). Special issue 'The KADS approach to knowledge engineering'.

de Greef, P., Breuker, J. A., & de Jong, T. (1988a). Modality: An analysis of functions, user control and communication in knowledge-based systems. ESPRIT Project P1098, Deliverable D6 (task A4), University of Amsterdam. Available from: University of Amsterdam, Social Science Informatics, Roetersstraat 15, 1018 WB, The Netherlands.

de Greef, P., Breuker, J. A., Schreiber, A. T., & Wielemaker, J. (1988b). StatCons: Knowledge acquisition in a complex domain. In *Proceedings ECAI–88*, Munich.

de Greef, P., Schreiber, A. T., & Wielemaker, J. (1987). The StatCons case study. ESPRIT Project P1098, Deliverable E2.3 (experiment F2), University of Amsterdam. Available from: University of Amsterdam, Social Science Informatics, Roetersstraat 15, 1018 WB, The Netherlands.

de Hoog, R. (1989). Een expertsysteem, bijstand voor bijstand. *Informatie & Informatiebeleid*, 7(1):47–53. In Dutch.

de Hoog, R., Sommer, K., & Vogler, M. (1990). Designing knowledge-based systems: a study of organisational aspects. Technical Report Report W17, ISBN 90 346 2400 5, Dutch Organisation for Technological Aspects Research NOTA. In Dutch.

de Jong, T., de Hoog, R., & Schreiber, A. T. (1988). Knowledge acquisition for an integrated project mnagament system. *Information Processing and Management*, 24(6):681–691.

de Kleer, J. (1986). An assumption-based TMS. *Artificial Intelligence*, 28:127–162.

DeMarco, T. (1978). *Structured Analysis and System Specification.* Yourdon Press, New York.

DeMarco, T. (1982). *Controlling Software Projects.* Yourdon Press, New York.

Diaper, D., editor (1989). *Knowledge Elicitation: principles, techniques and applications.* Series in Expert Systems. Ellis Horwood Ltd., Chichester.

Eshelman, L. (1988). MOLE: A knowledge-acquisition tool for cover-and-differentiate systems. In Marcus, S., editor, *Automating Knowledge Acquisition for Expert Systems*, pages 37–80. Kluwer Academic Publishers, The Netherlands.

Eshelman, L., Ehret, D., McDermott, J., & Tan, M. (1988). MOLE: a tenacious knowledge acquisition tool. In Boose, J. H. & Gaines, B. R., editors, *Knowledge Based Systems, Volume 2: Knowledge Acquisition Tools for Expert Systems*, pages 95–108, London. Academic Press.

Fikes, R. & Kehler, T. (1985). The role of frame based representation in reasoning. *Communications of the ACM*, 28(9):904–920.

Genesereth, M. R. & Nilsson, N. J. (1987). *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann, Los Altos, California.

Goel, A., Soundarajan, N., & Chandrasekaran, B. (1987). Complexity in classificatory reasoning. In *AAAI–87*, pages 421–425.

Gruber, T. R. (1989). *The Acquisition of Strategic Knowledge.* Perspectives in Artificial Intelligence, Volume 4. Academic Press, San Diego.

Harel, D. (1987). State charts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274.

Harmon, P. (1991). A brief overview of software methodologies. *Intelligent Software Strategies*, VII(1):1–19. Newsletter. Circulation office: 37 Broadway, Arlington. MA 02174 USA.

Hayes-Roth, B. (1985). A blackboard architecture for control. *Artificial Intelligence*, 26(3):251–321.

Hayes-Roth, F., Waterman, D. A., & Lenat, D. B. (1983). *Building Expert Systems.* Addison-Wesley, New York.

Hayward, S. A. (1987). How to build knowledge systems; techniques, tools, and case studies. In *Proceedings of 4th annual ESPRIT conference*, pages 665–680, Amsterdam. North-Holland.

Hayward, S. A., Wielinga, B. J., & Breuker, J. A. (1987). Structured analysis of knowledge. *International Journal of Man-Machine Studies*, 26:487–498.

Hull, R. & King, R. (1987). Semantic database modelling: Survey, applications, and research issues. *ACM Computing Surveys*, 19:201–260.

Jansweijer, W. N. H. (1988). *PDP.* PhD thesis, University of Amsterdam.

Jansweijer, W. N. H., Elshout, J. J., & Wielinga, B. J. (1986). The expertise of novice problem solvers. In *Proceedings ECAI–86, Brigthon.*

Jansweijer, W. N. H., Elshout, J. J., & Wielinga, B. J. (1989). On the multiplicity of learning to solve problems. In Mandl, H., de Corte, E., Bennett, N., & Friedrich, H. F., editors, *Learning and Instruction: European research in an international context*, pages 127–145. Pergamon Press, Oxford, UK.

Karbach, W., Linster, M., & Voss, A. (1989). OFFICE-PLAN: Tackling the synthesis frontier. In Metzing, D., editor, *GWAI–89: 13th German Workshop on Artificial Intelligence, Informatik Fachberichte 216*, pages 379–387, Berlin. Springer Verlag.

Karbach, W., Linster, M., & Voss, A. (1990). Model-based approaches: One label - one idea? In Wielinga, B., Boose, J., Gaines, B., Schreiber, G., & van Someren, M., editors, *Current Trends in Knowledge Acquisition*, pages 173–189. IOS Press, Amsterdam.

Karbach, W., Tong, X., & Voss, A. (1988). Filling in the knowledge acquisition gap: via KADS models of expertise to ZDEST–2 expert systems. In *Proceedings of EKAW 88*, Bonn.

Karbach, W., Voss, A., Schukey, R., & Drouwen, U. (1991). Model-K: Prototyping at the knowledge level. In *Proceedings Expert Systems–91, Avignon, France*, pages 501–512.

Kerschberg, L., editor (1986). *Expert Database Systems.* The Benjamin/Cummings Publishing Company, Inc.

Killin, J. (1992). The management and maintenance of an operational KADS system. In Schreiber, A. T., Wielinga, B. J., & Breuker, J. A., editors, *KADS: A Principled Approach to Knowledge-Based System Development.* Academic Press, London. Forthcoming.

Klinker, G., Bhola, C., Dallemagne, G., Marques, D., & McDermott, J. (1991). Usable and reusable programming constructs. *Knowledge Acquisition*, 3:117–136.

Konolige, K. (1992). Abduction versus closure in causal theories. *AI Journal*, 53:255–272.

Koster, J. (1990). Rule modelling: A comparison of three data modelling techniques. Technical report, University of Amsterdam, Social Science Informatics.

Krickhahn, R., Nobis, R., Mahlmann, A., & Schachter, M. (1988). Applying the KADS methodology to develop a knowledge-based system. In *Proceedings ECAI–88, Munich*, pages 11–17, London. Pitman.

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). SOAR: an architecture for general intelligence. *Artificial Intelligence*, 33:1–64.

Lemmers, M. (1991). A shell for systematic diagnosis: Structure-preserving design of a KBS. Master's thesis, University of Amsterdam, Social Science Informatics.

Lenat, D. B. & Guha, R. V. (1990). *Building large knowledge-based systems. Representation and inference in the Cyc project.* Addison-Wesley, Reading Massachusetts.

Linster, M. (1992). *Knowledge acquisition based on explicit methods of problem–solving.* PhD thesis, University of Kaiserslautern.

Linster, M. & Musen, M. A. (1992). Use of KADS to create a conceptual model of the ONCOCIN task. *Knowledge Acquisition*, 4(1). Special issue:'The KADS approach to knowledge engineering'.

Maes, P. (1987). Computational reflection. Technical report 87-2, Free University of Brussels, AI Lab.

Marcus, S., editor (1988). *Automatic knowledge acquisition for expert systems.* Kluwer.

Marcus, S. & McDermott, J. (1989). SALT: A knowledge acquisition language for propose-and-revise systems. *Artificial Intelligence*, 39(1):1–38.

Mars, N. J. I. (1987). Onderzoek van niveau: Kennistechnologie in wording. Inaugurale rede, 17 september, Universiteit Twente.

McDermott, J. (1988). Preliminary steps towards a taxonomy of problem-solving methods. In Marcus, S., editor, *Automating Knowledge Acquisition for Expert Systems*, pages 225–255. Kluwer Academic Publishers, The Netherlands.

Meyer, M. A. & Booker, J. M. (1991). *Eliciting and Analyzing Expert Judgement: A Practical Guide*, volume 5 of *Knowledge-Based Systems.* Academic Press, London.

Morik, K. (1989). Sloppy modelling. In Morik, K., editor, *Knowledge Representation and Organisation in Machine Learning.* Springer Verlag.

Musen, M. A. (1989). *Automated Generation of Model-Based Knowledge-Acquisition Tools.* Pitman, London. Research Notes in Artificial Intelligence.

Musen, M. A., Fagan, L. M., Combs, D. M., & Shortliffe, E. H. (1987). Use of a domain-model to drive an interactive knowledge-editing tool. *International Journal of Man-Machine Studies*, 26:105–121.

Neale, I. M. (1988). First generation expert systems: a review of knowledge acquisition methodologies. *The Knowledge Engineering Review*, 3(2):105–145.

Neches, R., Swartout, W. R., & Moore, J. D. (1985). Enhanced maintenance and explanation of expert systems through explicit models of their development. *IEEE Trans. Softw. Eng.*, 11:1337–1351.

Newell, A. (1982). The knowledge level. *Artificial Intelligence*, 18:87–127.

Newell, A. & Simon, H. A. (1963). GPS – a program that simulates human thought. In *Computers and Thought*, pages 279–296. MacGraw-Hill, New York.

Nilsson, N. J. (1991). Logic and artificial intelligence. *Artificial Intelligence*, 47:31–56.

Patil, R. S. (1988). Artificial intelligence techniques for diagnostic reasoning in medicine. In Shobe, H. E. & AAAI, editors, *Exploring Artificial Intelligence: Survey Talks from the National Conferences on Artificial Intelligence*, pages 347–379. Morgan Kaufmann, San Mateo, California.

Pople, H. (1982). Heuristic methods for imposing structure on ill-structured problems: The structuring in medical diagnosis. In Szolovits, P., editor, *Artificial Intelligence in Medicine*, pages 119–190. Westview Press, Boulder CO.

Porter, D. (1992). An inference strcuture for use in the fraud assessment domain. In Bauer, C. & Karbach, W., editors, *Proceedings Second KADS User Meeting*, ZFE BT SE 21, Otto-Hahn Ring 6, D-8000 Munich 83. Siemens AG.

Puerta, A., Egar, J., Tu, S. W., & Musen, M. A. (1991). A multiple-method knowledge-acquisition shell for the automatic generation of knowledge-acquisition tools. In *Proceedings of the 6th Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff, Canada*, pages 20.1–19. SRDG Publications, University of Calgary.

Puppe, F. (1990). *Problemlösungsmethoden in Expertensystemen.* Studienreihe Informatik. Springer Verlag.

Readdie, M. & Innes, N. (1987). Network management: Requirements analysis and feasibility analysis. ESPRIT Project P1098, Deliverable E3.1a, SciCon Ltd. (UK).

Reichgelt, H. & van Harmelen, F. (1986). Criteria for choosing represenation languages and control regimes for expert systems. *Knowledge Engineering Review*, 1:2–17.

Reinders, M., Vinkhuyzen, E., Voss, A., Akkermans, J. M., Balder, J. R., Bartsch-Spörl, B., Bredeweg, B., Drouven, U., van Harmelen, F., Karbach, W., Karssen, Z., Schreiber, A. T., & Wielinga, B. J. (1991). A conceptual modelling framework for knowledge-level reflection. *AI Communications*, 4(2-3):74–87.

Roth, E. M. & Woods, D. D. (1989). Cognitive task analysis: An approach to knowledge acquisition for intelligent system design. In Guida, P. & Tasso, G., editors, *Topics in Expert System Design*, pages 233–264, Amsterdam. North Holland.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. (1991). *Object-Oriented Modelling and Design*. Prentice Hall, Englewood Cliffs, New Jersey.

Russell, B. (1961). *History of Western Philosophy – and its connection with political and social circumstances from the earliest times to the present day*. Allen & Urwin, London.

Schachter, M. & Wermser, D. (1988). A sales assistant for chemical measurement equipment. In *Proceedings ECAI–88, Munich*, pages 191–193, London. Pitman.

Schank, R. (1975). *Conceptual information processing*. North Holland, Amsterdam.

Schoenmakers, E. T. M. (1992). Control of a blackboard system for acoustic analysis. Master's thesis, Technical University of Delft, Faculty of Electrical Engineering, Delft, The Netherlands. In Dutch.

Schreiber, A. T. (1992). The KADS approach to knowledge engineering. editorial special issue. *Knowledge Acquisition*, 4(1):1–4.

Schreiber, A. T., Akkermans, J. M., & Wielinga, B. J. (1991a). On problems with the knowledge level perspective. In Steels, L. & Smith, B., editors, *AISB–91: Artificial Intelligence and Simulation of behaviour*, pages 208–221, London. Springer Verlag. Also in: *Proceedings Banff–90 Knowledge Acquisition Workshop*, J. H. Boose and B. R. Gaines (editors), SRDG Publications, University of Calgary, pages 30-1 – 30-14.

Schreiber, A. T., Bartsch-Spörl, B., Bredeweg, B., van Harmelen, F., Karbach, W., Reinders, M., Vinkhuyzen, E., & Voss, A. (1991b). Designing architectures for knowledge-level reflection. ESPRIT Basic Research Action P3178 REFLECT, Deliverable IR.4 RFL/UvA/III.1/4, REFLECT Consortium. Available from: University of Amsterdam, Social Science Informatics, Roetersstraat 15, 1018 WB, The Netherlands.

Schreiber, A. T., Bredeweg, B., Davoodi, M., & Wielinga, B. J. (1987). Toward a design methodology for KBS. ESPRIT Project P1098, Deliverable D8 Uva/Stc-B2-Pr–001, Vf Memo 97, SWI, University Of Amsterdam. Available from: University of Amsterdam, Social Science Informatics, Roetersstraat 15, 1018 WB, The Netherlands.

Schreiber, A. T., Bredeweg, B., de Greef, P., Terpstra, P., Wielinga, B. J., Brunet, E., Simonin, N., & Wallyn, A. (1989a). A KADS approach to KBS design. ESPRIT Project 1098, deliverable B6 UvA-B6-PR–010, University of Amsterdam & Cap Sogeti Innovation. Available from: University of Amsterdam, Social Science Informatics, Roetersstraat 15, 1018 WB, The Netherlands.

Schreiber, A. T., Breuker, J. A., Bredeweg, B., & Wielinga, B. J. (1988). Modelling in KBS development. In *Proc. 2th European Knowledge Acquisition Workshop, Bonn, GMD-Studien 143*, pages 7.1– 7.15, St. Augustin. GMD. Also in: Proc. 8th Expert Systems Workshop, Avignon, 1988.

Schreiber, A. T., Wielinga, B. J., Hesketh, P., & Lewis, A. (1989b). A KADS design description language. ESPRIT Project 1098, deliverable B7 UvA-B7-PR–007, University of Amsterdam & STC Technology Ltd. Available from: University of Amsterdam, Social Science Informatics, Roetersstraat 15, 1018 WB, The Netherlands.

Schrijnen, L. M. & Wagenaar, G. (1988). Autopes: the development of an expert system for process control. In van Someren, M. W. & Schreiber, A. T., editors, *Proceedings First Dutch AI Conference NAIC–88*, pages 58–71, University of Amsterdam. Department of Social Science Informatics. In Dutch.

Shadbolt, N. & Wielinga, B. J. (1990). Knowledge based knowledge acquisition: the next generation of support tools. In Wielinga, B. J., Boose, J., Gaines, B., Schreiber, G., & van Someren, M. W., editors, *Current Trends in Knowledge Acquisition*, pages 313–338, Amsterdam. IOS Press.

Shortliffe, E. H., Scott, A. C., Bischoff, M. B., Cambell, A. B., van Melle, W., & Jacobs, C. D. (1981). ONCOCIN: An expert system for oncology protocol management. In

IJCAI–81, pages 876–881.

Smith, B. C. (1985). Prologue to "reflection and semantics in a procedural language". In Brachman, R. J. & Levesque, H. J., editors, *Readings in Knowledge Representation*, pages 31–40. Morgan Kaufman, California.

Sowa, J. F. (1984). *Conceptual Structures*. Addison-Wesley.

Sprenger, M. (1991). Explanation strategies for KADS-based expert systems. DIAMOD Project Bericht Nr. 10, GMD, St. Augustin, Germany.

Steels, L. (1990). Components of expertise. *AI Magazine*. Also as: AI Memo 88-16, AI Lab, Free University of Brussels.

Sticklen, J. (1989). Problem solving architecture at the knowledge level. *Journal of Experimental and Theoretical Artificial Intelligence*.

Taylor, R., Porter, D., Hickman, F., Streng, K.-H., Tansley, S., & Dorbes, G. (1989). System evolution - principles and methods (the life-cycle model). ESPRIT Project P1098, Deliverable Task G9, Touche Ross.

Tong, X., He, Z., & Yu, R. (1988). A survey of the expert system tool ZDEST–2. In *Proceedings ECAI–88, Munich*, pages 113–118, London. Pitman.

Ueberreiter, B. & Voss, A., editors (1991). *Materials KADS User Meeting, Munich, February 14/15 1991*. Siemens AG ZFE IS INF 32, Munich Perlach. In German.

van der Molen, R. & Kruizinga, E. P. (1990). OKS GAK: a feasibility study. Master's thesis, University of Amstrerdam, Department of Social Science Informatics. In Dutch.

van der Spek, R., van der Wouden, H., & Ysbrandy, C. (1990). The paint advisor. *Expert Systems*, 7(4):190–198.

van Harmelen, F. (1989). Classification of meta-level architectures. In Jackson, P., Reichgelt, H., & van Harmelen, F., editors, *Logic-Based Knowledge Representation*, chapter 2, pages 13–36. MIT Press. Also in: *Meta-Programming in Logic Programming (META88)*, Abramson, H. and Rogers, M. H. (eds.), MIT Press, 1989, pp. 103-122.

van Harmelen, F., Akkermans, J. M., Balder, J. R., Schreiber, A. T., & Wielinga, B. J. (1990). Formal specifications of knowledge models. ESPRIT Basic Research Action P3178 REFLECT, Deliverable R.1 RFL/ECN/I.4/1, Netherlands Energy Research Foundation ECN. Available from: University of Amsterdam, Social Science Informatics, Roetersstraat 15, 1018 WB, The Netherlands.

van Harmelen, F., Akkermans, J. M., Bartsch-Spörl, B., Bredeweg, B., Coulon, C. H., Drouven, U., Karbach, W., Reinders, M., Schreiber, A. T., Vinkhuyzen, E., Voss, A., & Wielinga, B. J. (1992). Knowledge-level reflection: Specifications and architectures. ESPRIT Basic Research Action P3178 REFLECT, Deliverable R.2 RFL/UvA/III.2, REFLECT Consortium. Available from: University of Amsterdam, Social Science Informatics, Roetersstraat 15, 1018 WB, The Netherlands.

van Harmelen, F. & Balder, J. R. (1992). $(ML)^2$: a formal language for KADS models of expertise. *Knowledge Acquisition*, 4(1). Special issue: 'The KADS approach to knowledge engineering'.

van Heijst, G., Terpstra, P., Wielinga, B. J., & Shadbolt, N. (1992). Using generalised directive models in knowledge acquisition. In Wetter, T., Althoff, K., Boose, J., Gaines, B., Linster, M., & Schmalhofer, F., editors, *Current Developments in Knowledge Acquisition: EKAW-92*, Berlin/Heidelberg. Springer-Verlag.

van Langevelde, I. A., Philipsen, A. W., & Treur, J. (1992). Formal specification of compositional architectures. In Neumann, B., editor, *Proceedings ECAI'92, Vienna*, pages 272–276, Chichester. Wiley. To appear. Longer version available as: Report IR-282, Mathematics and Computer Science, Free University of Amsterdam.

van Melle, W., Scott, A. C., Bennet, J. S., & Peairs, M. A. S. (1981). *The EMYCIN Manual*.

Vanwelkenhuysen, J. & Rademakers, P. (1990). Mapping knowledge-level analysis onto a computational framework. In Aiello, L., editor, *Proceedings ECAI–90, Stockholm*, pages

681–686, London. Pitman.

Voss, A., Karbach, W., Drouven, U., & Lorek, D. (1990). Competence assessment in configuration tasks. In Aiello, L., editor, *Proceedings of the 9th European Conference on AI, ECAI–90*, pages 676–681, London. ECAI, Pitman.

Wetter, T. (1990). First-order logic foundation of the KADS conceptual model. In Wielinga, B., Boose, J., Gaines, B., Schreiber, G., & van Someren, M., editors, *Current trends in knowledge acquisition*, pages 356–375, Amsterdam. IOS Press.

Wielemaker, J. (1991). *SWI-Prolog 1.5: Reference Manual.* University of Amsterdam, Social Science Informatics, Roetersstraat 15, 10-18 WB Amsterdam, The Netherlands. E-mail: jan@swi.psy.uva.nl.

Wielemaker, J. & Billault, J. P. (1988). A KADS analysis for configuration. ESPRIT Project P1098, Deliverable E5.1 Uva-F5-PR–001, SWI, University of Amsterdam. Available from: University of Amsterdam, Social Science Informatics, Roetersstraat 15, 1018 WB, The Netherlands.

Wielinga, B. J., Akkermans, J. M., Schreiber, A. T., & Balder, J. R. (1989). A knowledge acquisition perspective on knowledge-level models. In Boose, J. H. & Gaines, B. R., editors, *Proceedings Knowledge Acquisition Workshop KAW–89, Banff*, pages 36–1 – 36–22, University of Calgary. SRDG Publications.

Wielinga, B. J. & Bredeweg, B. (1988). Knowledge and expertise in expert systems. In van der Veer, G. C. & Mulder, G., editors, *Human-Computer Interaction: Psychonomics Aspects*, pages 290–297, Berlin. Springer-Verlag.

Wielinga, B. J. & Breuker, J. A. (1984). Interpretation of verbal data for knowledge acquisition. In OShea, T., editor, *Advances in Artificial Intelligence*, pages 41–50, Amsterdam. ECAI, Elsevier Science publishers. Also as: Report 1.4, ESPRIT Project 12, University of Amsterdam.

Wielinga, B. J. & Breuker, J. A. (1986). Models of expertise. In *Proceedings ECAI–86*, pages 306–318.

Wielinga, B. J., Schreiber, A. T., & Breuker, J. A. (1992a). KADS: A modelling approach to knowledge engineering. *Knowledge Acquisition*, 4(1). Special issue 'The KADS approach to knowledge engineering'.

Wielinga, B. J., van de Velde, W., Schreiber, A. T., & Akkermans, J. M. (1992b). Towards a unification of knowledge modelling approaches. In David, J.-M., Krivine, J.-P., & Simmons, R., editors, *Second Generation Expert Systems*. Springer-Verlag. To appear. Also as: Technical Report ESPRIT Project P5248, KADS-II/T1.1/TR/UvA/004/3.0.

Winkels, R. G. F., Achthoven, W., & van Gennip, A. (1989). Methodology and modularity in ITS design. In *Artificial Intelligence and Education*, pages 314–322, Amsterdam. IOS Press.

Wright, I., Hayball, C., Land, L., & Mulhall, T. (1988). Analysis report experiment F6. ESPRIT Project P1098, Deliverable E6.1, STC Technology Ltd. & Knowledge Based Systems Centre.

Yourdon, E. (1989a). *Managing the Structured Techniques.* Yourdon Press, Englewood Cliffs, New Jersey.

Yourdon, E. (1989b). *Modern Structured Analysis.* Prentice Hall, Englewood Cliffs, New Jersey.

Samenvatting

De vraag of het mogelijk is intelligente, "denkende" machines te bouwen wordt tegenwoordig veelvuldig gesteld. Het onderzoek naar *kunstmatige intelligentie* (artificial intelligence ofwel AI) houdt zich bezig met deze vraagstelling. In de AI gaat men ervan uit dat het mogelijk is intelligent gedrag te simuleren middels formele manipulaties van symbolen in een computerprogramma. Een bekend voorbeeld van AI systemen zijn de zgn. kennissystemen, soms ook wel 'expertsystemen" genoemd. Een kennissysteem is een computerprogramma dat in staat is om bepaalde probleemoplostaken uit te voeren, zoals het stellen van een diagnose bij een patïent of het ontwerpen van een apparaat. Een kennissysteem bestaat uit een kennisbank, die een of andere expliciete symbolische representatie van kennis in een bepaald domein (bv. ischaemische hartziekten) bevat, en uit redeneermechanismen die deze kennis gebruiken om een bepaald probleem op te lossen.

De eerste kennissystemen, die in de jaren zeventig ontwikkeld werden, hadden een simpele organisatiestructuur. Het systeem bestond uit een verzameling van brokjes kennis in eenzelfde formaat (meestal zgn. ALS/DAN regels) en één standaard redeneermechanisme. Aan deze aanpak bleken echter een aantal fundamentele problemen te kleven. Zo was het moeilijk, zo niet onmogelijk, om de benodigde kennis te vergaren, omdat er een wereld van verschil bestaat tussen de kennis zoals een expert die (verbaal of schriftelijk) uit en de voorgeschreven representatie van kennis in een systeem. Ook het onderhoud van de kennisbank en het geven van adequate uitleg over het gevolgde redeneerspoor bleek problematisch.

Deze beperkingen van de eerste kennissystemen waren in feite een uiting van een breder probleem van het AI onderzoek in de jaren zeventig. Men hield zich voornamelijk bezig met *hoe* men kennis kan representeren in een symbolische vorm, zonder zich af te vragen *wat* men nu eigenlijk wilde representeren en waarom. Als antwoord hierop formuleerde Newell (1982) zijn *kennisnivo-hypothese* ("knowledge-level hypothesis"). Het kennisnivo beschrijft de rationaliteit van een systeem (de "agent") in termen van doelen, acties en kennispartities, *onafhankelijk* van de feitelijke realisatie in bv. ALS/DAN regels of logica (het symbolische nivo ofwel 'symbol level").

Kennisnivo-beschrijvingen spelen tegenwoordig een belangrijke rol bij het ontwikkelen van kennissystemen. Het centrale thema van dit onderzoek is de vraag hoe Newell's idee van een kennisnivo op een principïele manier toegepast kan worden in dit ontwikkelproces. Centraal daarin staat het "kennismodel". Een kennismodel beschrijft de organisatie van kennis en de rol die de verschillende kenniselementen spelen tijdens het probleemoplosproces, in een implementatie-onafhankelijk en voor mensen begrijpelijk vocabulair. Kennismodellen kunnen gezien worden als een concretisering van Newell's conceptie. Uitgangspunt van dit onderzoek is dat de KADS modellen van expertise zoals beschreven door Wielinga en Breuker (1986) beschouwd kunnen worden als dergelijke kennismodellen.

De rol van kennisnivo-beschrijvingen bij het bouwen van kennissystemen is niet onomstreden. Als punten van kritiek worden onder meer genoemd de onmogelijkheid om controle-kennis te beschrijven, de mogelijke computationele inadequaatheid, het ontbreken van voorschriften voor ontwerp en implementatie en het feit dat de modellen geen voorspellingen kunnen genereren over het verwachte gedrag van het systeem. In Hoofdstuk 2 worden deze kritiekpunten besproken en verworpen. Zo wordt beargumenteerd dat kennismodellen *noodzakelijk* ondergespecificeerd zijn t.a.v. het te bouwen systeem. Ook

biedt de uitgebreide typering in het kennismodel voldoende garanties m.b.t. de computationele adequaatheid. Tevens wordt duidelijk gemaakt dat een kennismodel gebruikt kan worden om relevante voorspellingen te doen over het verwachte gedrag van het systeem en dus in zekere zin gezien kunnen worden als de theorie die aan een systeem ten grondslag ligt.

Hoofdstuk 3 geeft een overzicht van de aard van de kennismodellen zoals die in KADS gebruikt worden. KADS kennismodellen bestaan uit een viertal categorïen van kennis, die gezien kunnen worden als kennis*lagen* met onderling een beperkte interactie. De domeinlaag bevat een declaratieve beschrijving van alle domeinspecifieke kennis voor een bepaalde applicatie, zoals de concepten, attributen, relaties en structuren die onderscheiden worden. De inferentielaag beschrijft alle basale redeneerstappen die men wil maken tijdens het probleemoplossen. Elke redeneerstap wordt beschreven in een domeinonafhankelijk vocabulair, waarbij voor elke inferentieterm, ook wel "rol" genoemd, wordt aangegeven welke delen van de domeinkennis deze rol kunnen vervullen. Inferentiekennis kan grafisch worden gerepresenteerd in een zgn. *inferentiestructuur*. Deze inferentiestructuur beschrijft de afhankelijkheden tussen de verschillende basale redeneerstappen. De taaklaag beschrijft vervolgens hoe redeneerstappen dynamisch gecombineerd kunnen worden om bepaalde probleemoplostaken of -subtaken uit te voeren. Deze taakprocedures kunnen gezien worden als standaard-strategïen om een (sub-)probleem op te lossen. De strategielaag tenslotte beschrijft hoe het systeem eventueel toch tot een oplossing kan komen in het geval dat de standaard-strategïen zoals beschreven op de taaklaag falen.

Een belangrijk kenmerk van de KADS kennismodellen is dat de beschrijving van het probleemoplosproces, op de domeinkennis na, terminologie gebruikt die *onafhankelijk* is van het specifieke domein. Dit opent de mogelijkheid om delen van een kennismodel te hergebruiken voor andere domeinen, waarin een vergelijkbare taak moet worden uitgevoerd. Dergelijke partïele, generieke kennismodellen, die typisch bestaan uit en beschrijving van inferentiekennis en taakkennis, beschrijven in feite een bepaalde methode om een probleem zoals diagnose op te lossen. Deze modellen worden wel interpretatiemodellen genoemd, omdat deze veel gebruikt worden om het probleemoplosgedrag van een domein-expert te interpreteren. Herbruikbare elementen van kennismodellen zorgen ervoor dat een kennisingenieur (een term, die gebruikt wordt om degenen, die kennissystemen ontwikkellen, aan te duiden) niet steeds opnieuw het wiel hoeft uit te vinden.

In Hoofdstuk 4, 5 en 6 wordt nader ingegaan op een aantal meer gedetailleerde onderwerpen met betrekking tot kennismodellen. Hoofdstuk 4 beschrijft een modelleertaal voor het specificeren van de structuur van de domeinkennis. In een kennisnivo-analyse van de domein-specifieke kennis is men met name geïnteresserd in een schematische beschijving van de structuur van de kennis: het "domein schema". De invulling van de kennis in deze structuur kan dan in een latere verfijningsfase geschieden. In de literatuur wordt voor dit soort beschrijvingen gewoonlijk ófwel een specifieke kennisrepresentatietaal ófwel een conventionele datamodelleertaal gebruikt. De eerste oplossing is sub-optimaal, omdat het vereist dat men zich vastlegt op een bepaalde symbolische representatie. De tweede oplossing is evenmin bevredigend, omdat kennissystemen een aantal specifieke eisen stellen, waarvoor deze conventionele talen geen oplossing bieden. Een belangrijke eis is bijvoorbeeld dat de modelleertaal primitieven bevat om de structuur van een groep ALS/DAN regels te beschrijven. De domeinmodelleertaal, die in Hoofdstuk 4 beschreven wordt, is gebaseerd op bestaande modelleertalen, maar bevat een aantal additionele primitieven, die

voor kennissystemen noodzakelijk zijn. We laten zien dat deze taal gezien kan worden als een generalisatie van een aantal "symbol-level" kennisrepresentatietalen, hetgeen precies is wat men zou willen van een kennisnivo-beschrijving van domeinkennnis. Een aantal aspecten van de formele semantiek van deze taal moeten echter nog verder uitgezocht worden.

Hoofdstuk 5 bevat een conceptuele beschrijving van het specificeren van inferentiekennis in kennismodellen. Zoals reeds opgemerkt, is de inferentiekennis het eerste nivo waarop men abstraheert van het applicatie-domein en vervult dus een cruciale rol in het hergebruik van modellen. Een interpretatiemodel verschaft vaak wel een eerste versie van een inferentiestructuur, maar meestal moet deze enigzins aangepast worden om geschikt te zijn voor een nieuw domein. Een aantal technieken en methoden worden beschreven, die gebruikt kunnen worden tijdens dit proces, zoals kennisdifferentiatie en taakdecompositie. Ook wordt een voorbeeld gegeven van een volledige "top-down" specificatie van een inferentiestructuur, waarbij een eenvoudig initïeel hypothetico-deductief model stap-voor-stap verfijnd wordt. Dit soort verfijning kan ondersteund worden door generieke modelcomponenten van een kleiner omvang dan complete interpretatiemodellen. Dergelijke generieke componenten worden voor het voorbeeld-model beschreven.

Als eenmaal een kennismodel gebouwd is, rijst de vraag hoe men op basis hiervan een systeem kan construeren. Men staat dan feitelijk voor het probleem een adequate symbolische realisatie te vinden van de elementen van het kennismodel door het kiezen van geschikte computationele en representatie-technieken. Dit *operationalisatie-proces* moet ervoor zorgdragen dat het uiteindelijke systeem voldoet aan een aantal voorwaarden met betrekking tot onderhoud, aanpasbaarheid, herbruikbaarheid en uitlegfaciliteiten. In Hoofdstuk 6 worden de verschillende stappen en beslissingen in het ontwerp- en implementatieproces besproken. De notie van *structuur-behoudend ontwerp* wordt geïntroduceerd als een centraal principe voor de transformatie van een kennisnivo-beschrijving naar een beschrijving op symbolisch nivo. Met structuur-behoud wordt hier bedoeld dat zowel de inhoud als de structuur van informatie in een kennismodel bewaard blijven in het uiteindelijke artefact. Met behulp van dit principe kan men een prototypische systeem-architectuur specificeren die logisch volgt, maar niet identiek is aan, de structuur van het kennismodel. Deze architectuur wordt geïllustreerd middels een voorbeeld.

Ook wordt de ondersteuning, die gegeven kan worden d.m.v. gespecialiseerde programmeeromgevingen, besproken. Daarbij wordt geconstateerd dat de meeste van deze omgevingen ófwel de kennisingenieur teveel inperken in zijn/haar mogelijkheden om het gewenste systeem te realiseren, ófwel te weinig ondersteuning bieden. Een alternatieve aanpak wordt voorgesteld, geïllustreerd door een prototype-omgeving, die gezien kan worden als een poging om de kennisingenieur zowel flexibiliteit als maximale ondersteuning te bieden d.m.v. kleine herbruikbare code-modules,

Hoofdstuk 7 beschrijft een applicatie van de beschreven technieken in een domein waarin kamers aan werknemers moeten worden toegewezen. Dit domein is gebruikt als voorbeeld-domein in het "Sisyphus" project, met als doelstelling de verschillende aanpakken voor kennismodellering te vergelijken. Dit hoofdstuk bevat een gedetailleerd voorbeeld van een KADS kennismodel en laat zien hoe de ontwerpprincipes beschreven in Hoofdstuk 6 gebruikt kunnen worden om op basis van dit model een systeem te bouwen. In Appendices A en B zijn respectievelijk de beschrijving van de structuur van de domeinkennis en de uiteindelijke systeemcode te vinden.

In Hoofdstuk 8 wordt de KADS aanpak vergeleken met twee vooraanstaande methodieken voor het ontwikkelen van conventionele systemen, te weten de functionele aanpak van Yourdon en de object-georïenteerde aanpak. Dit soort vergelijkingen zijn belangrijk, omdat kennissystemen meestal niet geïsoleerd worden toegepast, maar in combinatie met meer conventionele applicaties. De verschillende aanpakken worden vergeleken door aan te geven welke modelleerprimitieven elke aanpak biedt voor het beschrijven van drie gezichtspunten die men kan innemen op een systeem, te weten het funtionele, het data- en het dynamische (of controle-) gezichtspunt. Ondanks de verschillen in terminologie blijken er veel overeenkomsten te zijn. De gevonden overeenkomsten en verschillen worden besproken.

Er zijn momenteel een aantal verschillende methoden voor kennismodellering. Er bestaat dan ook een groeiende behoefte om te komen tot een unificatie van de verschillende aanpakken, zodat modellen onderling uitgewisseld kunnen worden. Daarvoor zijn allereerst vergelijkende studies nodig. Hoofdstuk 9 beschrijft een dergelijke studie. Hierin wordt een kennismodel geconstrueerd van de "cover & differentiate" probleemoplosmethode. Dit model wordt vervolgens vergeleken met het heuristische-classificatie model. Hierbij komen een aantal verschillen aan het licht, die bij een informele of computationele beschrijving gemakkelijk over het hoofd gezien kunnen worden. De modelleertaal, die in dit hoofdstuk gebruikt wordt, is een vroege versie van een formele taal voor de representatie van KADS modellen.

Hoofdstuk 10 tenslotte vat de conclusies van dit onderzoek samen.

Curriculum Vitae

August Theodoor (Guus) Schreiber was born on June 24 1956 in Heerlen. He went to grammar school at the Bischoppelijk College in Roermond, In 1974, he took up a study in medicine at the University of Utrecht. During his studies he worked as a tutor on courses for interviewing techniques, as an information officer for the Faculty of Medicine and as a volunteer for the alternative health care organisation Release Utrecht. In 1983 he received a M.Sc. in medicine and followed a post-graduate programme in computer science. From 1984 to 1986 he was with the Medical Informatics Unit of the University of Leiden, where he worked on the development of knowledge-based systems for medical decision support. Since 1986, he is with the Department of Social Science Informatics of the University of Amsterdam. He worked as a research scientist and project manager on a number of EC-sponsored research projects in the area of methodologies for knowledge-based system development. He is currently project manager for UvA on an EC-project for medical knowledge-based systems and works also on the KADS-II project in which a European standard for knowledge-based systems is developed. In 1988 he organised together with Maarten van Someren the first Dutch AI Conference (NAIC'88). In 1990 he was co-organiser of the European Knowledge Acquisition Workshop (EKAW'90).