AI3/MSc

Knowledge Engineering

Spring 2003

Qiang Shen[1] and Alan Smaill[2]

Division of informatics

[1]Email: Q.Shen@ed.ac.uk
[2]Email: A.Smaill@ed.ac.uk

Contents

1 **Introduction** 2

2 **Knowledge Systems and Human Interface** 6

3 **Knowledge Acquisition** 10

4 **Software synthesis** 26

5 **Ontologies** 34

6 **Reasoning Maintenance** 43

7 **Model-Based Reasoning Systems** 49

8 **Agent Systems** 59

9 **Methodologies for Intelligent Systems Development** 78

Chapter 1

Introduction

1.1 Overview

This module introduces a variety of methodologies important to the development of knowledge-based systems (KBSs) and their applications. The module covers topics regarding different processes within a KBS lifecycle, ranging from knowledge capture and analysis, systems design and implementation, to knowledge maintenance and system evaluation.

In particular, within this module, students will learn about:

- The life cycle of a KBS.
- Basic methods by which knowledge appropriate to the construction of a KBS can be gathered and organised systematically.
- Techniques by which a KBS suitable for a given domain may be automatically synthesised.
- Ontologies that provide a common vocabulary of a domain, and define the meaning of the terms and relations between them.
- Ideas of reasoning maintenance that a KBS can use to make hypotheses and explore their consequences.
- Example approaches for building practical problem solvers that (re-)use knowledge.
- Maintenance, evaluation and cooperation of KBSs.

1.2 Data, Knowledge and Knowledge Engineering

The main aim of the development of an intelligent system is to represent and to reason about, as adequately as possible, the existing domain knowledge, in order to perform a certain problem-solving task. In the most general sense, problem-solving can be thought of as a process of mapping the domain space onto the solution space of the given problem. This mapping is enabled by the use of knowledge, which reflects domain theory and heuristic rules, and/or analysis of the problem data available.

Here, *data* is the raw material that information comes from; they could be numbers or nominal values. *Information* is generally regarded to be any structured data which has contextual meaning. What is knowledge then? *Knowledge* is a concise presentation of information, which condenses structured

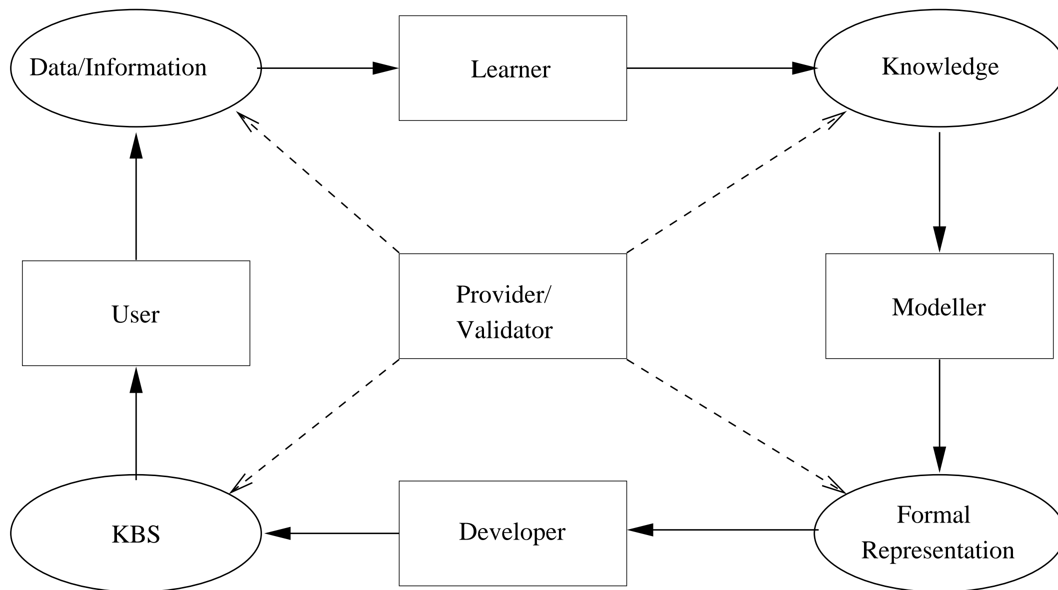

Figure 1.1: Major processes for knowledge engineering

data and abstracts previous experience at a high level. Knowledge reflects general, stable and long-term information of the domain, whilst data conveys specific, volatile and short-term information. Of course, most broadly speaking, information includes both data and knowledge, and together they make the core of an intelligent system. Thus, data and knowledge must coherently fit each other.

Knowledge engineering concerns the basic issues involved in building intelligent problem-solvers. It includes (though not exclusively) the learning, representation, interaction, explanation, validation and adaptation of knowledge and data [Kasabov, 1996]. The central tasks of knowledge engineering cover:

- *Learning* acquires knowledge, typically through generalising examples, with or without supervision, or via experiments.
- *Modelling* transforms existing domain knowledge to a particular computer format, often by expressing the knowledge in a formal language that supports *inference* (which performs a chain of matchings between current facts and the knowledge, thereby deducing new facts).
- *Development* designs and implements the system, which reasons with the formally represented knowledge, to solve the problem at hand.
- *Validation* tests the performance of the resulting system, usually in terms of given quality factors in comparison with alternative systems.

These are summarised in figure 1.1. Note that the knowledge (and other forms of information and data) may well be provided by the same person as the system validator, usually a specialist in the problem domain. In addition, this person may happen to be the end user of the resultant system.

To support performing the above central tasks, the following additional processes are also very important for knowledge engineering:

- *Interaction* allows communications between the system and the environment/user, where the user may be another computer system.
- *Explanation* traces the process of inferring solutions in a contextually comprehensible way.
- *Adaptation* modifies the system during its service to cope with a dynamically changing environment.

Note that in the literature, the term knowledge engineering is sometimes restricted to the process of building a knowledge base within an intelligent system [Russell and Norvig, 1995], where only the determination of what concepts are important in a particular domain, and the creation of a formal representation of the objects and their relations in that domain are considered. However, it is the more general interpretation of the term as outlined above that is adopted herein, although not all the issues are addressed in equal detail (many of which are covered elsewhere, e.g. in the KR and LfD1/2 modules).

1.3 Knowledge Classification

No matter what issue of knowledge engineering is considered, it is important to know what sort of knowledge is being addressed. In general, knowledge used to build an intelligent system can be characterised at least along the following two distinct dimensions:

- *knowledge source* that shows where the knowledge comes from
- *knowledge orientation* that determines what subject the knowledge addresses

Different combinations of these two dimensions indicate different characteristics of the given knowledge. Typical classes of knowledge that can be identified along these dimensions are given below.

There are two major sources from which knowledge can be obtained, which are commonly referred to as empirical and theoretical:

- *Empirical knowledge* relates to that gathered directly from first-hand experience. It captures what has been induced from direct observation of the problem domain or system. Empirical knowledge is, therefore, highly effective when applicable, although limited in its generality. It is this kind of knowledge that has been mostly utilised within a vast majority of developed rule- or case-based systems.
- *Theoretical knowledge* is, on the other hand, the knowledge of scientific laws and principles. It is very general and transferable from one application to another. The use of such knowledge helps remove much of the difficulty encountered in the knowledge acquisition phase associated with empirical knowledge. However, this very generality may mean that it can be much less efficient and even less effective. This limitation is a major reason that its use in intelligent systems has, until fairly recently (with the development of model-based systems – see later), been almost exclusively restricted to numerical descriptions for uncertainty-handling only.

Clearly, these two sources of knowledge are complementary; a combined application of empirical and theoretical knowledge may well offer the best trade-off between generality and efficiency.

Against the knowledge orientation dimension there also exist two commonly referenced classes of knowledge:

- *Object-level knowledge*, which describes the properties or behaviours of the domain concepts themselves.
- *Meta-level knowledge*, which usually reflects reasoning knowledge about the knowledge at the object level.

For example, in developing a medical diagnostic system, the medical rules associating symptoms to possible diseases encode the object-level knowledge, whilst the mechanisms used to focus the rule-firing are normally based on the use of meta-level knowledge. Another example to compare object- and meta-level knowledge is to look at the differences between an ordinary fuzzy logic controller and a self-organising fuzzy logic controller [Lee, 1990]. The former applies object-level knowledge (or modelling knowledge as termed in Control Engineering) that directly relates state variables to control variables via a collection of if-then rules. The latter works by following a two-step process: i) measuring the performance of an ordinary fuzzy logic controller on-line, and ii) using the resultant performance measurements to modify the if-then rules employed by the object-level controller. Such modifications are guided by the meta-level knowledge that reflects knowledge about the knowledge used to control, rather than the control knowledge itself.

At a more detailed level, knowledge can be further classified into many more categories. For example, these may include those determined by the following modelling dimensions [Kasabov, 1996]:

- *Global vs. local knowledge*. The former is applicable to the problem as a whole and the latter is for particular aspects only.
- *Explicit vs. tacit knowledge*. Explicit knowledge shows the structure in which it is itself declared, whilst tacit knowledge is usually hidden in a knowledge source.
- *Complete vs. incomplete knowledge*. A piece of knowledge is complete if it allows a solution to the problem (or part of the problem) in all situations, or incomplete if it only applies to some cases.
- *Certain vs. uncertain knowledge*. The former ensures an exact solution is produced if an exact problem description is given, and the latter allows for an approximate solution given an inexact problem description.

Understanding these knowledge category dimensions helps to identify and establish useful methodological links between the required knowledge engineering tasks and the available knowledge, in dealing with knowledge-based problem-solving.

Chapter 2

Knowledge Systems and Human Interface

2.1 Knowledge Acquisition

In constructing a Knowledge Based System (KBS), there are two interfaces that have to be taken into account. We will first consider knowledge acquisition: how can we go from a body of expertise into the KB of a KBS? We will then consider the need for an explanation facility.

An initial problem in constructing a KBS is the source of the knowledge that is to be encoded. In early systems, the builders of the systems were themselves often domain experts, but this is not usually so. Getting hold of knowledge is in fact a major bottleneck in producing applicable expert systems.

In building a Knowledge System, usually the system builders are experts as far as the programming knowledge is concerned, but where knowledge of an application domain is concerned, the knowledge has to be elicited from domain experts (e.g. the Amphion KB on astronomical observation).

Stages of knowledge acquisition

We can represent stages of knowledge acquisition as follows (see Figure 1). Note how this resembles the software engineering process – in particular this typically involves iteration through the loops shown until a satisfactory system is achieved. However, the *conceptualisation* stage is easier or absent in a standard SE model.

The following stages are involved.

1. **Identification**
   What problems will the system be expected to solve? (*eg* what data are available, what counts as a solution?). Also the resources available (people, time, computing power) determine how the problem may be tackled.

2. **Conceptualisation**
   This is perhaps the hardest stage. The expert's knowledge is usually unstructured, and it is important to work out what the key concepts are, and how they are related, so that the KB will eventually be easy to construct and comprehensible.

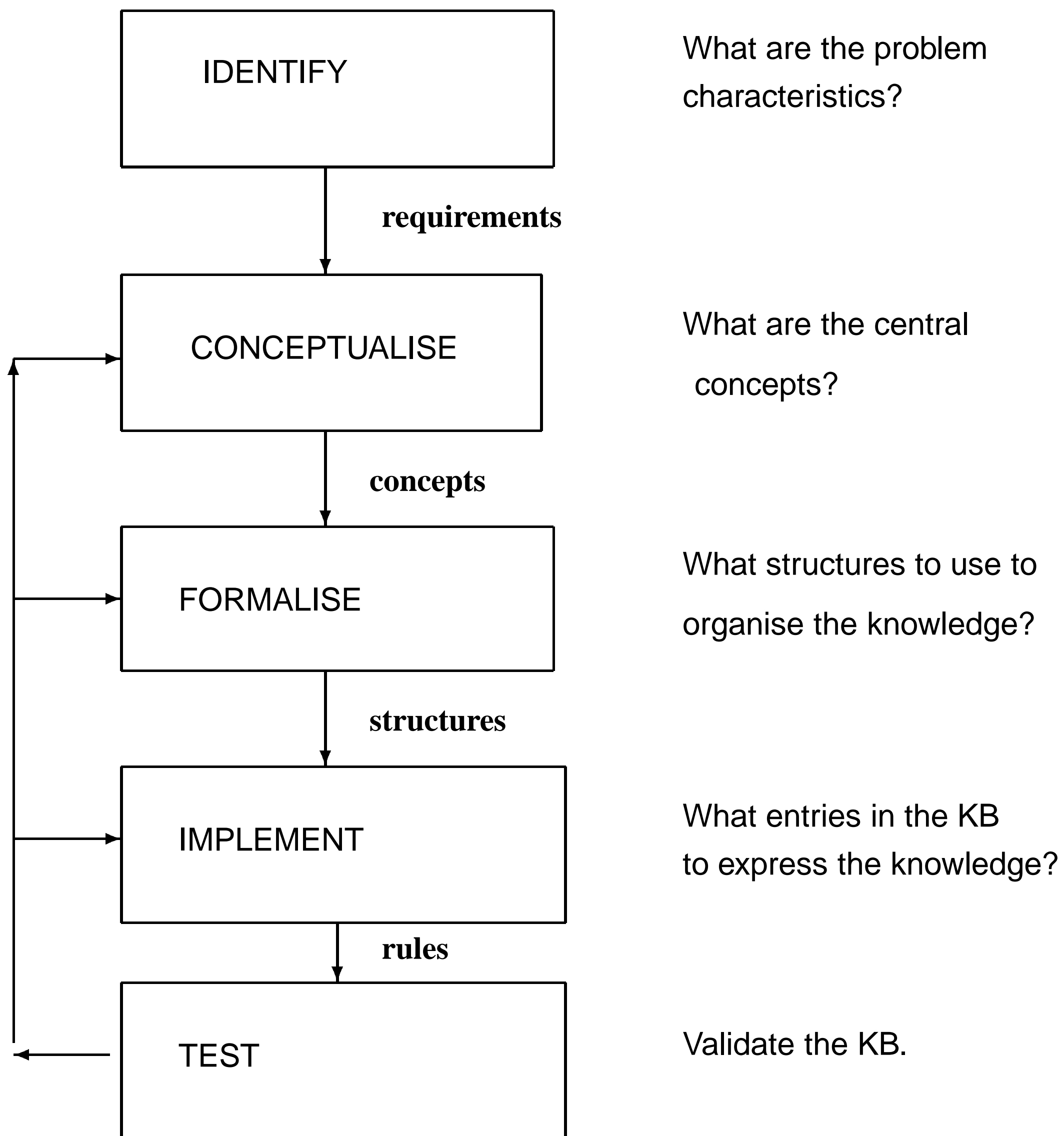

Figure 2.1: The Knowledge Acquisition Process

3. **Formalisation**
   At this stage, an appropriate formalism has to be chosen (eg production rules, logic, etc.). This depends on the nature of the search that will be required, and factors such as
   - Is the information complete?
   - Is the information certain?
   - Is there time dependency?
4. **Implementation**
   Now the knowledge should be put into the selected formalism, using appropriate data structures, and an appropriate *control regime* chosen.
5. **Testing**
   The final stage is to test the program on sample data. This may reveal missing or incomplete rules. It also often throws up combinations of circumstances that had not been considered. To do this systematically, it is best to start from a well-defined *class of problems*.

Machine Learning

Since the knowledge acquisition process is time consuming, various methods to provide some automated support for it have been proposed. One is the use of techniques from machine learning. The idea is that the rules used in the system should be inferred automatically from some number of examples, rather than entered explicitly. There are several possible approaches.

1. An interactive program could learn by "conversing" with the expert.
2. A program could scan textbooks and written sources to convert this into knowledge.
3. A program could learn the central concepts and rules from the study of a series of examples.

All of these have been attempted, but only the last seems to work. For one thing, the first two require a natural language interface to the program, and it is hard to make this sufficiently robust.

Machine learning works as follows.

Given a number of examples which are either *positive* (fit the concept) or *negative* (don't fit the concept), and a language for describing examples, find a description in the language that best describes the examples.

This helps the knowledge acquisition process, since often the expert uses rules she is not aware of, whereas it is easy to decide on individual cases. Thus, for example, given a large number of judgements of whether a loan should be given to an applicant, and the evidence used to come to the decision, a machine learning program can determine a set of rules that would give the same results, and that can be applied to future examples.

2.2 Explanation

We want our KBS to *explain* its reasoning, for two reasons.

- We want to convince the end user that the reasoning is substantially correct.
- We want to convince the knowledge engineer that the system is working properly.

Note that these ask for different sorts of explanation. The second calls for something like a trace mechanism. The first calls for the implementation details to be hidden.

Examples

In a rule-based expert system, we can easily generate an augmented trace of the rules that fire during inference, and similarly retrieve parameter values. This yields a verbose explanation – for larger systems, this sort of explanation will be opaque, as the quantity of detail obscures the important steps.

One solution is to have *levels* of trace, to pick out the features of interest. For example, the user is usually not interested in knowing about rules that were tried and failed (though the knowledge engineer may be).

We can also have an explanation of *control* – why was one inference preferred above another. This is important in systems that support SE *design*.

Sometimes the user is queried interactively for information, rather than entering all information into the KB – when a goal cannot be solved from the given KB, the user is queried to see if relevant information is known to the user, but not in the KB.

Then we can generate explanations based on the dialogue, for example:

```
To show Fred should take exam_51
I used rule
   <person> should take <exam>
   IF <person> studies <subject>
   AND <exam> examines <subject>
   AND NOT <person> exempted <exam>
You said Fred studies CS2.
I know exam_51 examines CS2.
I can show not Fred exempted exam_51.
```

Logic-Based Explanation

In the logic-based approach, we can use the *derivation* as the source of an explanation. This simply records which inference rules were used, and which conclusion were drawn. This is like looking at the successful part of the trace of the derivation search. This however says nothing about control (why this derivation is found rather than another).

However, a good explanation should pick out the *key* steps of a derivation – we do not know how to do this.

Logic can be used as

- Specification language for expert systems;
- representation language for domain knowledge;
- meta-language for controlling and justifying inference.

For explanation, we want aspects of each of these. For good explanations, we'd need an *explicit* formulation of structural knowledge (*ie* how the KB fits together, or the conceptualisation); and also some model of the user's knowledge and skill, so that the explanation is in terms that make sense to the user.

For the second, this suggests incorporating some ideas from tutoring systems.

Chapter 3

Knowledge Acquisition

3.1 Introduction

Before you can build an intelligent reasoning system you have to investigate the problem domain thoroughly, discover the basic concepts, terminology, inference strategies and so on. The process is known as *knowledge acquisition*. This process is very difficult for any but simple systems; if it were easy there might not be a need for an intelligent system in the first place. It is also an art rather than a science, although there are now some techniques which can ease the burden a bit. This chapter introduces several basic approaches to coping with knowledge acquisition.

3.2 Knowledge Elicitation

A related term to knowledge acquisition is *knowledge elicitation*, which is regarded by some people as synonymous and by others as referring to the more specific part of the process in which you and a domain expert are actually communicating. This is discussed below.

3.2.1 Some issues

Suppose you are asked to create an intelligent system to advise customers in a large DIY store about choosing a glue. Most of its users will not know the jargon associated with glue technology. You have to go to an expert (or several experts, as long as they are reasonably agreed about the domain) to acquire initial knowledge that you need to build the system. Do not say "tell me all you know about glues"; if s/he does you're probably in trouble, more probably s/he won't know where to start and will classify you as unprofessional. You have a serious problem to solve at this point. If you start with high-level generalities you *may* lose the expert's interest and enthusiasm; if you start with interesting specific cases you may find it very hard to draw out the kind of generalities that make the resulting system much more than a humble database.

For example, if you are discussing the so-called 'super-glues' the expert might point out that some of them are useless if the joint will later be subjected to unusual humidity or significant moisture. At this point you may ask yourself: what is 'significant', what is 'unusual', is this issue about susceptibility to water a common feature applying to many glues which ought to be a factor in any dialogue, or is it some subsidiary factor to be raised after tentatively deciding that the 'super-glue' might be the one for the job? All of these questions, and many more, have to be resolved at some stage – but when,

and what should you do right now? Keeping track of all such threads over several interviews is very demanding and calls for good personal organisation and forethought.

In the domain of recommending a glue you can at least start by observing that there are just a finite number of ultimate recommendations – the available glues, plus the recommendation that some other form of joining such as welding or riveting or bolting may be better, and perhaps others such as to seek professional advice. Thus you can ask for categorisations of the available glues – in what circumstances would each be recommended, what are the limitations of each, and so on. However, you should not just trust that the expert will do a thorough job of answering such pedantic questions. Remember that the expert gets to answer a lot of rather boring but specific requests, and probably enjoys trying to answer the relatively few more exotic requests such as how to glue ostrich feathers onto a giant inflatable nylon model of an ostrich, or stick down empty barnacle shells to real rocks in a salt-water tank in a large public aquarium. So, you should get down to such cases as part of the 'knowledge mining' process. You have to invent them or get the expert to resurrect them from past experience, or both. The feedback you get may help to check that you are getting a good coverage of the knowledge, or that you are missing some issues, attributes, processes or decisions.

An early step is perhaps to list the types of glue and the possible properties of an unspecified glue, such as price, strength, ease of use, drying time, what it is suitable for, what it is not suitable for, conditions of use, shelf life, precautions when applying it, available sizes. Each property has a set of possible values, although some are rather fuzzy – strength, for example, could depend on what is being joined and the subsequent environmental conditions. Some properties such as `shelf-life` may only need a single value such as `weeks`, `months` or `years`; others such as `available-sizes` will have a set of values; others may require a numeric range or set of ranges. You may also find that all this information can be more conveniently organised by constructing a categorisation of glues into cyanoacrilates, synthetic-latex glues, rubber-latex glues, resin-based glues and so on, although it is not so easy to decide on a taxonomy – a complete categorisation without overlap.

It is a lot harder to list the types of application about which a user might enquire, and the possible salient properties of such applications. There are also intermediate levels of the whole process to worry about; it is unlikely that the user's information will immediately determine the answer. Instead, the process is rather like diagnosis. For instance, if the user starts by saying that s/he wants to join two metal parts, this may suggest possible answers of a resin-based glue, a cyanoacrilate, soldering, welding, or mechanical fastening and the task becomes one of asking enough to decide between the possibilities. You may decide, therefore, to build a system which will interact with the user so that they can converge on an answer – the user's information suggests initial recommendations, the system provides feedback about the options which trigger the user to provide more detail about the application, and so on.

The task of sorting out attributes, values and so on for all the entities within a proposed system is usually demanding. Various techniques are commonly used, the basic ones are described below.

Obviously you must also put a lot of thought into your interview technique. Common-sense observations include: don't interrupt (much); try to deal in specifics rather than generalities; be well prepared in advance; try to tape or video interviews rather than relying purely on handwritten notes; don't impose your own views about how to communicate the material; be thoughtful about suggesting what kind of aids might be used (some people don't like whiteboards, or flowcharts, or algorithms, or rules, or numbers).

3.2.2 Interviewing: teach-back

A straightforward but under-specified technique is to try to reformulate and summarise what the expert has said and teach it back to him/her. The same kind of idea is sometimes used in teaching. It has drawbacks: it is hard to do well, and the expert may of course be tempted just to approve of your version without assessing it for more subtle qualities such as completeness and conciseness.

3.2.3 Interviewing: laddered grids

This technique explicitly views the domain as laid out, in some sense, along an axis from the general to the highly specific. Starting with some 'seed' item you grow a network of how the domain items relate to each other by asking a number of 'directional' questions such as:

- To move towards the specific:

  "How can you tell it's a ... ?"
  "Can you give examples of ... ?"

- To move towards the general:

  "What have ... and ... got in common?"
  "Where do ... and ... arise from?"

- To move orthogonal to the axis:

  "What are alternative examples to ... ?"
  "What distinguishes ... from ... ?"

You are liable to cover a lot of large sheets of paper. Note that there are also tools to support this technique (e.g. PC-PACK produced by Epistemics).

3.2.4 Interviewing: card sorting

This technique is specifically for the job of discovering what attributes of domain elements serve to distinguish them from each other. The domain elements are written on cards, one per card. You ask the expert to sort the cards into piles somehow. Afterward, you ask what was the basis of the sorting procedure, for example by asking what each pile represents. You may go through each pile and ask why each member is representative of that pile, or to what extent it is representative. You can also ask the expert if there are any cards missing from the pile. The procedure is repeated as often as you can both stand it!

Hierarchical card sorting is an easy variant. After a sort, each pile can be treated as a domain in itself and sorted into sub-piles.

3.2.5 Interviewing: repertory grids

This technique is based on a model of how people regard the world, called 'personal construct theory', devised by psychologists. The idea is that each domain element in some set of interest is classified according to a set of *constructs* that apply to all the elements of the set to some degree – a construct is a linear scale with two extreme values, such as heavy/light, cheap/expensive, wet/dry, conscientious/dilatory or whatever.

Every construct of the set of constructs that apply to the domain set is then expressed as a numeric scale, using the same range of numbers each time, typically 1—5 or 1—9, with the centre of the scale representing some kind of medium value for that construct. A grid is constructed and the expert is asked to assign a number for each construct for each domain element, showing how s/he would place that domain element on that construct scale.

An example will help to make this clearer. Imagine that the domain elements are certain types of crime: petty theft, burglary, drug-dealing, murder, mugging and rape. It should be obvious from figure 3.1 that the various constructs are being ranked on a scale of 1—5 and that it is not always the value on the left-hand side of a scale that corresponds to the value 1.

	petty theft	burglary	drug-dealing	murder	mugging	rape	
anybody	2	1	1	1	1	5	women only
long sentence	2	1	1	2	3	5	short sentence
sensational	2	5	1	1	4	5	common-place
premeditated	5	3	1	2	5	4	spur of the moment
OK for the victim	3	2	2	5	5	5	nasty for the victim
impersonal	2	2	1	5	4	5	personal
petty	1	3	1	5	4	5	major
non-violent	1	1	2	5	5	5	violent

Figure 3.1: A rep grid example

You may well disagree with the numbers in this example, but it is the expert's view that is at issue rather than yours. The next step of the analysis is to consider carefully whether any pair of constructs are very similar, by comparing horizontal lines in this grid. The closest pair of constructs is probably the personal/impersonal one and the major/petty one. Beware, when making this comparison, that the expert may have 'inverted' the scale for just one of two similar constructs. For example, in the example the major/petty construct has a value of 5 for 'major'. If the expert had chosen 1 instead, and 5 for 'petty', then this construct and the personal/impersonal one would look very different.

If, after considering each construct as shown, and inverted scales, you find a pair which are similar, you may have to consider omitting one of the pair since they might arguably represent the same information. In this case suppose that all of them are kept, since it can be argued that there is a clear difference in meaning and so the similarity is due to other factors.

The next stage is to draw up a table showing how similar or dissimilar each domain element is from the others. For example, when the absolute-value metric is used, the (numeric) difference

between petty theft and burglary is

$$|2-1|+|2-1|+|2-5|+|5-3|+|3-2|+|2-2|+|1-3|+|1-1|=10$$

Also, the minimum difference possible is 0 and the maximum is 32. A table of differences would look as shown in figure 3.2.

	petty theft	burglary	drug-dealing	murder	mugging	rape
petty theft	-					
burglary	10	-				
drug-dealing	10	10	-			
murder	18	18	16	-		
mugging	15	15	21	9	-	
rape	23	21	29	13	10	-

Figure 3.2: Comparing domain elements

Note that any other distance metric may be utilised for this, though the introduction of a more complex metric may increase the overheads over the calculation of the distances. Note also that it is sometimes more convenient to standardise the numbers by expressing them as percentages of the maximum possible difference but this can, by showing larger numbers, introduce a superficial air of precision.

It is reasonably evident from the table that mugging, murder and rape are fairly similar to each other; so are petty theft, burglary and drug-dealing. Rape and drug-dealing are the most dis-similar. A more sophisticated cluster analysis can be done if there are many domain elements. The final output is useful mainly for two purposes: to help sort out the categorisation of the domain elements, and to find similarities and differences for further discussion.

3.2.6 The importance of jargon

You need to learn to talk the same kind of jargon as the expert, but you also need to pay careful attention to the kind of jargon which the end-user will expect and will be comfortable with. Many intelligent systems projects have failed on this point alone. For example, in some financial application system that advises about the financial health of small companies one of the atomic ingredients used in describing the domain knowledge might be a quantity called the 'quick ratio'. The 'quick ratio' is defined to be current assets less inventory, divided by current liabilities; it is used as a rapid (and crude) measure of health, because it shows how well a company could meet its liabilities without selling anything that is vital to its business. In general, regarding this quantity, any value larger than 1.0 is taken to be good!

You must then ask yourself: do the end-users know what the 'quick ratio' is, or must it be explained? Would any users feel patronised if it were explained to them? Do they call it the 'quick ratio', or do they call it the 'QR'? Some places call it the 'acid test ratio' instead. If you show due sensitivity to the local culture by paying attention to such details (perhaps even making them customisable by some separate tool which edits the knowledge base easily) you are much more likely to win acceptance for your system.

Sometimes you may find that cultural dependencies must be built into the knowledge base. For instance, a British medical diagnostic system might want to adjust the likelihood of cholera as a diagnosis according to whether the patient has recently returned from a tropical or semi-tropical country. Obviously the related question would be inappropriate if this system were being run in Bangladesh, or even in Italy. Intelligent systems need to have relevant local common knowledge wired in rather than asking for it.

3.3 Rule Induction

In addition to using standard interview techniques, sometimes it is also possible to use machine learning techniques in general and rule induction in particular to generate rules from data semi-automatically. Informally, the notion of *rule induction* concerns the condensation of an existing set of highly specific rules into a new and more efficient form. This often appears in the literature as 'learning from examples'. To give you a brief account of the basic ideas of rule induction, two such learning algorithms are introduced in this section.

3.3.1 Entropy-based approach

Consider the following scenario: you are stranded on a desert island, with only a pile of records and a book of poetry, and so have no way to determine which of the many types of fruit available are safe to eat. They are of various colours and sizes, some have hairy skins and others are smooth, some have hard flesh and others are soft. After a great deal of stomach ache, you compile the table of data shown in table 3.1. This is pretty tedious to look up each time you feel hungry; is there some kind of pattern to it? This can be viewed as a set of examples, where each line states the values of certain variables and a conclusion. On the other hand, it can be viewed as a set of rules: the first line is

```
if   skin = hairy
and  colour = brown
and  size = large
and  flesh = hard
then conclusion = safe
```

Such rules are not constrained to have a fixed number of preconditions, of course. Suppose that there were a rule which had no precondition about the size of the fruit. Then it could be replaced by two rules, one in which `size = large` and one in which `size = small` were extra preconditions, or it could be replaced by a rule with the single equivalent extra precondition `size = Anything` or the precondition `size = large or size = small`.

It is possible to devise a *decision tree* to replace this set of rules, automatically. The decision tree is of the general form

```
if        variable1 = value1 then <subtree 1>
else if variable1 = value2 then <subtree 2>
else if ...
...........
else if variable1 = valueN then <subtree N>
```

This corresponds directly to a set of rules, with as many rules as there are leaf nodes in the whole tree. Each rule is a tracing out of the path from the top of the tree to a leaf node. The process is not limited

Conclusion	*Skin*	*Colour*	*Size*	*Flesh*
safe	hairy	brown	large	hard
safe	hairy	green	large	hard
dangerous	smooth	red	large	soft
safe	hairy	green	large	soft
safe	hairy	red	small	hard
safe	smooth	red	small	hard
safe	smooth	brown	small	hard
dangerous	hairy	green	small	soft
dangerous	smooth	green	small	hard
safe	hairy	red	large	hard
safe	smooth	brown	large	soft
dangerous	smooth	green	small	soft
safe	hairy	red	small	soft
dangerous	smooth	red	large	hard
safe	smooth	red	small	hard
dangerous	hairy	green	small	hard

Table 3.1: Identifying what is safe to eat

to preconditions of the form `This = That`, either, although for the purpose of explaining the idea only this kind of precondition will be considered. The key question is which of the variables (or attributes) is the most useful determiner of the conclusion of the rules. Whichever it is, that variable ought to be the first or topmost one in the decision tree. Information theory provides one answer, as well as providing a meaning for the notion of 'most useful determiner'. To formalise it, suppose that the conclusion C can have any of the values $c_1 \cdots c_n$ (in the example above, $n = 2$ and the conclusions are 'safe' or 'dangerous'). Suppose that a certain variable A can take values $a_1 \cdots a_m$. Then the expression

$$-\log_2 p(c_i|a_j)$$

is deemed to be a measure of the 'amount of information' that a_j has to offer about conclusion c_i – it is, in information theory terms, the *bit length* of the probability, and this is where the term 'bits' in computer science came from. Information theory defines

$$\text{entropy} = -\sum_{i=1}^{n} p(c_i|a_j) \log_2 p(c_i|a_j)$$

to be the *entropy* of a_j with respect to the conclusion C. Entropy is a measure of 'degree of doubt' – the higher it is, the more doubt there is about the possible conclusion.

This definition of entropy is not arbitrary. Suppose there is a set of possible events, but all that is known about them is their probabilities $p_1 \cdots p_k$. Any measure of the uncertainty about which event will occur should be some function $H(p_1, \ldots, p_k)$ of the probabilities, and should satisfy the following very reasonable conditions:

1. It should be a continuous function of each argument.

2. If the probabilities are all equal ($= 1/k$) then entropy should increase as k does. With equally likely events, when there are more events there is more uncertainty about the outcome.

3. If the ‘choice’ between events is split into a set of successive ‘choices’ then the entropy of the original set should be the same as the weighted sum of the entropies at each step of the successive ‘choices’. That is, the entropy should not depend on the ‘choice’ method. For example, this means that if the events are concerned with throwing a die, and are specifically “three or less”, “four or five” and “six”, then this could be cast as a two step process: “is it three or less, or higher?”, then “if higher, is it six or not?”. So, in mathematical terms, this independence means that

$$H(\frac{1}{2}, \frac{1}{3}, \frac{1}{6}) = H(\frac{1}{2}, \frac{1}{2}) + \frac{1}{2}H(\frac{1}{3}, \frac{2}{3})$$

It can be shown that the entropy function above is the only one which satisfies these conditions.

To return to the original problem now, the lower the entropy of a_j with respect to C is, the more information that a_j has to offer about C. Using this expression, the entropy of the variable A with respect to C is

$$-\sum_{j=1}^{m} p(a_j) \sum_{i=1}^{n} p(c_i|a_j) \log_2 p(c_i|a_j)$$

and the variable which has the lowest entropy is the most useful determiner.

Consider the variable *Size* in the above example. From the table, when frequency measures are approximately taken to be the probabilistic measures, it follows that

$$\begin{aligned} p(safe|large) &= 5/7 \\ p(dangerous|large) &= 2/7 \\ p(large) &= 7/16 \\ p(safe|small) &= 5/9 \\ p(dangerous|small) &= 4/9 \\ p(small) &= 9/16 \end{aligned}$$

Thus the entropy for *Size* as a determiner of *Conclusion* is, using logarithms to base 2:

$$7/16(5/7\log(5/7) + 2/7\log(2/7)) + 9/16(5/9\log(5/9) + 4/9\log(4/9)) = 0.9350955\ldots$$

Since it is only necessary to compare entropies it makes no difference to use $log_e(\ldots)$ rather than $log_2(\ldots)$. If you do all the calculations you will find that *Colour* has the smallest entropy. The next step is to partition the set of examples according to the possible values for the colour, and apply the same process to each subset. This eventually leads to the rules:

```
if   colour = brown
then conclusion = safe

if   colour = green
and  size = large
then conclusion = safe

if   colour = green
and  size = small
then conclusion = dangerous
```

```
if   colour = red
and  skin = hairy
then conclusion = safe

if   colour = red
and  skin = smooth
and  size = large
then conclusion = dangerous

if   colour = red
and  skin = smooth
and  size = small
then conclusion = safe
```

This set of six rules, each smaller than any of the originals, is an improvement on having those sixteen. They are all smaller because, as you might have guessed, the variable *Flesh* is irrelevant. If you look at the table, you will find some examples of lines in it which differ only in the value of this variable. From such an observation you can only conclude that the variable is, under certain combinations of the other variables, irrelevant; it should be a little surprising to you that it is always so.

The six generated rules can be even more succinctly put, however – only small green ones or large smooth red ones are dangerous. Of course, that word 'only' conceals the fact that there are only two possible conclusions. Often such a further apparent compression of the rule set will not be easy to find, if it exists at all. One easy method is to try omitting each rule condition in turn, testing to see whether this results in any misclassification of data.

If you consider using this algorithm then there are, as ever, points to bear in mind. First, if the table of data about edibility had included a record of which day of the week the sample had been eaten on, the algorithm would have blindly considered this as a factor. So, wholly spurious correlations are possible, since the algorithm takes no account of any meaning the data it works on may have. Second, the algorithm considers just one variable at a time. If two variables turn out to have almost identical entropies it will still select the lower entropy, whereas it might have been better to consider them jointly (as though they made up a single variable). Suppose that the following data is given:

Outcome	*X*	*Y*
yes	3	3
no	2	1
yes	4	4
no	2	4
no	1	3
yes	1	1
yes	2	2
no	2	3

The entropy-based algorithm is incapable of spotting the obvious summary, that:

```
if X = Y then outcome = yes else outcome = no
```

Third, when inducing rules from large sets of examples in which there are a large number of possible outcomes (that is, n is large), then the algorithm can be very sensitive to apparently trivial changes in the set of examples. The original algorithm, named ID3, tried to cut down on effort by inducing a set

of rules from a small subset of data, and then testing to see if those rules explained other data. Data not explained were then added to the chosen subset, and new rules induced. This process continued until all the data was accounted for. The letters ID stood for 'iterative dichotomiser', a fancy name for this simple algorithm. However, tests have suggested that the algorithm often does not save much effort, you might as well just induce rules using the full set of data at the start! Fourth, the algorithm cannot generate uncertain rules or handle uncertain data (though recent attempts have been reported to extend the basic ideas of ID3 to cope with uncertain data via the use of fuzzy sets [Janikow, 1998]).

In practical use of this algorithm there are other issues requiring consideration. These include the determination of an appropriate depth in growing the decision tree, the preprocessing of training samples involving continuous attributes and/or missing values, and the use of alternative measures for selecting attributes. Fortunately, the algorithm has been extended to address most of these issues (see [Quinlan, 1993] for further details). A brief summary of the extensions made to cover these issues can be found in [Mitchell, 1997].

Although ID3 and its derivatives (represented by C4.5) have been used successfully to condense sets of examples into rules, they must be used with care. You must be sure that the variables you choose to include in describing the examples (the *Colour*, *Skin* and so on in the example above) must include the 'right' ones. If the algorithm is given a complete set of examples then it can usually do a worthwhile job; if it is given only a subset then it is up to you to make sure that the subset is somehow representative of the whole. If a brown-skinned dangerous fruit were to be added to the data table above, the generated rules would change radically. For these reasons, rule induction algorithms are usually used to suggest correlations that can be confirmed by logical analysis afterward, rather than being trusted to get things right by themselves.

3.3.2 Version space-based approach

Many established approaches to learning from examples are *data-driven*, in the sense that revisions to the current (partially) learned results are determined in response to the observed discrepancies between these results and the newly obtained examples or data. Some of these are tailored to performing *concept learning*, that is, to form a general description of a class of objects from a given set of examples. The basic ingredients are:

- a formal rule (or concept) language, perhaps with some semantic information attached,
- a series of examples which are instances of the concept to be learned, and
- a series of examples which are not instances of the concept to be learned.

The learning algorithms' job is to find a 'good' expression in the rule language which is true of all the examples and false for all the non-examples.

A particular approach is based on the idea of *version spaces* [Mitchell, 1982], where a version space is a set of all plausible versions of emerging rules. Considering data-driven learning as a search problem, subject to constraints imposed by given examples, the version space algorithm can then be thought of as a method to keep track of the search through general and specific versions of possible solutions. Here, the solutions are emerging learned rules. Instead of guessing at the correct version of a rule and revising the guess with the increment of examples later on, all versions that have been in agreement with the data so far are kept. Different versions of a rule are related by hierarchies, with the leaves indicating the most specific, the roots denoting the most general and the truth lying between. The number of the hierarchies required is the same as that of the attributes to be considered within

```
(and (love Andy Mary) (love Mary Andy))

(and (love Andy Mary) (love Mary ?))

(and (love Andy ?) (love Mary ?))

(and (love Andy ?) (love ? ?))

(and (love ? ?) (love ? ?))
```

Figure 3.3: General vs. specific premises

the premises of the rules to learn. Within such a hierarchy, as more examples are presented, the gap narrows between the most general and the most specific versions, of the rule under learning, which remain to be consistent with those examples.

Examples used can be either positive or negative. A *positive example* is one to which the rule concerned is applicable and, hence, causes the more specific versions of that rule to be discarded in favour of generalisation that covers all the positive examples already presented. On the contrary, a *negative example* is one to which the rule cannot be applied and causes the more general versions to be discarded.

To illustrate the term of the most general or specific, figure 3.3 presents five different premises for two attributes that might lead to the same conclusion within a rule. In this figure, the generality of these premises monotonically increases from top to bottom, with the first being the most specific and the last being the most general.

Again, in terms of search, the goal of this algorithm is to find all the elements within a version space which are consistent with all encountered examples. However, it is not practical, in general, to perform an exhaustive search in order to reach this goal, as the size of a version space is usually quite large. Fortunately, it is sufficient to keep track only of the "boundaries" within the rule hierarchies, or of the most specific and the most general versions that have been satisfied with the examples presented so far. In particular, as a new positive example is presented, the specific boundary is modified towards more general; whilst as a negative example is added, the general boundary is changed towards more specific. Given enough additional data, these two boundaries eventually converge to a set of learned rules or even to a single general rule. (It is also possible to find an empty set of learned rules if all the given examples are themselves contradictory, though such a situation is of little interest.)

More formally speaking, the version space based algorithm for concept learning [Mitchell, 1982] can be stated as shown in figure 3.4, where E denotes the set of available examples, and S and G respectively represent the set of the most specific and that of the most general versions that were consistent with all previously presented examples.

To see how the version space algorithm can be used to perform data-driven learning, consider an example. Suppose that the task is to learn about which of the many types of fruit are safe to eat, based on the description of some basic characteristics of each fruit type. To start with, hierarchies of fruit characteristics have to be provided with respect to all versions of those characteristics possibly to be considered. For simplicity, assume that only two distinct characteristics, i.e. the $colour$ and $size$, of different types of fruit are considered and the corresponding hierarchies are shown in figure 3.5. Given a set of examples as presented in table 3.2 the version space algorithm works as explained below.

Initialisation: Let S and G, respectively, be the set of the most specific and that of the most general versions that match the first, positive example. If the first example in E is negative then re-order its elements such that the first becomes positive.

The Algorithm: **For** each subsequent example $e \in E$ **do**

- **If** e is a positive example **then do**
  - Retain in G only those versions which match e
  - Generalise those versions in S which do not match e, only to the extent necessary for them to match e while still remaining more specific than or, at most, equal to those in G
  - Remove from S those versions which are more general than any version in G
- **If** e is a negative example **then do**
  - Retain in S only those versions which do not match e
  - Modify those versions in G which match e, only to the extent necessary for them not to match e while still remaining more general than or, at least, equal to those in S
  - Remove from G those versions which are more specific than any version in S

Figure 3.4: The version space algorithm

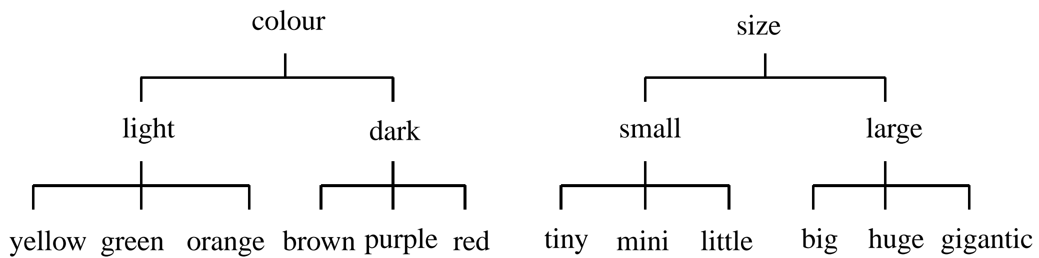

Figure 3.5: Characteristic hierarchies for colour and size

Colour	**Size**	**Conclusion**
green	mini	safe
green	little	safe
brown	huge	dangerous
orange	little	safe
orange	huge	dangerous

Table 3.2: Fruit safety vs. fruit colour and size

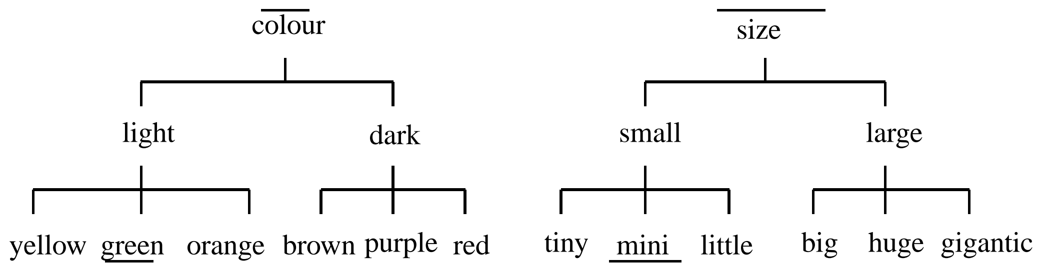

Figure 3.6: Initial version space

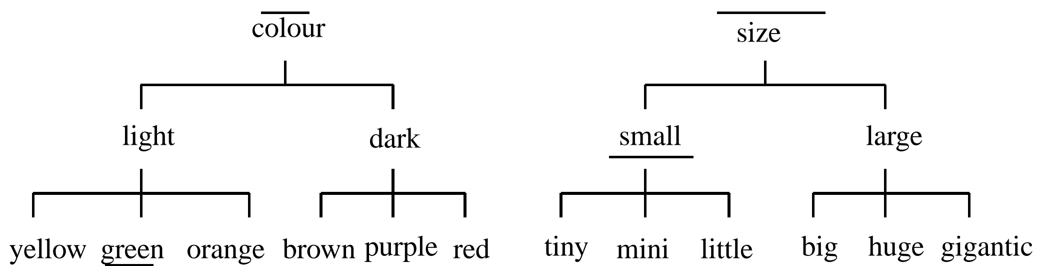

Figure 3.7: Version space after presenting example 2

The given examples form the available data set E. Clearly, to learn which types of fruit are safe to eat, those data associated with a conclusion of safe are treated as positive examples whilst those concluded to be dangerous are seen as negative ones. To initialise the algorithm, the most general and specific sets of versions are assigned to be the following:

$$G = \{[colour, size]\}$$

$$S = \{[green, mini]\}$$

Diagrammatically, the above initialisation can be shown in figure 3.6, where the markers put on the top and bottom of some versions indicate the most general and the most specific, respectively, thereby delimiting the (initial) entire version space concerned. It is worth mentioning that the two boundary sets G and S reflect two extreme cases, with the former representing that every kind of fruit having a $colour$ and a $size$ is safe to eat and the latter expressing that only those types of fruit whose colour is $green$ and whose size is $mini$ are safe.

In response to the second datum in E, which is another positive example, no change is made to the colour hierarchy, since this hierarchy has already been made to suit this example initially. However, the size hierarchy must be altered such that the most specific version becomes $small$. This is required in order to generalise the revised S to be consistent with both of the first two examples, whilst remaining more specific than those versions in G. After presenting this example, the modified sets of the most general and specific versions are shown in figure 3.7, with $G = \{[colour, size]\}$ and $S = \{[green, small]\}$, meaning that, once again, every kind of fruit is safe and that only $small$ $green$ fruit is safe to eat.

The third available example is negative, which states that *huge brown* fruit is dangerous. Following the algorithm, the version space has to be adjusted so that the set G no longer covers this datum, while

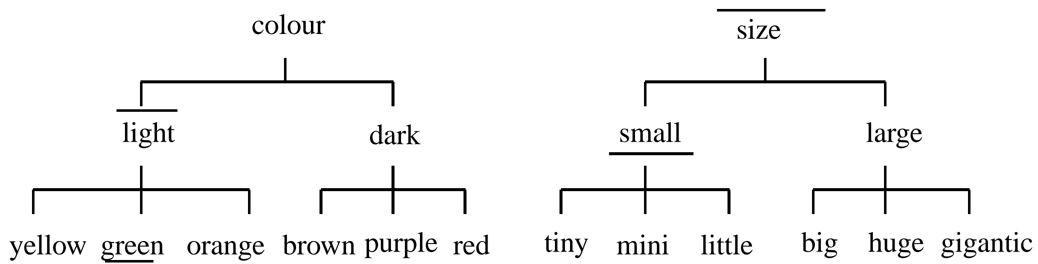

Figure 3.8: Version space after presenting example 3

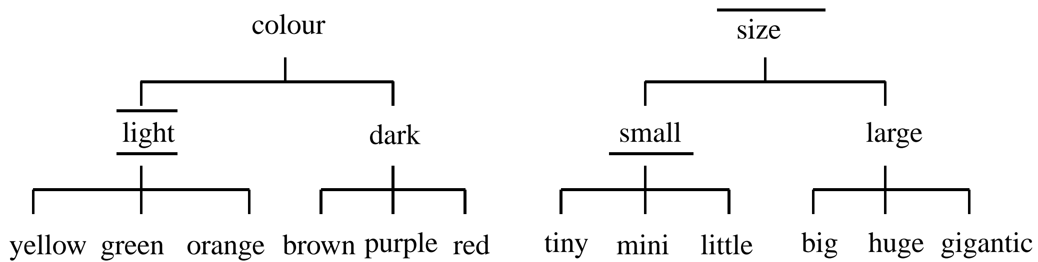

Figure 3.9: Version space after presenting example 4

no change is required for the set S because the only element $[green, small]$ of it does not match this example anyway. There are two ways to modify G; one is to move down the least upper bound of the colour hierarchy such that $brown$ is out of the revised colour hierarchy and the other is to lower that of the size hierarchy such that $huge$ is excluded. Assume that the first choice is made, then the least upper bound of the colour hierarchy is placed at $light$ as illustrated in figure 3.8. This results in that $G = \{[light, size]\}$ and $S = \{[green, small]\}$. (Work out what results if the second choice is made.)

The fourth example is positive, implying that those types of fruit which are $orange$ in colour and $little$ in size are safe to eat. This datum requires the adjustment of the greatest lower bound previously marked at the bottom of $green$ in the colour hierarchy, so that the most specific version will cover both $green$ and $orange$. This lifts the marker up to the bottom of $light$. However, this fourth example does not invoke any modification in the size hierarchy as the version space has already included the case of $little$ fruit. After applying the algorithm to this example, the most general and specific version sets become the following: $G = \{[light, size]\}$ and $S = \{[light, small]\}$. These two sets indicate that every kind of fruit is safe to eat if it is $light$ in colour and that only fruit of a $small$ size and of a $light$ colour is safe. The resultant modified version space is shown in figure 3.9. Note that, within the colour hierarchy, both the greatest lower bound and the least upper bound are placed on the $light$ as a result of this modification step, thereby leaving no room for further manipulation in this hierarchy.

Presented with the final example, which is negative and shows that *huge orange* fruit is dangerous to eat. Knowing that no more revision can be made in the colour hierarchy, adjustment has to be made in the size hierarchy so that the resulting version space does not cover this example. This is done by lowering the upper marker of the hierarchy to exclude the *huge*-sized fruit (which is a kind of $large$ fruit). Thus, the least upper bound is placed at the top of *small*, which is also the current position of the greatest lower marker. With both the most general and the most specific version sets being

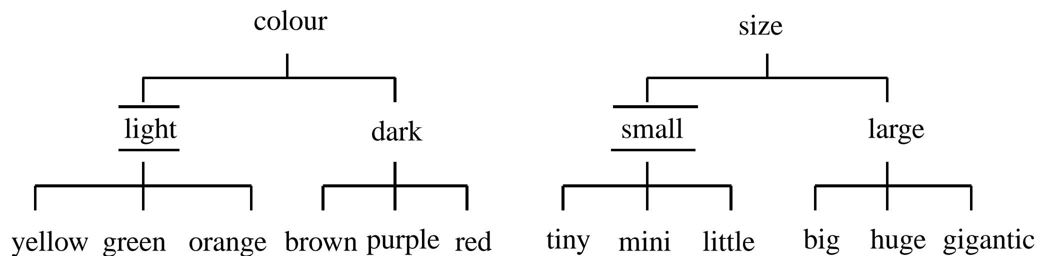

Figure 3.10: Version space after presenting example 5

$\{[light, small]\}$, a single rule which satisfies all given examples is obtained and the corresponding final version space is given in figure 3.10. This learned rule states that if the size of a given fruit is $small$ and the colour of it is $light$ then the fruit is safe to eat (but please do not apply this rule in real life!).

The version space algorithm allows the construction of rules by generalising given examples. However, this algorithm is practical only for problems in which the concept "more specific than", or "more general than", can be identified explicitly. This is required so that a version space can be represented with the hierarchies consisting of versions of the rule or rule set to learn. In particular, within each hierarchy there must be a set of the most general versions and a set of the most specific. The most specific version set summarises the information embedded within the positive examples whilst the most general set summarises the information implicit in the negative examples. Satisfied with this requirement, the algorithm is guaranteed to find, if any, learned rules that are consistent with the observed positive and negative examples, independent of the order of presenting these examples (except that the first example is required to be positive).

3.4 More General Points about Learning

Rule induction, as presented above, is rather simple in principle. Another form of rule discovery is enjoying a vogue: *genetic algorithms*. The basic idea is that new rules are invented somehow (maybe at random, according to some rule language) and then tested for explanatory power. There are endless obvious variations on this theme, and some subtle ones.

Also, as indicated earlier, there are many other learning algorithms for *concept learning*. In fact, variations in the setup of the version space algorithm can lead to different algorithms. Such variations may address questions like

- is the learning supposed to happen as the examples are presented one by one, or can everything wait until all examples are known?
- who says whether an example is positive or negative?
- is the information about the status of an example absolutely reliable, or can there be 'noise' in the data?
- is the algorithm allowed to ask for certain kinds of example, or must it take what comes?
- is the concept expressible purely as a conjunct of conditions ('red and big and heavy') or purely as a disjunct ('red or big or heavy') or neither?

- ... and so on ...

The most general form of concept learning is very hard. After all, the specification, when dealing with reliably classified examples, of

```
(<positive example 1>
 or <positive example 2>
 or .... )
and not (<negative example 1>
         or <negative example 2>
         or .... )
```

is a description of the concept, but not a very useful one, since it is not likely to apply to any unmentioned examples. The task is a search problem through the possible expressions in the language. Even a test for the success of such a search can be difficult to define. Good information about the semantics of the language is usually needed to guide the search.

For further reading about machine learning, see *Machine Learning* by Mitchell [Mitchell, 1997], or a short overview of 'Machine Learning for Intelligent Systems' by Langley [Langley, 1997].

Chapter 4

Software synthesis

4.1 AMPHION: putting together subroutines

There has been recent work on systems to aid in composition of programs for domain-specific tasks (see the Amphion home page[1]).

The aim is to help scientific programmers get more out of the large libraries of routines they often have available. This is a tool aimed to make available the knowledge of just what routines are available, and what they do. Very few programs get re-used in any way, and the problem of working out just what some routine is supposed to do is one reason why new programs get written from scratch, even when something nearly suitable is already available.

4.1.1 Methodology

The system is designed to be used as follows. The user is guided in putting together a formal specification of the problem at hand; a program that meets the specification is then assembled automatically from the subroutines present. The aim is to let users who have the basic concepts of the application domain to be able to describe the problem at the abstract level of specification, rather than in terms of programming constructs. Thus there are generic components to the system (the specification acquisition guidance, and the program synthesis), and for particular domains, a *domain theory* is developed, i.e. a KB about the domain for which an application is being developed.

Components

Thus the system comprises:

- *Specification acquisition subsystem* (generic) –
  To build up domain characterisation.
- *Program synthesis subsystem* (generic) –
  To assemble programs from routines and overall specifications
- *Domain specific subsystem* –
  Theory of the domain, interface routines and automation routines for the synthesis system.

[1] http://ase.arc.nasa.gov/docs/amphion.html

The first two components are *generic* (the same for each application). The specification acquisition system is used to build the domain theory; a characterisation of the software routines is needed, together with an interface between the high-level specification language, and the programming description language. Now the synthesis module works out how to customise and combine the library routines.

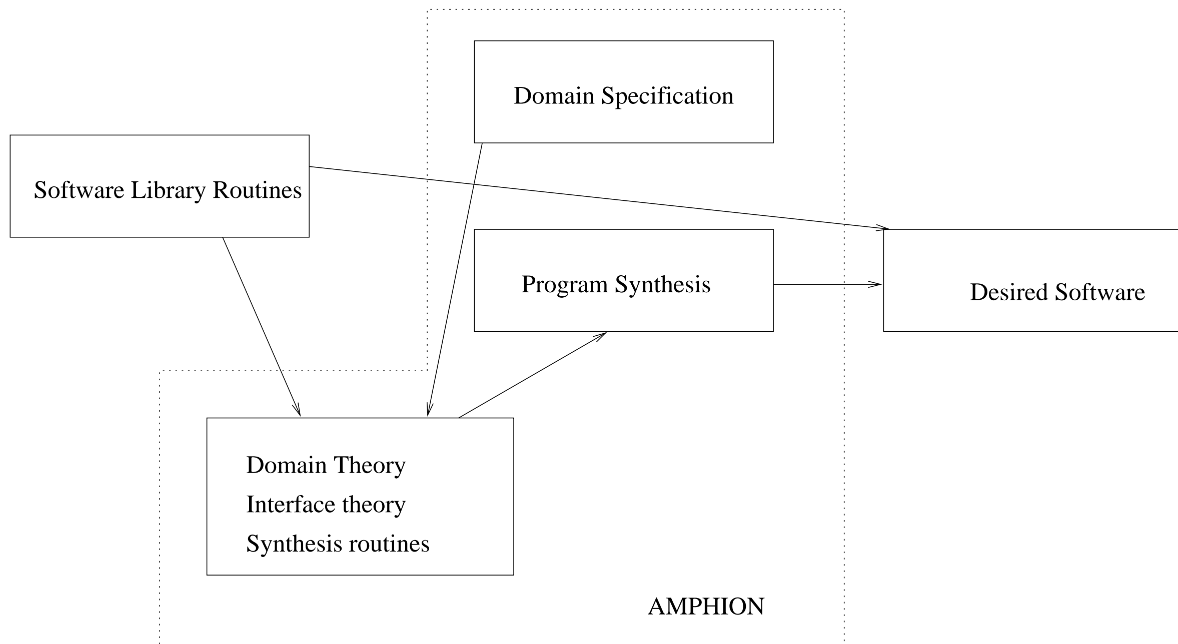

Figure 4.1: Amphion system architecture

4.1.2 Context

The system has been used successfully to develop routines to control astronomical observations, building on existing library of astronomical routines, developed by NASA.

Experience show that programs can be assembled much faster than by normal use of the library, and users can use the system after a short tutorial.

The system uses *deductive synthesis* to put together programs *automatically* i.e., by finding the derivation of a query in a particular logic, it is possible to derive a functional program that fits the specification.

Specification and derivation

These are written in first order logic, augmented with lambda-calculus
(like ML `fn x=> ...` syntax).

The shape of a specification is:

$\mathtt{lambda(inputs)}\ \mathtt{find(outputs)}\ \mathtt{exists(intermediates)}$
$\quad \mathtt{conj}_1 \wedge \mathtt{conj}_2 \cdots \wedge \mathtt{conj}_n$

where the conjuncts are either an equation defining a function, or a constraint (i.e. a statement of the form $\mathtt{P}(\mathtt{v}_1, \ldots, \mathtt{v}_m)$). Specifications are checked to see that they can be solved abstractly. If they

cannot be solved, error information is returned to the user.

Deriving the program

Given a satisfiable specification, look to find a proof of a statement:

$$\forall \texttt{inputs}\ \exists \texttt{outputs}\ \texttt{spec}(\texttt{inputs}, \texttt{outputs})$$

A derivation of a statement of this form gives directly a functional program that fits the specification, ie a program F such that

$$\forall \texttt{inputs}\ \texttt{spec}(\texttt{inputs}, \texttt{F}(\texttt{inputs}))$$

Use a functional language where the terms correspond to routines in the target (imperative) programming language; the functional program can get translated.

Only at this point are any *Language specific routines* needed; the system will generate variable declarations and sequence of subroutine calls. So, the system could be easily adapted to different programming languages.

4.1.3 Example problem

Suppose want to compute angle of sun light at a point on a planet's surface visible from the satellite bore-sight. We can specify this by a series of statements:

Let *Solar-Incidence-Angle* be angle between rays *Surface-Normal* and *Ray-Intersection*. Let *Surface-Normal* be ray normal to *Jupiter-Body* at the point *Boresight-Intersection*.
...

These are the conjuncts of the specification, in the language of Euclidean geometry.

A diagrammatic representation is given to the user, and the program synthesised.

4.1.4 Domain Engineering

Amphion needs to know how to use the domain information (eg geometry).

The subroutines are assumed to be given, and correct w.r.t their own specification. So Amphion has

- and abstract theory of domain (eg geometry);
- a concrete theory of what the subroutines do;
- an implementation relation between abstract and concrete theories.

The *abstract theory* is put together in an iterative process, involving the domain expert, *and* Knowledge Systems expert. For the astronomical domain, it involved collaboration over some time. For example, here it is necessary to choose a time system, and a notion of coordinate frame.

The abstract theory here has:

- types for objects (points, lines, ellipsoids, ...)
- constructors (eg ray from point and direction)
- geometric operations (eg intersection)

- ...

together with other, non-definitional relations; e.g., need to know the relationship between times and places corresponding to light travelling from one to the other.

The library routines have characterised also, in terms of programming language data-types (vectors of reals, eg). Conversion functions are needed too.

4.2 Specification Acquisition

When putting together the individual problem specification, the user interacts via a GUI interface. This builds on the domain theory syntax, and ensures the consistency of the input format.

Specifications may be put together in a top-down, or bottom-up style, according to taste.

The specification will be checked abstractly, to weed out some obvious problems; warnings are given of over-constrained and under-constrained variables. Some overloading of functions and relations is allowed.

Constructing Domain Theories

We now look at tools to help construct a domain KB, and to specify problems in the domain.

Domain Theories and Specification Alongside descriptions of code capabilities, we want descriptions of the domain that the final system is intended for (astronomical observations, computer vision systems, telephone exchange software, etc.). Then, having built such a KB, we want also to support the construction of a specification of a particular problem specification in the domain.

We looks at ways in which these two tasks can be aided, with reference to the Amphion system and others. In each case, we assume there is an underlying logical formulation of the domain theory and of the problem specification. There are several ways we can be helped here:

- Provision of graphical representation;
- Consistency checking: is the domain theory consistent on its own?
- Well-formed input: checking that the specification makes logical sense (e.g. only uses syntax present in the domain theory).

This assumes that the task is specified in terms of the domain theory, and not in terms of computation — this is more acceptable to end users.

Entity Hierarchy It's common to categorise objects in the problem domain in a hierarchy, going from more general to more specific classes (thus the class *polygon* is more general than *quadrilateral* which is more general than *rectangle*, etc.

Logically, the classes correspond to single argument *predicates*; the relation between classes corresponds to logical implication (from the specific to the general). Thus:

$$\begin{array}{lcl} \forall x\, quadrilateral(x) & \rightarrow & polygon(x) \\ \forall x\, rectangle(x) & \rightarrow & quadrilateral(x) \end{array}$$

and we can deduce the relationship between *rectangle* and *polygon*. Notice that this allows us a version of *inheritance*: any property we have of a *polygon* we can deduce will also hold for a *rectangle*.

We can also have several subclasses of a given class. It is natural to express this hierarchy in graphical form; the two statements above suggest a directed graph with three nodes and two edges. A GUI here can be used to build the KB statements as above simultaneously with entering the hierarchy graphically.

Graphical/Logical Translations There are other places where there is such a close link between graphical description and logical that we can translate from one to the other.

For example, we can think of an *electrical circuit* example; it is possible to build up circuit descriptions graphically, and get a corresponding logical description automatically, in terms of a fixed syntax such as one we saw (`andg`, `org`, ..., `output`, `input`, `connected`).

Here we assume that the properties of the components are already known (or supplied by the user), and the circuit specified via a GUI.

4.2.1 Consistency

We want to know that the domain theory, and the (non-negated!) specification query are *consistent*. Why is this?

In a standard logical formalism, a KB is inconsistent if we can derive from it some query Q and also its negation $\neg Q$. This is already a problem for that query (which answer do we believe), but this means we can derive *any* query Q' at all! (This is because $Q \wedge \neg Q \rightarrow Q'$ always holds.) This makes the KB *useless*. So we want to know about the consistency of our KB, if possible.

Testing for Consistency We can sometimes test for consistency of a KB in predicate logic.

For example, put it into clausal form, and look for a derivation of the empty clause. If this fails (i.e. terminates using an appropriate strategy), then we can conclude the KB is consistent. Note that this can be very expensive if done frequently; and we can't guarantee that this will terminate, even if the KB is consistent.

When the KB takes a particular shape (in its logical syntax), then consistency checking becomes possible. For example:

- If only propositional connectives are used (still expensive);
- If only positive definite clauses are used (always consistent).

(A clause is *positive definite* if it has exactly one positive literal – this is the shape of a Prolog program clause.)

4.2.2 Graphical problem specification

This is the approach used in Amphion to allow end users to specify a problem. Recall that there is a background domain theory here of astronomical physics that the end user is comfortable with. The end user wants to put together software to carry out a particular observation.

For example, consider trying to compute the angle between the vertical at the point on Jupiter nearest Galileo, and the sun.

Logical version One way is to start by saying:

Let x be the angle between rays u,v.
Let p be the point on Jupiter closest to Galileo.
Let u be the ray from p to the sun.
Let v be the surface normal at p.
Let x be measured in radians.
Let t be time at Galileo as a string.

The background theory should explain *normal*, *angle* etc.

Graphical version Here the user interacts by creating a graph that describes the relationship between the entities in question. This relies on the system already having the concepts represented in logical form; the specification is thus built up in the course of interaction with the GUI.

The user enters some geometrical objects (planet, ray, angle), and constraints between them as above. The user also specifies which values are *inputs* and which *outputs.* A configuration is then generated by interaction between the user and the system. The basic operations are

- adding objects;
- deleting objects;
- moving edges between objects to define relationships;
- merging objects (to allow compound operations);
- declaring as input/output.

(The example above is shown at the end of this note in graphical form.) Because the syntax is restricted here, Amphion can check specifications, for

- consistency – has a value been over-constrained (e.g. as the angle between x and y, and also between y and z);
- completeness – is there enough information to compute output values (e.g. incomplete if x is angle between u and v, and we only know one point on v).

Feedback identifies the problematic components of the specification. This enables

- reuse of specifications;
- simpler editing of specifications to update;
- bug-free code (as long as Amphion is soundly implemented).

For example, can change input/output representation; reuse a grouped set of objects and relations; change the input/output pattern.

4.2.3 Experience

End users have found this approach congenial. Target end users can use such a tool after a one hour tutorial.

Extension with new subroutines are easy. The domain theory can be easily extended with new sorts of objects by system developers. However, development of initial domain theory was time consuming – it has to be natural for the end users.

This work suggests that such graphical interfaces are helpful in building well-formed logical specifications, via the provision of a translation between the graphical language, and an underlying logical representation.

Graphical Problem Specification in Amphion

The graphical representation produced by Amphion for the specification mentioned above is shown is figure 4.2.3.

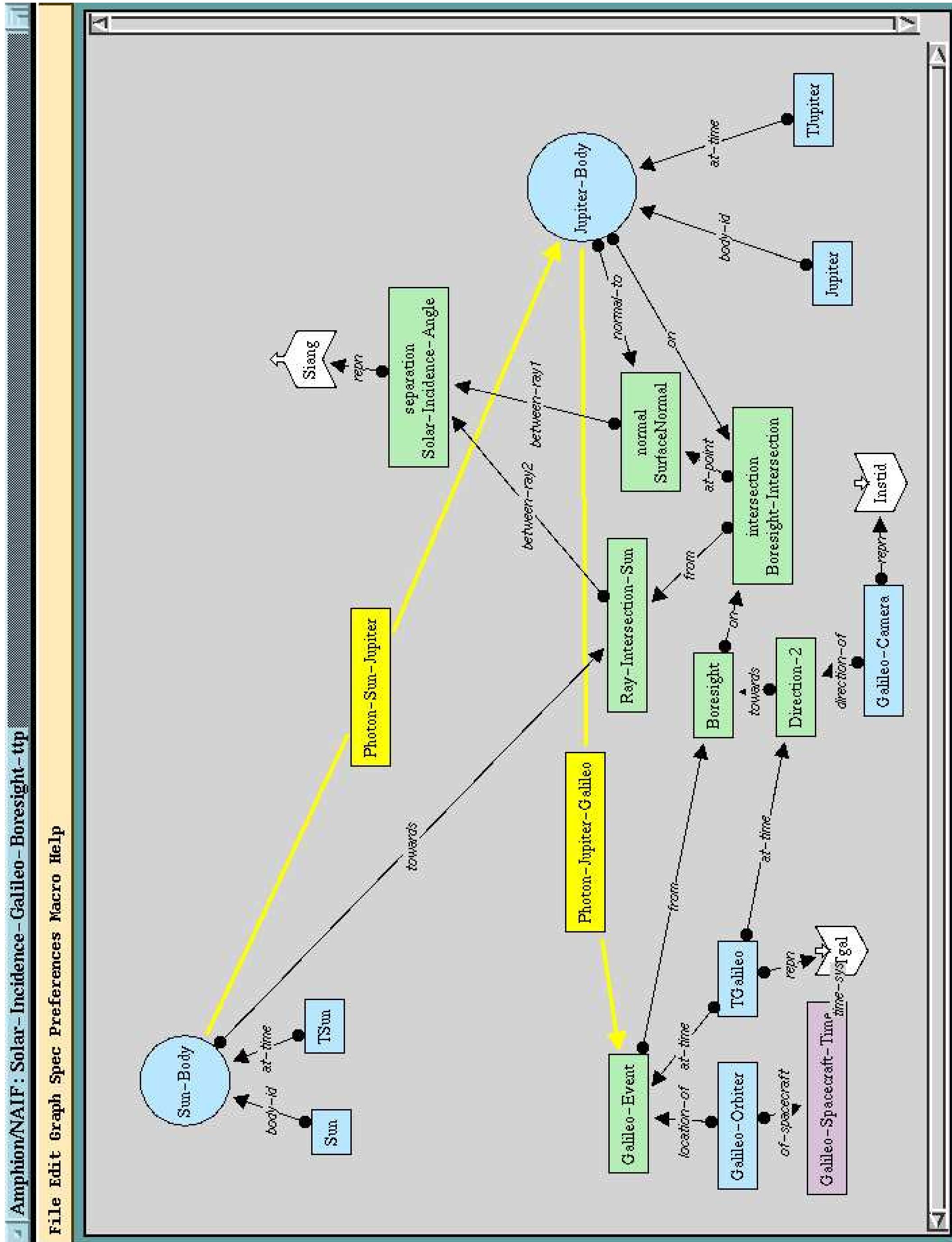

Figure 4.2: Graphical Specification

Chapter 5

Ontologies

The development of the notion of *ontology* has marked recent work in Knowledge Engineering. We look here at the underlying ideas: what are ontologies, and how are they intended to help with the knowledge engineering activity? A good resource is Stanford's *Ontolingua* work

5.1 Ontology – a fluid notion

There are several different characterisations, e.g.

- One of the early definitions: "An ontology defines the basic terms and relations comprising the vocabulary of a topic area as well as the rules for combining terms and relations to define extensions to the vocabulary".
- 'According to [Borst et al., 1997], 'Ontologies are defined as a formal specification of a shared conceptualisation" so that: "Conceptualisation refers to an abstract model of some phenomenon in the world by having identified the relevant concepts of that phenomenon. Explicit means that the type of concepts used, and the constraints on their use are explicitly defined. Formal refers to the fact that the ontology should be machine-readable. Shared refers to the notion that an ontology captures consensual knowledge, that is, it is not primitive to some individual, but accepted by a group"
- Guarino,N. and Giaretta,P., in [Guarino and Giaretta, 1995] give a survey of definitions.
- From [Uschold, 1998]: "An ontology may take a variety of forms, but necessarily it will include a vocabulary of terms, and some specification of their meaning. This includes definitions and an indication of how concepts are inter-related which collectively impose a structure on the domain and constrain the possible interpretations of terms. An ontology is virtually always the manifestation of a shared understanding of a domain that is agreed between a number of agents. Such agreement facilitates accurate and effective communication of meaning, which in turn leads to other benefits such as inter-operability, reuse and sharing."
- From [Fikes and Farquhar, 1999]: "We consider ontologies to be domain theories that specify a domain-specific vocabulary of entities, classes, properties, predicates, and functions, and to be a set of relationships that necessarily hold among those vocabulary terms. Ontologies provide a vocabulary for representing knowledge about a domain and for describing specific situations in a domain."

- Similarly, we could assume that an ontology is a logical theory that constrains the intended models of a logical language.

  This refers to the set of non-logical symbols (predicates and functions) of a logical language used as ”primitives” for a particular representation purpose. An example of this signature is the set of symbols what was called a conceptualisation in KR: on, above, clear, table. An ontology in this case would provide the axioms which constrain the meaning of these predicates, like, for example, $on(X, X)$.

- A fancier view of Ontology occurs with the following proposal:

  > “An ontology is an explicit, partial specification of a conceptualisation that is expressible as a meta-level viewpoint on a set of possible domain theories for the purpose of modular design, redesign and reuse of knowledge-intensive system components.” [Schreiber et al., 1995]

5.2 Ontologies: what for?

A good reference here is “Understanding, Building, And Using Ontologies”, by Nicola Guarino ([Guarino and Giaretta, 1997]).

Knowledge Sharing and Reuse: how can knowledge be shared if it is represented in different ways by different people? Some differences are just cosmetic (“color” or “colour”?), other differences make communication hazardous or impossible.

For a particular domain, it looks feasible to agree on an underlying vocabulary. But what if we want to share and re-use between different domains?

A distinction should be made between the *Knowledge Level* (thought of in terms of the abstract concepts), and the *Symbolic Level* (given in terms of a particular formalisation of a domain, using a particular vocabulary). The intention of recent work in ontologies is to try to work at the Knowledge Level.

Can we get away from the way this knowledge is encoded, and look more directly at the relevance relationship between the knowledge and the problem?

Knowledge Level and granularity An *ontological commitment* is made when the terms in which a domain is to be conceptualised are decided. (Traditionally, philosophy uses this term to indicate a commitment to the *existence* of corresponding entities; we remain agnostic on this.)

Ontological commitments always reflect *particular points of view*.

The aim is to pursue reusability across multiple tasks or methods systematically, even when modeling knowledge related to a single task or method. This way, we hope in getting near to task-independent aspects of a given domain, the more this knowledge can be reused for different tasks.

This can also be thought of in terms of the *granularity* of the domain knowledge. Different expertises (each with a fine-grained conceptualisation) can interact about a problem to which they all contribute, provided they share a (coarse-grained) conceptualisation involving the same basic knowledge related to the domain structure.

This suggests:

- When modelling domain knowledge with a single task in mind, reusability may be pursued by paying the cost of higher granularity.

- The expected gain from the higher generality is that resulting knowledge base, which can become (part of) a sharable ontology.

This suggested research in a discipline like formal ontology appears evident. Say various papers by Guarino on connections between formal ontology, conceptual analysis and knowledge engineering, aiming to establish the foundations of the emerging field of “ontological engineering”.

5.2.1 Some kinds of ontologies

Ontologies can be used in many places, as the following suggests.

- **Application ontologies**: associated with a particular task or problem;
- **domain ontologies**: associated with a problem domain;
- **generic ontologies**: common across problem domains;
- **method ontologies**: associated with problem-solving
- **representation ontologies**: associated with the method of representation itself. See e.g. those defined in Ontolingua’s Frame Ontology ([Gruber, 1993]).

In terms of the KR module, we can associate this with meta-level descriptions of the representations relevant to the previous kinds of ontology. (This is a disputed analysis, however!)

5.2.2 The interaction problem

One problem ontologies should solve is the following:

> Representing knowledge for the purpose of solving some problem is strongly affected by the nature of the problem and the inference strategy to be applied to the problem. ([Bylander and Chandrasekaran, 1988])

However, the interaction problem does not hold to the same extent for all concepts; we can therefore distinguish between an *ontology library*, that contains more or less reusable knowledge across different applications, and an *application ontology*, containing the definitions specific to a particular application. This indexing scheme can surely simplify the construction of application ontologies

However, the important question is the methodology used to update an incomplete ontology library while building a particular application ontology. Typically, we are working on an application, and introduce new terminology. How can we limit the effects of the interaction problem, separating the domain knowledge from the method knowledge?

> To this purpose, the relevance relationship between domain concepts and methods must be captured. With the explicit introduction of a method ontology, this relevance relationship can be represented by means of a “mapping relation” between the application ontology and the method ontology, where the role played by each single concept within a particular method is made explicit. In this way, the effects of the interaction problem can be limited by representing the nature of the interaction, rather than assuming its effects as intrinsic to the concepts being modeled. ([Guarino and Giaretta, 1997])

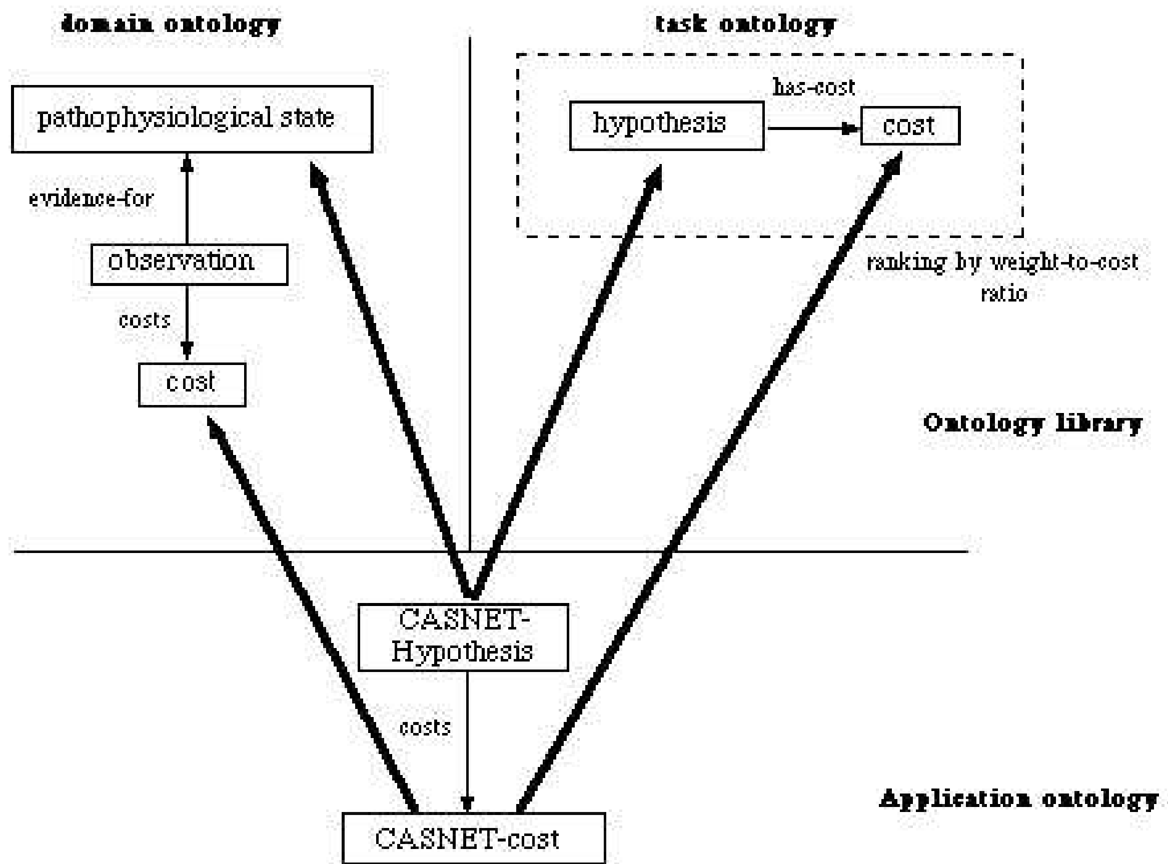

Figure 5.1: Specialising ontologies

Updating ontology: an example

In a slightly different approach, consider the set-up in figure 5.1.

The application ontology is built by specialising *both* the domain and the method ontology.

The concept `CASNET-cost` is the cost of an observation leading to a pathophysiological state, which plays the role of the cost of an hypothesis within the method “ranking by weight-to-cost-ratio”.

It is placed in the application ontology because it plays a specific role in CASNET’s ranking procedure. The ontological requirements of this procedure are represented explicitly in the method ontology, and the CASNET-specific `cost` satisfies the range restriction of the attribute `has-cost` of the concept `hypothesis` belonging to the method ontology.

According to this view, all concepts appearing in the application ontology are specialisations of both the domain ontology and the method ontology. Focusing on the application ontology amounts to highlighting those concepts which are relevant for a particular application, being specialisations of its method(s) and its domain: the application ontology is just a view of the more general ontology.

NB: the left part of the diagram (the domain ontology) can be considered as relatively static, while the bottom and right parts change when the problem solving strategy changes. In this way, the ISA arcs linking the application ontology to the task ontology “can be seen as attributing context-specific semantics to domain knowledge elements” ([Schreiber et al., 1995]). For a given application ontology, it is possible to give a semantic account similar to that we saw for predicate calculus.

5.2.3 How to make use of Ontologies

We want to get round the interaction problem.

- Ontologies should be described at the knowledge level – and sometime their full translation to the symbol level is not even necessary.
- Their purpose is to characterised a conceptualisation, limiting the possible interpretations of the non-logical symbols of a logical language in order to establish consensus about the knowledge described by that language.

The Knowledge Acquisition process should involve *domain analysis*.

Our example uses an explicit representation of task and method knowledge, in order to systematically analyse the knowledge roles played by the domain knowledge within a particular problem solving strategy, resulting in a very simple link between the application ontology and the ontology library, aimed to maximise abstraction, reusability and semantic coherence.

The whole enterprise assumes some common agreement about meaning between various users and sharers of knowledge; despite getting away from the symbolic level, sharing involves common understanding of some representational devices.

We can get help from linguistic resources such as thesauruses; such resources can help with building ontologies by helping to find *generality*; by identifying *ambiguities*; and enforcing *readability and consistency*.

5.3 Using Ontologies: some examples

There are many areas where ontologies are being exploited.

One area where work is ongoing is in *Natural Language*. There typically agreement is sought on

- linguistic categories,
- roles,
- and common-sense objects.

Various projects have built such ontologies for a large part of common (English) vocabulary; eg the CYC project (Lenat)

For another example, Molecular Biology looks for some consistency between various models and data (especially with the advent of the Grid and e-Science). For example, the Riboweb project at Stanford has built an ontology for ribosomes, models, data, reports ...

Here scientists want to describe molecular structure, experimental data, etc. There is a lot of work to encode by hand the relevant literature.

As to be expected, there is a trade off in general between *usability* and *reusability*. Experience suggests that large, sparse ontologies are good for re-use, since they make few commitments, and cover large domains. However,small but detailed ontologies are more immediately useful, since they make commitments specific to the domain of immediate interest.

Example – Documents

An ontology for documents should say what sorts of properties are present, and what related sub-fields are involved.

We can have a hierarchy indicated informally as follows:

Document *with attributes*
– publisher
– title
– date
with subclasses

> **Book** *with attribute*
> – ISBN
> *with subclass*
> **Edited book**
> ...
> **Periodical** *with attributes*
> – Volume
> – Number *with subclass*
> **Journal**
> ...
> **Magazine**
> ...

Ontology for business organisations

For details here, see the Toronto Enterprise Integration Laboratory.

This deals with roles in business, e.g. organisation unit, organisation position, agent, goal, policy, authority, commitment, empowerment, achievable goal, etc.etc.

5.3.1 Features of Ontology

An ontology can be given in terms of classes, frames, etc. Here

- *class* is a distinguished set of objects
- *frame* is any object
- *instance* is a class member
- *slot* is a 2-place relation
- *facet* is a 3-place relation between frame, slot and value
- *constraint* is a statement constraining possible interpretations of a frame.

These all fit into predicate calculus, in the sense that a predicate calculus translation can be carried out in a fairly obvious way.

Ontologies vs OOP

Building an ontology is a bit like building an OOP.
Note however:

- Classes and objects in a program deal with *data structures.*
- Classes and objects in an ontology deal with *features of the world.*

Still, there can be a correspondence between suitable data structures and definitions in an ontology.

Example In OOP, code reuse is encouraged. We can extend a class, eg:

```
class circle {
    int x; int y; int height; }

class ellipse extends circle {
    int width; }
```

Note that, while this is legal (if ugly) in OOP, this does not correspond to the use of subclasses in ontologies; `ellipse` is *not* a subclass of `circle`, in our usual understanding of geometry.

We should also avoid commitment to programming choices in the ontology; e.g., we do *not* want to say that a *name* is a string of a certain maximum length (even if that is how it is implemented). We keep to a more abstract view – a *name* is likely to have many properties, anyway.

We should not assume *any* properties of the implementation of the application (e.g. about relative storage locations of information).

5.3.2 Tools for ontologies

The Ontolingua system provides support for ontology building and sharing.

It provides a suite of tools for constructing and editing ontologies, together with a library on ontologies. It allows the user to give axioms about a given domain in logical form (using the Knowledge Interchange Format, KIF); eg, to encode

> If any two animals are siblings, then there exists someone who is the mother of both of them.

use a predicate calculus representation as follows:

```
(=> (sibling ?sib1 ?sib2)
    (exists (?mom)
        (and (has-mother ?sib1 ?mom)
             (has-mother ?sib2 ?mom)))
```

This makes use of the KIF format already mentioned.

In general, however, the logic representation is generated from the frame-based representation by a uniform mechanism. So, simply editing the frames, slots etc will build up the logical representation.

The aim of Ontolingua is to allow, when possible, systems to be built by reusing components of existing libraries; the definitions there can be extended. Note that different extensions can be incompatible – the aim is that groups of users may work simultaneously on constructing an ontology, with support for spotting and dealing with such incompatibilities.

To this end, several design principles are adopted:

- While the *presentation* is OO-based, the corresponding *representation* is in predicate calculus.
- New objects can be added.
- Existing ontologies can be included, and
- they can be extended (but *not* diminished!).

The representation system itself is built round the so-called *frame ontology*:

Class *with attributes*
– AllInstances
– Arity
– SubclassOf
– SubclassPartition
ClassPartition
Facet
Thing

Users are expected to query the ontology server to determine such things as:

- is a term defined?
- what is the relationship between terms?
- to manipulate the contents of the ontology.

We can use this within browsing and editing tools, ontology analysis tools, knowledge based applications, etc, etc.

5.3.3 Summary

- Ontologies provide a framework for interactive use and reuse of Knowledge Bases.
- Computer tools help us to edit and manage large ontologies, following principles of representation.
- An OO-style presentation to the user corresponds to a logical representation with a semantic reading, and where deduction can be implemented.

Chapter 6

Reasoning Maintenance

6.1 Introduction

Sometimes it is necessary to make guesses or assumptions in one's reasoning, but those guesses or assumptions might turn out to be wrong so that all that followed from them becomes potentially invalid. In what is called a *monotonic* logic, new conclusions can never invalidate what is already known or has been deduced; in *non-monotonic* systems they can. How then, does one keep track of how new deductions depend on earlier information?

The cheap and nasty, but sometimes workable, approach is to attach a 'time tag' - usually this just means a sequence number - to each deduced fact or rule. (Note that a rule here does not have to be a production rule but can be a procedure, a constraint or some other atomic form of knowledge representation.) If a fact/rule added first at time T is, in subsequent reasoning, invalidated, then just erase it and also anything deduced since then. Just restart all reasoning from time $T - 1$, adding just enough to stop following the same line of reasoning as before. This is clearly uneconomic:

```
Time T: add "earth is flat"
Time T+1: add "Edinburgh is the capital of Scotland"
Time T+2: contradict "earth is flat"
```

– it is wasteful to erase the second fact which does not depend on the first. Better approaches to the problem are described below.

In a truth or reasoning maintenance system (alias TMS or RMS), facts and rules are recorded as labels of nodes. The node names are a convenient shorthand for these facts and rules. A node can have zero or more justifications, a justification being an associated record describing how that fact or rule came to be known. Thus, a network of nodes and the interlinking justifications gets built up, like a tower of scaffolding. Some of this tower is rooted in unequivocal facts, but partly it is rooted in potentially invalid assumptions.

There are two main flavours of TMS. In Doyle's formulation [Doyle, 1979], a tower of deductions gets built, but sometimes the reasoning engine may signal that a certain deduction cannot be sustained; that part of the tower must fall. The TMS's job is to decide which piece of the scaffolding might have given way to cause this, and usually there are several assumptions that might take the blame. The TMS blames one and then demolishes such of the scaffolding that depends on it, in such a way that if the blame is later found to have been wrongly assigned all the demolished scaffolding can be resurrected without further work by the reasoning engine. In de Kleer's formulation [de Kleer, 1986], all possible scaffolding is built up from the initial facts and assumptions by forward chaining, so that

the TMS discovers all the deductions that might be supportable by some kind of scaffolding. For each such deduction it records all the sets of assumptions that could be used to support it. This chapter introduces these two reasoning maintenance systems, whilst the module focuses on the latter which is far more popular in use now.

6.2 Doyle's TMS

In this version, a justification may be valid or invalid – usually invalid justifications started life as valid ones. Any node that has at least one valid justification is said to be IN (that is, 'in the set of believed nodes') and those with no valid justification are said to be OUT ('of the set of believed nodes'). In practical systems, OUT nodes that have no use are usually erased after a short while to save space. Note that an OUT node may have a use, for instance it may be mentioned in a valid justification of an IN node, since it may happen that one thing is believed because another is not.

In this type of TMS, there are two kinds of justification:

- *Support List* justifications, of the form

  (SL *INlist OUTlist*)

  where *INlist* and *OUTlist* are each lists of nodes. The justification is valid if all the nodes in the *INlist* are IN, and all those in the *OUTlist* are OUT.

- *Conditional Proof* justifications, of the form

  (CP *node INlist OUTlist*)

  which is valid if *node* is IN whenever all the nodes in *INlist* are IN and all those in *OUTlist* are out.

Here are some observations about these:

- A fact or rule can be made an axiom by giving the node a justification of

  (SL () ())

  so that it is permanently IN.

- Nodes that have a non-empty *OUTlist* are assumptions; there are (at least apparent) conditions under which they could be contradicted.

- CP nodes are useful in supporting deduced rules, e.g.

Node	*Fact/Rule*	*Justifications*
n_1	A	maybe none valid
n_2	B	maybe none valid
n_3	if A then B	(CP n_2 (n_1) ())
n_4	C	maybe none valid
n_5	if C and A then B	(CP n_2 (n_3 n_4) ())

- A reasoning maintenance system is *not* a reasoning system; it is a slave of a reasoning system.

A TMS allows default reasoning. For example, suppose that $F_1 \ldots F_n$ are nodes representing a range of choices, e.g. F_1 represents 'lecture on Monday' and F_7 represents 'lecture on Sunday'. Imagine that node G is a rule that calls for the making of a default choice. Node F_i can be made the default by giving it a justification of the form

(SL (G) (F_1 F_2 $\ldots F_{i-1}$ F_{i+1} $\ldots F_n$))

so F_i will be IN if G is IN and there is no reason to believe in the competitors of F_i (that is, they are OUT). Sometimes it is impossible or inconvenient to enumerate the competing choices like this. Then a node F can be made the default by a simpler expedient. Create, if it doesn't exist, a node to represent the negation of what F represents; call this E. Then F is made the default by giving it the justification

(SL (G) (E))

and giving E various justifications such as

(SL (F_i) ())

for those F_i explicitly known to be competitors of F. Thus, if any F_i is IN, then E will be IN and F will be OUT. Otherwise, F will be IN whenever G is IN and E is OUT. As an example, consider the question of a person's salary:

Node	*Fact*	*Justifications*
n_1	salary=10000	(SL (...) (n_2))
n_2	salary≠10000	(SL (n_3) ()) (SL (n_4) ())
n_3	salary=9314	...
n_4	salary=11206	...

Therefore n_1 will be IN whenever a choice is required, unless any of n_2, n_3 or n_4 is IN for some other reason.

A TMS permits dependency-directed backtracking, that is, it allows the reasoning system to decide to retract some assumption without setting up contradictions caused by continuing to believe any consequence of a retracted assumption. It is worth noting that the TMS itself has no built-in notion of inconsistency; it is up to the reasoning system to detect them and to let the TMS know.

The algorithm typically used to implement this type of TMS is mildly ingenious. Imagine that N is some node representing a fact or rule that the reasoning system finds does not hold, that is, it is a contradiction. The algorithm is:

- Trace the dependencies backward from N (which is currently IN and causing trouble) to find the set of assumptions that support it. Find the maximal ones – those that do not support other assumptions. Suppose that these are nodes $A_1 \ldots A_n$.
- Create a new node NG (called a "no good") to represent the occurrence of the contradiction, e.g. the rule

  not (A_1 and $\ldots$ and A_n)

  Give it justification

  (CP N (A_1 ..$\ldots A_n$) ())

which means it is currently IN.

- Rather than being sophisticated, just pick a culprit at random from the set of maximal assumptions $A_1 \dots A_n$. The aim is to force it OUT, so that N will also be OUT. Suppose that the culprit chosen is A_i.
- A_i is an assumption, so has a non-empty *OUTlist* in a valid justification. Find the set of nodes $D_1 \dots D_k$ which are currently OUT, such that if any one came IN than A_i would be OUT. Pick one, say D_j.
- Give D_j a new valid justification, so it comes IN. A suitable justification would be

  $$(\text{SL } (NG\ A_1 \dots A_{i-1}\ A_{i+1} \dots A_n)\ (D_1 \dots D_{j-1}\ D_{j+1} \dots D_k))$$

  Now D_j will be IN, so A_i will be OUT, so N will be OUT. If any other D comes in, or if any other A goes OUT then D_j will be OUT. This may bring A_i back IN without bringing N back IN, so the arbitrary choice of A_i is not irrevocable. Also, the node NG records the essence of the contradiction, so that the reasoning system will not be troubled with that particular one again, at least in that form.

A small example may give the flavour. Suppose that a reasoning system is trying to solve a timetabling problem, and assumes that some lecture is going to happen at 10am at Forrest Hill:

Node	Fact	Justifications
n_1	time=10am	(SL () (n_2))
n_2	time≠10am	...
n_3	place=FH	(SL () (n_4))
n_4	place=HPSq	...

Currently, the IN set is $\{n_1, n_3\}$. Suppose that the reasoning system now finds the problem:

n_5	(the snag)	(SL (n_1 n_3) ())

That is, some node n_5 turned up, assuming 10am at Forrest Hill, and the reasoning system then found out that this was not possible. The IN set is now $\{n_1, n_3, n_5\}$. The maximal assumptions for n_5 are n_1 and n_3. To clear up the trouble, introduce the "no good"

n_6	not (n_1 and n_3)	(CP n_5 (n_1 n_3) ())

Pick on n_3. It has one OUT node, n_4, which can be brought IN to take it out. Thus, give n_4 the justification

n_4	place=HPSq	(SL (n_6 n_1) ())

The IN set is now $\{n_1, n_4, n_6\}$; the problem has been cured. The existence of n_5 and n_6 stops the reasoning maintenance system from allowing the reasoning system to make the same mistake in future.

In fancier applications, the reasoning system may be used to help the TMS make a more rational choice of culprit than just picking one at random. However, the entire algorithm is not flawless. Suppose that the TMS has been asked to force some newly-discovered contradiction C to be OUT, and the nature of the contradiction is that it depends on, say, an assumption (not C) which is currently IN. Arranging for (not C) to be OUT may cause C to go OUT, by the algorithm, but then this may cause C to come back IN thanks to the rule $C \vee$ (not C) being always IN. Thus the TMS may loop forever. Although it would be easy to build in a check against this simple kind of flaw, there are much more elaborate forms of it that can be very hard to spot.

6.3 de Kleer's ATMS

In de Kleer's version, named assumption-based TMS (ATMS), justifications are always valid. They are made from rules and facts which hold with some sets of assumptions. ATMS does not care about whether any node or assumption is actually believed – there is no question of IN and OUT. Instead, it records only how deductions would depend on any assumptions. That is, the job of ATMS is to record under what sets of assumptions any conclusion holds.

In ATMS, each node has, in addition to the tag showing what it stands for (what you would normally call its label), something else which de Kleer confusingly terms a *label*; this is a set of sets of assumptions. The node follows from any of such sets of assumptions. If a node turns out to be contradictory, according to the reasoning system, then the ATMS notes that all the sets of assumptions labelling it lead to a contradiction. The convenient way to do this noting is to have a node which stands for 'false', and to add any such contradictory sets to its label. For each node, a check is made such that none of its sets of assumptions is a superset of any other; if so, it should be deleted. The ATMS does this to ensure that no superfluous assumption appears anywhere in a set within any label.

To have a look at the idea of how ATMS functions, suppose that early on in the deductive process (performed by a reasoning engine) there have been nodes for assumptions $a_1 \ldots a_5$, and that there are already two nodes A and B, with labels as follows

A	$\{a_1, a_2\}\ \{a_2, a_5\}$
B	$\{a_1\}\ \{a_2, a_3\}\ \{a_4\}$

For instance, the label of node B means that B holds if a_1 does, or if a_2 and a_3 do, or if a_4 does. Suppose also that the only contradiction so far discovered is that a_4 and a_5 together lead to an inconsistency. This means that the label of the specific 'false' node currently contains just one set of contradictory assumptions: $\{a_4, a_5\}$.

Now, the reasoner wants to add another justification, that A and B imply C. The ATMS records the justification, and from the labels for A and B creates a new label for C as follows (if C is new, a node for it is first created):

- Find the collection of pairwise unions of sets of assumptions, with one set per pair from each label (akin to the cross-product of the labels of A and B); this is

  (1) $\{a_1, a_2\}$
  (2) $\{a_1, a_2, a_3\}$
  (3) $\{a_1, a_2, a_4\}$
  (4) $\{a_1, a_2, a_5\}$
  (5) $\{a_2, a_3, a_5\}$
  (6) $\{a_2, a_4, a_5\}$

- Remove any which has another as a subset. In this case, sets (2), (3) and (4) each have (1) as a subset. This step removes any superfluous dependence on assumptions. Thus only sets (1), (5) and (6) remain.

- Remove any set which has a contradictory set of assumptions as a subset of it. Since $\{a_4, a_5\}$ is already known to be contradictory, this disposes of set (6), leaving sets (1) and (5) to form the new label for C.

- If C is new, this becomes its label. If not, this label is blended with the previous label: take the union of the two labels, delete any set in this union which has another member in the union as a subset of it. Since the new and old labels were already consistent there is no need to go looking for sets which have contradictory subsets, this time.
- If the whole process changes the label for C, and recorded justifications show that other nodes depended on C, then the label changes have to be propagated forward to those other nodes, and on from them, and so on.

In the above illustration, if, at a certain stage, C is found to be a nogood by the reasoning system (that is, every set of assumptions labelling it indicates a contradiction), then each member of its label is added to the label of the specific node standing for 'false', and all such members, and their supersets, are removed from the label of every other node. Any set of (inconsistent) assumptions being a superset of another within the 'false' node's label itself is also removed.

ATMS has two advantages. First, all the consequences of a set of assumptions can be explored together; no backtracking is involved, and the system is not striving to maintain one mutually consistent set of assumptions as in Doyle's TMS. This suits certain kinds of application. Second, it can be very efficiently implemented, since the main operations involved are set operations such as union and subset checking. If sets are represented as bit strings these kinds of operations can be performed directly by hardware. The obvious disadvantage is that the system is driven by forward-chaining reasoning, so there is no natural progression towards a particular desired goal.

There is considerably more to ATMS than this; this section has only covered the basic idea. For example, in some applications it might be desirable to constrain the ATMS to considering only those sets of assumptions which contain at least one, or perhaps exactly one, of a given set (of particular assumptions). This calls for adding new justifications automatically whenever the ATMS throws up a set that violates such constraints – this is much more economical than adding all the justifications that these constraints entail right from the beginning. See various papers by de Kleer in the *AI Journal* for further details.

6.4 A Caution

In practice, whatever the flavour of TMS, there are further complications to consider. For example, consider the following rule:

```
if X is burning
and X is on Y
and <..various physical conditions
       indicating Y will char..>
then Y is charred
```

Even when 'X is burning' is no longer true, the conclusion 'Y is charred' will be true. However, this conclusion might be marked as OUT in Doyle's TMS or retracted in de Kleer's ATMS along with the retraction of 'X is burning'. In this case the trouble really lies with the rule rather than the TMS, but the example does suggest that there are special difficulties in handling time-based rules in which the effect lasts even when the cause does not. Some solutions to this can be taken from recent work on automated planning and/or temporal reasoning. They are not described here, however.

Chapter 7

Model-Based Reasoning Systems

7.1 Introduction

One of the recent trends in developing intelligent problem solvers has been the building of *model-based reasoning* systems. Imagine that you have built a knowledge-based system to diagnose faults in coin-in-the-slot telephones and to suggest cures. If the symptom is that your money is not returned after an unsuccessful call, bash the telephone sharply on its left-hand side at a point about two inches lower than the level of the coin slot. However, when BT[1] replaces these machines with slightly more modern coin-in-the-slot boxes you would need to amend the knowledge-based system. It would be nicer if the system could reason out for itself where you should hit the box, by considering a model of the mechanism. This, crudely, is the idea behind the model-based systems.

The term 'model' may refer to the description of the structure and behaviour of a physical mechanism. It could equally be a description of a social or business process, or that of biochemical balances within the human body. The common idea behind all such descriptions is that a model represents a simulation of something, a simulation which is faithful and complete at some sensible level of detail. For example, a model of how an electric refrigerator works need not be at the level of the physical layout of the parts. Rather, it might be in terms of how an electric motor compresses a gas, whose later expansion causes the gas to cool and the cooled gas then circulates through pipes within the refrigerator drawing heat out of it by conductivity through the walls of the pipes, and how convective cooling cools those parts not close to the internal pipes.

Diagnosis has been, and still remains, the largest application category of most intelligent application systems. This chapter is, therefore, focused on discussions of model-based systems which perform diagnostic tasks. Although it addresses the issues of finding faults in physical systems the techniques introduced can be applied to other domains.

To determine why a physical system has not worked correctly compared to its design intention, it is useful to know how it was supposed to work in the first place. Reflecting this viewpoint, the basic paradigm of model-based diagnosis can be best seen as the interaction of observations, obtained from the physical system under diagnosis, and predictions, generated from the system's model (which is usually qualitative). Observations indicate what the system is actually doing whilst predictions indicate what it is intended to do. Thus, this approach essentially depends on the use of an explicit structured model for behaviour prediction. Whenever there exists a difference, or a discrepancy, between observations and predictions, it is presumed that the system is not working correctly and certain faults have occurred within the system. Given detected discrepancies a model-based diagnostic

[1] BT = British Telecom

system is set to identify, ultimately, which of the system components could have failed and led to such discrepancies.

It is not unusual for there to be multiple faults simultaneously. One of the drawbacks of early rule-based approaches was an unwarranted assumption that any one symptom had a single cause, so when one cause had been diagnosed all the symptoms it accounted for could be ignored when trying to account for the remaining symptoms. Many model-based approaches, however, provide a reasonably successful method for trying to handle multiple faults. A representative of these, the General Diagnostic Engine (GDE) [deKleer and Williams, 1987] is introduced in the next section, in terms of isolating multiple fault candidates efficiently via the use of an ATMS. GDE also provides a mechanism for differentiating among the set of possible candidates, which incorporates probabilities and information theory into a unified framework. An introduction to this second part of GDE is given in section 7.3.

Since its birth, GDE has been widely applied and tested in a variety of problem domains, and its limitations as well as its strengths in performing diagnosis have been extensively studied in the literature. As a result, many improvements and extensions have been made to GDE (see [W. Hamscher and de Kleer, 1992]). Among them, one of the best known is the GDE+ system [Struss and Dressler, 1989]. An overview of this system is presented in section 7.4, where another extended system, Sherlock [de Kleer and Williams, 1989] is also outlined to show the diversity of different modifications possible.

7.2 Postulation of Multiple Faults

GDE is intended to isolate or locate multiple simultaneous faults in a device consisting of interconnected components. It makes a number of important assumptions:

- Faults are in components rather than in any interconnections (though interconnections could be represented as components themselves).
- The device representation is faithful; there are for instance no interconnections in the real device which are not also present in the model.
- Faults are not intermittent.

To illustrate the kind of problem it can deal with, consider a much-used example system as shown in figure 7.1.

According to the design of this device, the three multipliers, M_1, M_2 and M_3 are each supposed to multiply their two inputs, and the two adders, A_1 and A_2 are supposed to add their two inputs. Thus, given the inputs:

$$A = 3, B = 2, C = 2, D = 3, E = 3$$

to a working version the result will be:

$$F = 12, G = 12$$

However, if $F = 10$ instead it might be suggested that the problem was that M_1 was only outputting 4 instead of 6. The trouble is that this is not the only diagnosis, even though it makes good sense to consider it first. The difficulty then is to find the others. Another might be that M_2 was outputting 4, A_1 was working whilst A_2 was faulty but just compensating for the error made by M_2. Another could be that M_2 was erroneously outputting 4 and M_3 was erroneously outputting 8, but the two adders were working properly. Of course it is even possible that every component is faulty!

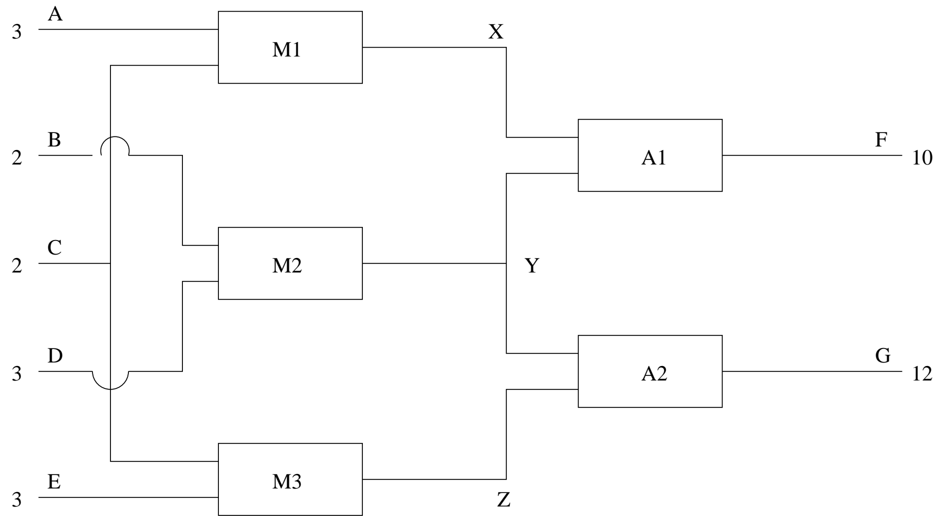

Figure 7.1: A familiar circuit

The idea behind GDE is to treat the initial observations:

$$A = 3, B = 2, C = 2, D = 3, E = 3, \quad F = 10, G = 12$$

as facts and to treat statements such as 'M_1 is working correctly' as assumptions (with a set of assumptions termed an environment for easy reference). An ATMS can then be used to keep track of what deductions depend on what environments. For example, for notational simplicity (and hopefully without causing confusion), let M_1 stand for the assumption that the multiplier M_1 is working, let A_2 stand for the assumption that the adder A_2 is working and so on. Thus, the following extra 'nodes' can be generated under the assumptions given in $\{\ldots\}$, though not necessarily in this order:

n1: $X = 6\ \{M_1\}$

n2: $Y = 6\ \{M_2\}$

n3: $Z = 6\ \{M_3\}$

n4: $Z = 6\ \{M_2, A_2\}$ (from **n2** and $G = 12$)

n5: $X = 4\ \{M_2, A_1\}$ (from $F = 10$ and **n2**)

Note that already **n5** conflicts with **n1**, so it cannot be the case that all of M_1, M_2 and A_1 are working. From this, GDE deduces that at least one of the components M_1, M_2, and A_1 is faulty and introduces the notion of *minimal candidates*: $\{M_1\}, \{M_2\}, \{A_1\}$. The process can continue:

n6: $Y = 4\ \{M_1, A_1\}$ (from $F = 10$ and **n1**)

This produces the same *conflict* as before. The process continues:

n7: $Z = 8\ \{M_1, A_1, A_2\}$ (from $G = 12$ and **n6**)

Now **n7** disagrees with **n3**, so it cannot be the case that all of M_1, A_1, A_2 and M_3 are working. This is a different conflict. Given the following two assertions:

- By **n5** and **n1** or **n6** and **n2**, at least one of $\{M_1, M_2, A_1\}$ is faulty.
- By **n7** and **n3**, at least one of $\{M_1, M_3, A_1, A_2\}$ is faulty.

GDE deduces that at least one of the following sets of components is faulty, and treats each of them as an updated minimal candidate:

$$\{A_1\}, \{M_1\}, \{A_2, M_2\}, \{M_2, M_3\}$$

The reason behind this deduction can be best shown by examples. The set $\{A_1\}$ is considered a minimal candidate because A_1 belongs to both conflicts in the above two assertions, if it is faulty both assertions are explained. Similarly, the set $\{A_2, M_2\}$ is considered a minimal candidate because if A_2 and M_2 are faulty simultaneously they jointly explain both assertions. However, unlike A_1, one and only one of $\{A_2, M_2\}$ being faulty does not form a consistent explanation for these two assertions.

For this simple device, given the above minimal candidates, it should be clear that attention is to focus on M_1 or A_1. This suggests that the value at X should be measured, in order to better isolate the possible faults. The measured value of X becomes a new fact, and the process continues.

Electrical engineers will realise that this whole process is a bit suspect. To start with, a common kind of fault is for there to be an open (broken) connection somewhere or for there to be a spurious extra connection caused by faulty manufacture; the underlying assumption that any fault lies within a component is too restrictive. Further, even if the fault is in a component it can happen that components fail in more interesting ways than just producing the wrong output for given inputs. For example, if M_3 fails it may drag input C down to 0, causing an input to M_1 to be 0 too. This could perhaps be handled within GDE using a more elaborate scheme of representation and assumptions. A more serious problem is that the number of deductions from assumptions and other intermediate deductions is going to proliferate exponentially as the circuits get more complicated. GDE brings in component failure probabilities in an attempt to deal with this, by ignoring any combination of faults which seems too unlikely to pursue in the meantime [de Kleer, 1990].

7.3 Choosing Measurements for Candidate Discrimination

Using the above method for fault candidate generation, or indeed one of the many other model-based techniques, can lead to a quite large number of postulated faults. In order to reduce the remaining candidates to make the diagnostic result practically more useful, a diagnostic system may use further measurements to differentiate amongst the candidates. GDE offers a method for choosing the best measurements which would most effectively distinguish between the remaining candidates. A best measurement in this context is one which will, on average, allow the discovery of the actual fault(s) in the physical system using a minimum number of subsequent measurements [de Kleer and Williams, 1989].

Before showing how to choose the best measurements, it is useful to examine what consequences a possible measurement may bring to the set of generated candidates. Recall that, in GDE, each model-based prediction is treated as a fact and stored as a node in the ATMS with the set of minimal environments supporting it as its label. Here, each environment is a set of assumptions made to support a partial, behavioural description of the physical system. For instance, a value v of a variable x, which is predicted under the environments $e_1, e_2, ..., e_m$, can be represented by

$$< x = v, \{e_1, e_2, ..., e_m\} >$$

In general, a variable x may have more than one predicted value, each having a set of supporting environments. If x is measured to be v then the supporting environments of any value different from it are necessarily conflicts, since they did not lead x to taking v as its true value. If, however, the measurement is not equal to any of the predicted values of x, then every supporting environment for each predicted value of x must be a conflict.

Consider the electrical device shown in figure 7.1 again. Recall that, given the facts that $A = 3, B = 2, C = 2, D = 3, E = 3, F = 10, G = 12$, and the identified conflicting environments that led to contradictory predicted and/or measured values for individual variables (e.g. environment $\{A_1, M_2\}$ or $\{A_1, A_2, M_3\}$ led to $X = 4$, while environment $\{M_1\}$ resulted in $X = 6$), GDE deduces the following minimal candidates, meaning that at least one of these sets of components must be faulty:

$$\{A_1\}, \{M_1\}, \{A_2, M_2\}, \{M_2, M_3\}$$

Suppose that a new measurement is made at point X. This measurement will result in three possible outcomes:

- $X = 4$, in this case environments $\{A_1, M_2\}$ and $\{A_1, A_2, M_3\}$ do not lead to a contradictory value for X, but $\{M_1\}$ does and forms a conflict (as it supports $X = 6$ rather than $X = 4$), the minimal candidate now becomes $\{M_1\}$.
- $X = 6$, in this case $\{A_1, M_2\}$ and $\{A_1, A_2, M_3\}$ are conflicts and the new minimal candidates are $\{A_1\}$, $\{M_2, M_3\}$ and $\{A_2, M_2\}$.
- $X \neq 4$ and $X \neq 6$, in this case $\{A_1, M_2\}$, $\{A_1, A_2, M_3\}$ and $\{M_1\}$ are conflicts and the minimal candidates are $\{A_1, M_1\}$, $\{M_1, M_2, M_3\}$ and $\{A_2, M_1, M_2\}$.

The above example illustrated the effect of an appropriate new measurement upon the discrimination between possible fault candidates. For this example, measurement at X is selected to explain the potential consequences since, intuitively, measuring X is likely to produce the most useful information to further isolate the faults. This is because the two (more probable) singleton candidates $\{A_1\}$ and $\{M_1\}$ are only differentiable when such a measurement is made. For this toy device, it is feasible to recognise this by looking at the circuit diagram. However, when the system structure becomes more complex, how can a best measurement point be automatically chosen?

GDE gives an answer to this question by adopting a one-step lookahead strategy based on Shannon's information theory. This strategy requires the consequences of each possible measurement to be analysed with respect to a given set of (minimal) fault candidates, in order to determine which measurement to actually make next. The following entropy function is used as the basis for the analysis of the possible consequences of *hypothesised* measurements:

$$-\sum_i p_i log(p_i)$$

where p_i denotes the probability that the i-th (remaining) candidate would be the actual culprit given the hypothesised measurement outcome.

This entropy function has several important properties. In particular, if every candidate is equally likely then it will reach a maximum value, indicating that little information is available for differentiating among the candidates. If, however, one candidate is more likely than the rest this function will approach its minimum. Thus, the best next measurement should allow the minimisation of the expected entropy of candidate probabilities. From this, and by substantial algebraic manipulations,

it can be derived that the best measurement is the one which minimises (over all possible measuring points x_i within the model of the physical system):

$$\sum_{k=1}^{m}[p(S_{ik}) + \frac{p(U_i)}{m}]log[p(S_{ik}) + \frac{p(U_i)}{m}] - p(U_i)log\frac{1}{m}$$

Where m is the number of possible values for variable x_i, S_{ik} is the set of candidates in which x_i must be of a certain value v_{ik} (equivalently, the set of candidates which will be eliminated if x_i is measured not to be v_{ik}), and U_i is the set of candidates which will not be eliminated no matter what value is measured for x_i (equivalently, the set of candidates which do not predict a value for x_i).

This method relies on the availability of failure probabilities for system components, in order to calculate the probabilities $p(S_{ik})$ and $p(U_i)$. However, in many cases this information is unknown. In addition, this method can be computationally very expensive if there are many points in a device that could be measured and/or if each variable may take on many possible values. To avoid these problems, de Kleer proposed a considerably simplified version of this minimum entropy technique [de Kleer, 1990], assuming that all components fail independently with equal probability (which is somewhat suspect since dependent failures can be very common in some domains and certain components may fail more often than others), and that components fail with a very small probability.

Supported with strong assumptions, the simplified version states that the best measurement is the one which minimises:

$$\sum_{k} c_{ik}log(c_{ik})$$

where c_{ik} is the number of those remaining fault candidates which have the minimum cardinality N *and* which predict that the value of variable x_i is v_{ik}. Here, N being the minimum cardinality means that all candidates with less than N components have been exonerated from suspicion.

What this method implies is that, to determine the best next measurement, it is only necessary to consider the fault candidates of minimum cardinality, i.e. those candidates containing the least number of components. In fact, given the minimum cardinality being N, every candidate with less than N possibly faulty components has been eliminated and every candidate having more than N components has a negligible probability in comparison with any candidate of cardinality N. Thus, this one-step lookahead strategy always first proposes measurements which will differentiate amongst single-component fault candidates (with $N = 1$ being the smallest cardinality possible). If all single-component candidates are eliminated, it proposes measuring points which differentiate amongst double-fault candidates, and so on.

It is important to notice that calculations involved in this method are very simple. More importantly, no actual probability values are required to perform such calculations. Only the counts of the number of candidate faulty components that support each hypothesised measurement outcome are needed. Therefore, this method can be applied even when the exact probability values are unknown, provided that assumed conditions are satisfied such that faults occur independently and with a very small, equal probability.

Now, go back to the example of figure 7.1. Given the inputs, $A = 3, B = 2, C = 2, D = 3, E = 3$, and the measured outputs $F = 10$ and $G = 12$, there are two single-component fault candidates: $\{M_1\}$ and $\{A_1\}$. In other words, $\{M_1\}$ and $\{A_1\}$ are the remaining candidates which have the minimum cardinality of $N = 1$. Possible measurement outcomes divide these candidates in the way as summarised in table 7.1. For instance, as discussed earlier, if the hypothesised measurement results in $X = 4$ then M_1 is faulty (since it would otherwise lead to $X = 6$) and A_1 is not (because it is part of both environments $\{A_1, M_2\}$ and $\{A_1, A_2, M_3\}$, which support the predicted value $X = 4$). Note

that, although it is possible for, say, Y to take a predicted value of 4 rather than 6, it could not have a measured value of 4 given the *present* consideration of, at least, one of A_1 and M_1 being faulty. In fact, the supporting environment of the prediction $Y = 4$ is $\{A_1, M_1\}$, an obvious contradiction (unless both single-component candidates have been eliminated). For similar reasons, the predicted value $Z = 8$ under the environment $\{A_1, A_2, M_1\}$ is excluded as a hypothesised measurement from this table.

Hypothesised Measurement	Candidates
$X = 4$	$\{M_1\}$
$X = 6$	$\{A_1\}$
$Y = 6$	$\{A_1\}, \{M_1\}$
$Z = 6$	$\{A_1\}, \{M_1\}$

Table 7.1: Fault candidates divided by possible measured values

Given this table, it is easy to count that at point X, the number of single-component candidates which predict either of X's two possible values is 1, and that at point Y or Z, the number of single-component candidates which predict Y or Z's only possible value is 2. This leads to the following:

$$X: \quad 1log1 + 1log1 = 0$$
$$Y: \quad 2log2 = 1.4$$
$$Z: \quad 2log2 = 1.4$$

where the base for the logarithm is set to e; a different base may be used, if preferred, as the logarithm is a monotonic function and the use of a different base will not alter the order of the results. Thus, these results show that X is the best next measuring point for candidate discrimination. This matches well with common-sense intuition.

7.4 Extending GDE with Fault Models

GDE uses only one fixed model to predict the normal behaviour of the physical system under diagnosis. It determines a component to be faulty if a retraction of its corresponding correctness assumption makes the predicted behaviour consistent with the observations.

GDE does not make use of knowledge about how components may behave when they fail. As a result, the fault candidates isolated by GDE are logically consistent with the observations, but may well be physically implausible or even wrong. A typical example for this involves a simple electrical circuit consisting of a battery and several bulbs wired in parallel with some being lit and some not. Intuitive diagnoses in such a situation are those bulbs which are not lit. However, in addition to finding these faulty bulbs, GDE may generate many other physically impossible, though logically consistent, fault candidates, unless further measurements are made. For instance, a resulting candidate, consisting of the battery and lit bulbs, might suggest that the battery is broken and provides no power (therefore, some bulbs are not lit), and those lit bulbs are faulty since they produce light without power supply. This is a rather weird explanation, of course.

Fortunately, the GDE system has been extended to exploit fault models (when available) so as to identify the actual misbehaviour of the faulty components and, thus, to limit the occurrence of implausible or incorrect diagnoses. A good example of such extensions is the GDE+ system [Struss and Dressler, 1989], in which each physical component is characterised by the description

of its normal behaviour plus a set of possible descriptions of fault behaviour and in which each component is assumed not to fail outside these fault descriptions.

More concretely, within GDE+, a component, C, is allowed to fail in a certain number of ways, and each possible way is explicitly described by a fault model of the behaviour of that component. Each fault model is, in turn, represented by two nodes in the ATMS: C_i and $\neg C_i$, with C_i denoting the i-th potential fault behaviour of C and $\neg C_i$ denoting the negation of C_i. At any time, there must be one and only one of C_i and $\neg C_i$ which is consistent with the observations, with the other contradicting the observations. A fault model of a component, for instance the i-th model C_i of C, is said to contradict an observation, if there is a (possibly empty) set of other components such that C_i cannot be consistently joined with any combination of the normal and fault representations of those components when presented with that observation. The introduction of fault models in GDE+ supports the following *inference rule*: If each of the known possible faults of a component contradicts the observations, then the component is not faulty and should be exonerated from suspicion.

Consider a simple circuit as shown in figure 7.2, where a battery S is connected, in parallel, to three bulbs B_1, B_2 and B_3 of the same type via pieces of wire $W_i, i = 1, 2, ..., 6$. Observations indicate that only B_3 is lit whilst B_1 and B_2 are off. The problem is given such observations to determine what went wrong in the circuit.

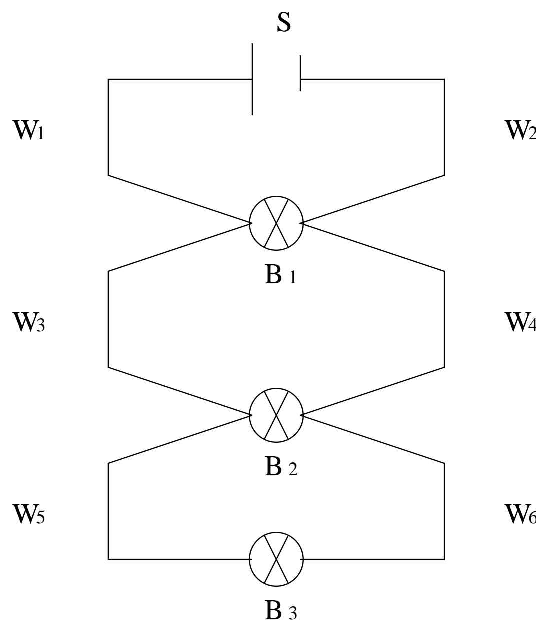

Figure 7.2: A circuit of a battery and three bulbs

Using GDE's candidate generation method, for this simple diagnostic problem, a total of 22 minimal candidates are generated, including $\{B_1, B_2\}$, $\{S, B_3\}$, $\{S, W_5\}$, $\{W_1, B_3\}$, $\{S, W_6\}$, $\{W_1, W_5\}$, $\{W_1, W_6\}$, $\{W_2, W_5\}$, $\{W_2, W_6\}$, $\{W_2, B_3\}$, ... (Note that these candidates are not necessarily generated in this order.) Although GDE does return the plausible diagnosis, i.e. the first candidate, $\{B_1, B_2\}$, in the above list, it also generates many physically implausible diagnoses. Here, the first one is plausible because B_3 being lit indicates that the battery does supply sufficient voltage, and a bulb that is off despite this fact is considered to be broken. However, all the other candidates are not acceptable diagnoses, since they imply either a wire producing voltage out of nothing or a bulb being

lit without voltage. Only further measurements would enable GDE itself to rule out such implausible diagnoses.

This problem can be solved by GDE+ without requiring further measurements, however. Let each type of component have exactly one fault model, informally as listed in table 7.2, where $i = 1, 2, 3$ and $j = 1, ..., 6$. The only fault model for each bulb, and hence the one for B_3, directly contradicts

Fault	Predicted Value
battery empty	voltage(S) $= 0$
bulb broken	light(B_i) $=$ off
wire broken	resistance(W_j) $= \infty$

Table 7.2: Fault models for the battery-bulb example

the observation that shows B_3 being lit. B_3 is thus inferred to be not faulty and removed from any minimal candidates (e.g. $\{S, B_3\}$ and $\{W_1, B_3\}$) which would otherwise contain it. This is because GDE+ works based on the strict, though retractable, assumption that a component fails in no other way than those specified in its fault models. Any minimal candidate with part of it being deleted (e.g. $\{S\}$ or $\{W_1\}$) will no longer be regarded as a candidate, since whatever remain in that set of inconsistent assumptions are not sufficient to explain all the observed discrepancies (the very reason that the original candidate was referred to as a minimal candidate).

The elimination of B_3 being a possible faulty component means that there is current passing through it given the fact that it is lit. Having current at B_3 contradicts the only fault behaviour of each broken wire and the empty battery. Therefore, candidates involving pieces of broken wire and/or empty battery are also eliminated. In so doing, GDE+ is able to conclude that the only minimal candidate is $\{B_1, B_2\}$, as desired.

The price paid for this important capability of GDE+ for candidate elimination is the significant increase of the complexity of the original GDE system. GDE+ worsens the combinatorial problem of GDE: There are already $O(2^n)$ possible diagnoses for a physical system of n components without fault models in GDE, with k possible fault descriptions per component, there may be $O(k^n)$ possible diagnoses for GDE+ to examine. Therefore, there exists a potential risk in utilising GDE+ of creating a vast number of combinations of normal and fault component models. In general, certain control strategies are necessary to attain a trade-off between its discrimination power and efficiency.

Another representative of extensions made to GDE is the Sherlock system [de Kleer and Williams, 1989]. Sherlock views the central task of diagnosis as identifying the behavioural description of each system component which is consistent with the observations. Whilst allowing every component to have different behavioural descriptions, normal or faulty, it maintains the basic intuition behind the GDE system: To determine faulty components without necessarily knowing how they fail. It ensures this by assigning each component with an unknown mode.

As with GDE+, however, each component now has multiple models and each model predicts a different behaviour which must be considered. There are far more behaviours to reason about and the useful concepts of minimal conflicts and minimal candidates in GDE becomes difficult to handle (if not virtually meaningless). To circumvent this problem, Sherlock resorts to probabilistic information about the likelihood of each behavioural description, introducing the most probable fault behaviours to components in order to restrict candidate generation. In particular, to keep the combinatorics in check, Sherlock performs best-first search to generate the diagnoses in decreasing likelihood and utilises heuristics to decide when enough diagnoses have been found. In this way, Sherlock avoids

considering most of the possible candidates that GDE has to deal with and performs significantly better than GDE.

Chapter 8

Agent Systems

8.1 Agent Oriented Programming

A good survey of recent thoughts on this topic is in [Jennings, 1999].

We'll look at a proposal by Shoham for a proposal for a style of programming based on agents. See [Shoham, 1993] for more details.

8.1.1 Agents and Objects

Shoham contrasts his proposal with Object Oriented Programming. Agent Oriented Programming (AOP) is a specialisation of OOP, where the modules (here called *agents*) come with a notion of state given in mentalistic terms (beliefs, decisions etc). The main computational mechanism is *message passing*, and Shoham makes use of speech act theory to classify the messages according to their role (informing, requesting, offering, and so on).

	OOP	AOP
basic unit	object	agent
state parameters	unconstrained	beliefs, commitments, ...
process of computation	msg passing & response	msg passing & response
types of message	unconstrained	inform, request, offer ...
constraints on methods	none	honesty, consistency

Figure 8.1: Comparison of OOP and AOP

8.1.2 Components of the system

There are three main components:

- An agent theory: this is a formal language for describing mental states, and how the states change. This uses several modal operators.
- An agent language: this is an interpreted programming language in which agents can be programmed. The semantics for this language is required to be faithful to the semantics of the agent theory.

- An "agentifier" – some way of turning devices into agents.

Shoham says little about the last component. We will concentrate of the first two.

8.1.3 The agent theory

Here we want a theory of mental states that we can use to say how agents should behave.

Shoham organises the theory around three modalities: *belief*, *obligation* and *capability*. (The notion of *decision* or *choice* then corresponds to obligation of an agent towards itself.)

The model here is that the agents decides what to do on the basis of *beliefs* about the state of the world, past present or future, about the mental states of other agents, and on the capabilities of the agents. An agent will not try to do something it believes it is incapable of. The decisions should also be consistent – the agent should not decide to be in two places at the same time, for example.

Shoham has chosen here not to represent explicitly *motivation*, though this will be an important part of a more developed agent theory.

8.1.4 A language for time, belief, obligation and capability

Shoham adopts a standard approach to reasoning about time: a distinguished extra argument to a predicate is used to denote the point of time where the statement is taken to be true (the extra argument is here written as a superscript). So

$$\mathtt{boiling(kettle)}^{\mathtt{t}}$$

means that the kettle is boiling at time $\mathtt{t}$.

Actions are treated here like facts, associated with times. This means that actions are modeled as instantaneous.

Beliefs are represented by a three place modal operator, with arguments $\mathtt{a}, \mathtt{t}, \mathtt{F}$ corresponding to agent, time, and belief, written

$$\mathtt{B}^{\mathtt{t}}_{\mathtt{a}}\mathtt{F}.$$

These can be nested –

$$\mathtt{B}^{3}_{\mathtt{a}}\mathtt{B}^{4}_{\mathtt{b}}\mathtt{boiling(kettle)}^{2}$$

means that at time 3, agent $\mathtt{a}$ believes that at time 4, agent $\mathtt{b}$ will believe that the kettle boiled at time 2.

Obligation This modality has four arguments: $\mathtt{a}, \mathtt{b}, \mathtt{t}, \mathtt{F}$ expressing an obligation at time $\mathtt{t}$ of agent $\mathtt{a}$ toward agent $\mathtt{b}$ about statement $\mathtt{F}$. The obligation is about a statement, and not about an action, as we will see. This is written

$$\mathtt{OBL}^{\mathtt{t}}_{\mathtt{a,b}}\mathtt{F}.$$

Decision can now be treated just as an obligation of an agent towards itself:

$$\mathtt{DEC}^{\mathtt{t}}_{\mathtt{a}}\mathtt{F} =_{def} \mathtt{OBL}^{\mathtt{t}}_{\mathtt{a,a}}\mathtt{F}$$

For example, we can think of the decision to go to a concert this evening as the forming of an obligation to oneself to be at the concert at the appropriate time, circumstances permitting.

Capability is not necessarily a mental notion, since it is more directly to do with the structure and embodiment of a given agent. Here a three place modality is used, with arguments $\mathtt{a}, \mathtt{t}, \mathtt{F}$ saying that at time $\mathtt{t}$ agent $\mathtt{a}$ is capable of $\mathtt{F}$. Thus

$$\mathtt{CAN}_{\mathsf{a}}^{5}\mathtt{boiling}(\mathtt{kettle})^{8}$$

means that agent $\mathtt{a}$ is capable at time 5 of getting the kettle to boil at time 8. This capability might no longer be there at a later time.

A version of capability that is linked to the immediate time of an assertion is also defined. Define $time(\mathtt{F})$ for a formula $\mathtt{F}$ as returning the outermost time mentioned in $\mathtt{F}$.

Thus $time(\mathtt{boiling}(\mathtt{kettle})^{\mathtt{t}}) = \mathtt{t}$. Then we can take

$$\mathtt{ABLE}_{\mathsf{a}}\mathtt{F} =_{def} \mathtt{CAN}_{\mathsf{a}}^{time(\mathtt{F})}\mathtt{F}$$

so, for example,

$$\mathtt{ABLE}_{\mathsf{a}}\mathtt{boiling}(\mathtt{kettle})^{8} = \mathtt{CAN}_{\mathsf{a}}^{8}\mathtt{boiling}(\mathtt{kettle})^{8}.$$

8.1.5 Properties of the language

What should we assume about the language and logic we have seen so far?

We need to know how these various modalities interact, and what we can conclude using them. Shoham gives a fairly weak theory (that is, the set of statements we can deduce is smaller than in a stronger theory), but this is still enough to be able to design an interpreter.

Internal consistency: assume that the beliefs and the obligations of any agent are logically consistent, ie

- for any $\mathtt{t}, \mathtt{a}$, $\{\ \mathtt{F} : \mathtt{B}_{\mathsf{a}}^{\mathtt{t}}\mathtt{F}\ \}$ is consistent;
- for any $\mathtt{t}, \mathtt{a}$, $\{\ \mathtt{F} : \mathtt{OBL}_{\mathsf{a,b}}^{\mathtt{t}}\mathtt{F}$ for some $\mathtt{b}\ \}$ is consistent.

Good faith: is the assumption that agents only commit themselves to things they believe they are capable of doing, and also they believe they will honour the commitment:

- for any $\mathtt{t}, \mathtt{a}, \mathtt{b}, \mathtt{F}$, $\quad \mathtt{OBL}_{\mathsf{a,b}}^{\mathtt{t}}\mathtt{F} \to \mathtt{B}_{\mathsf{a}}^{\mathtt{t}}((\mathtt{ABLE}_{\mathsf{a}}\mathtt{F}) \wedge \mathtt{F})$.

Introspection The assumption is that agents are aware of their own obligations (and lack of obligations), though agents may be unaware of obligations made towards them.

- for all $\mathtt{t}, \mathtt{a}, \mathtt{b}, \mathtt{F}$, $\quad \mathtt{OBL}_{\mathsf{a,b}}^{\mathtt{t}}\mathtt{F} \to \mathtt{B}_{\mathsf{a}}^{\mathtt{t}}\mathtt{OBL}_{\mathsf{a,b}}^{\mathtt{t}}\mathtt{F}$;
- for all $\mathtt{t}, \mathtt{a}, \mathtt{b}, \mathtt{F}$, $\quad \neg\mathtt{OBL}_{\mathsf{a,b}}^{\mathtt{t}}\mathtt{F} \to \mathtt{B}_{\mathsf{a}}^{\mathtt{t}}\neg\mathtt{OBL}_{\mathsf{a,b}}^{\mathtt{t}}\mathtt{F}$.

Persistence of mental state The above constraints only rule out certain mental states at one point in time. There are also constraints on how beliefs etc change through time. We want to rule out belief sets which fluctuate widely from moment to moment. The assumption is that agents hang on to their old beliefs, and only abandon them if they learn a contradictory fact. Beliefs will thus persist by default. Also, new beliefs will not be taken up at random – only if an agent learns a new beliefs explicitly will it get added to the belief set.

Obligations are likewise taken to persist by default. However, the agent should be able to get out of an obligation, either by being released from the obligation by the other agent in question, or from realising that it is in fact not capable of honouring the obligation. (This topic is covered in [Cohen and Levesque, 1990].) This means that decisions of an agent can be overturned, as the agent can release itself from the obligation.

Capabilities are assumed to be fixed – what the agent can do at one time, it can do at another, possibly subject to some conditions holding at the time.

8.2 A generic agent interpreter

We assume that each agent has an *agent program* controlling the agent's behaviour. Actions occur as a side effect of an agent's commitment to an action at a given time; the actions include sending of messages. We'll look later at the agent programming language AGENT0.

Computation consists of each agent iterating the following loop at regular intervals:

1. read current messages, update mental state, including beliefs and commitments, according to the agent program.

2. execute commitments for the current time, possibly resulting in further change of beliefs (this is independent of the agent's program).

It's assumed that messages can be passed to other agents, addressed by name. The temporal aspects need to be coordinated, and a background clock is assumed to control the iteration of the interpretation loop, ensuring that the computation is regular. We need to be able to relate the clock scale to time as represented by the agents, and in particular, we need a notion of *current time* to say when commitments have matured.

A simplifying assumption is that any single iteration through the interpretation loop takes less time than the clock step – otherwise synchronisation problems become more serious. In a real distributed implementation, this assumption needs to be relaxed, and the loop will have to take account of other possibilities, for example, that messages may be received in a different order from that they were sent in.

8.3 The AGENT0 language

Shoham proposes this language for programming agents, together with an interpreter that is a simplified version of the generic agent interpreter. (I follow Shoham's original Lisp-like syntax here, but clearly other concrete syntaxes are possible.)

8.3.1 Syntax of AGENT0

The language is designed to specify how agents come to adopt commitments. Commitments here are simply primitive actions – the agent is not expected to commit to complex tasks that might need some

planning, for example.

Facts have two roles — the content of actions, as well as the conditions for execution. These are atomic basic statements (no logical or modal connectives will be used. The time argument appears as `t` in the following example:

```
(t (boiling kettle))
```

Actions are classified in two ways: they may be *private* or *communicative*, and they may be *conditional* or *unconditional*. The idea is that the private actions correspond to statements that only make sense to the agent concerned. *Communicative* actions involve IO, and are in a uniform language available to all agents.

The syntax for private actions is `(DO t p-action)` for some private action `p-action`. For communicative actions, there are three types of communication:

1. *Informing:* this is straightforward passing of information, indexed by some time `t` at which the informing happens:

   ```
   (INFORM t a fact)
   ```

   where the informing of fact `fact` goes to agent `a`.

2. *Requesting* at some time to some agent of some action is written

   ```
   (REQUEST t a action)
   ```

   Note that the time here is the time the request is made — the action may contain time information about the time the action is requested.

3. *Cancelling* a request is done similarly:

   ```
   (UNREQUEST t a action)
   ```

4. There is also a non-action notation, to indicate where commitments are to be avoided; this is written

   ```
   (REFRAIN action)
   ```

   and is local to the agent.

Conditional action statements Here actions are made dependent on some property of the mental state of the agent. The idea is that the agent concerned at the time relevant for the action will check the condition to see if the action is to be performed or not.

Mental conditions have two forms, `(B fact)` or `((OBL a) action)`, corresponding to a fact being believed, and to there being a commitment towards an agent of some action. A conditional action can be made by attaching a condition to an action, eg:

```
(IF (B (t' (boiling kettle)))
    (INFORM t a (t' (boiling kettle))))
```

saying that if, at time `t` the agent believes that the kettle boiled at time `t'`, then inform agent `a` of this.

Mental conditions can be combined with boolean connectives (`AND,OR,NOT`).

Pattern matching à la Prolog is used to indicate any possible match. Given that negation is present, we need to say how this works: a variable (written eg `?x` here) is existentially quantified, with a scope upwards to the first `NOT`, or the whole formula if it is not inside a negation. Thus

```
(IF (NOT ((OBL ?x) (REFRAIN sing))) sing)
```

is an action to sing, provided the agent is not committed to any other agent to refrain from singing.

Universal quantification is also allowed. This goes beyond what appears in traditional Prolog, and makes sense here because of the restricted syntax in other respects (there are no function symbols). Universally quantified variables (written `?!x`) have scope the whole formula. So

```
(IF (B (t (employed ?!x acme)))
    (INFORM a (t (employed ?!x acme))))
```

asks for information about *all* those employed by `acme`.

Commitment rules

Individual agents make use of an appropriate set of action statements. Most of these actions are unknown where the agent is programmed, however: the program specifies under what conditions the agent will enter into new commitments.

Commitments may be conditional, not only on mental conditions, but also on message conditions, referring to incoming messages. A *message pattern* has the shape

```
(From Type Content)
```

matching on the sender's name, the type of message (inform, request, ...) and `Content` is a statement. Conditions are then made up of boolean combinations of message patterns.

A commitment rule thus has the shape:

```
(COMMIT msgcnd mntlcnd (agent action))
```

where the `(agent action)` may be repeated (or absent). The rule says that, when the message and mental conditions are satisfied, there should be a commitment towards an agent for some action. For example,

```
(COMMIT (?a REQUEST ?action)
        (B (now (myfriend ?a)))
        (?a ?action))
```

is a commitment rule which commits to all requests from agents currently believed to be friends.

An agent is programmed, then, by assigning a sequence of commitment rules to the agent. An agent will also have some initial beliefs and capabilities.

8.3.2 The AGENT0 interpreter

This is a specialisation of the generic interpreter. Since the capabilities are fixed, the first step in the top-level loop consists of updating the beliefs and commitments of the agents.

Each agent has its own belief and commitment database. Beliefs are updated as a result of incoming information, or as a result of running a private action.

For the inclusion of new information, in general the problem of how to integrate new information, while avoiding inconsistency, is known as *belief revision* (see [Gärdenfors, 1992]). In AGENT0, as the syntax is kept simple here (only allowing negation as a logical connective), checking for consistency is feasible. In addition, the AGENT0 agents are *gullible* — they will simply believe any new information, and throw out older information which is contradictory.

The commitment database holds a set of pairs of the form `(agent action)`. The capability database contains pairs `(privateaction mntlcnd)` of *private* actions and mental conditions. (The condition is here to stop commitment to incompatible actions.)

Commitments are removed in two ways:

- as a result of belief change: an agent is required to believe it can honour any commitment; if beliefs change, so that the mental condition for an action is no longer met, then a commitment will be removed.
- as a result of an `UNREQUEST` message.

Commitments are added according to the following algorithm:
for each program commitment statement[1]

```
(COMMIT msgcnd mntlcnd (a action)*)
```

if:

- the message condition holds of incoming messages, and
- the mental condition holds of the current mental state, and
- the agent is capable of all the named actions
- the agent is not committed to refrain from any named action, and
- for any named refrain actions, the agent is not committed to the action,

then commit to all the named agent/action pairs.

Carrying out commitments The second step in the loop is to honour those commitments that have matured. In this step, the interpreter executes all actions whose time has come; since the time grain for executing the loop may not coincide with the single step in the time scale, this means commitments associated with a time between the current time, and one time-grain interval previously.

Different actions are executed as follows:

- `INFORM, REQUEST, UNREQUEST`: send the appropriate message.
- `REFRAIN`: No effect here.

[1] * here has the usual regular expression sense.

- `DO`: check belief and commitment databases, check mental condition associated with the action, and all being well perform the action.
- `IF`: consult the belief and commitment databases, and test the mental condition, and if all well, execute the action (recursively).

8.3.3 Examples

Consider the following natural language dialogue in the context of booking an airline flight.

```
P to C: Please tell me what flights you have from SF
        to NY on April 18th
C to P: Flight 354 leaves at 0830, flight 293 at 1000,
        flight 441 at noon, ...
P to C: Please book me on 354
C to P: That is sold out
P to C: Please book me on 293
C to P: That is confirmed with reservation number
        112358
P to C: Please book me also on 441
C to P: That conflicts with 293; I am not allowed
        to double book a passenger
P to C: Please get permission to do so
C to S: I request permission for the following ...
S to C: Permission denied
C to P: Sorry, I cannot get approval
```

For this example earlier, in an agent implementation, the airline booking agent could have a capability database with two items as below.

```
((issue_bp ?pass ?flight ?time) true)

((DO ?time (update_seats ?t1 ?flight_no ?extra_seats))
 (B (?time (remain_seats ?t1 ?flight_no ?current_seats))))
```

– here `issue_bp` describes the action of issuing the pass at the appropriate time; the second capability is to run an update on the number of available seats, provided the agent currently believes there are seats remaining on the flight.

The agent has a couple of commitment rules, one to inform the passenger of any information requested that is available:

```
(COMMIT (?pass REQUEST (IF (B ?p) (INFORM ?t ?pass ?p)))
          true
          ?pass
        (IF (B ?p) (INFORM ?t ?pass ?p)))
```

and another to reserve a seat on request, provided there is a seat available, and the customer is not already booked on another flight:

```
(COMMIT (?cust REQUEST (issue_bp ?pass ?flight ?time)) % msgcnd
        (AND (B (?time (remain_seats ?flight ?n)))   % mntlcnd
             (?n > 0)
```

```
          (NOT ((OBL ?anyone) (issue_bp ?pass ?fl ?time))))
     (myself (DO (now + 1)                     % fst action
             (update_seats ?time ?flight -1)))
     (?cust (issue_bp ?pass ?flight ?time)))   % snd action
```

Let's introduce a macro for convenience:

```
(query_which t asker askee q) =
   (REQUEST t askee (IF (B q) (INFORM (t + 1) asker q)))
```

Notice this will not return negative information, even if that is available. We need

```
(query_whether t asker askee q) =
   (REQUEST t askee (IF (B q) (INFORM (t + 1) asker q)))
   (REQUEST t askee (IF (B (NOT q))
                    (INFORM (t + 1) asker (NOT q))))
```

to be sure to get that also.

Now the interaction can be modelled by successive messages:

```
agent       action
-----------------------------------------------------------------
smith     (query_which 1march/1:00 smith airline
                (18april/?!time (flight sf ny ?!num)))
airline   (INFORM 1march/2:00 smith
                (18april/8:30 (flight sf ny #354)))
airline   (INFORM 1march/2:00 smith
                (18april/10:0 (flight sf ny #293)))
smith     (REQUEST 1march/3:00 airline
                (issue_bp smith #354 18april/8:30))
smith     (query_whether 1march/4:00 smith airline
                (OBL smith) (issue_bp smith #354 18april/8:30))
airline   (INFORM 1march/5:00 smith
            (NOT ((OBL smith) (issue_bp smith #354 18april/8:30))))
smith     (REQUEST 1march/6:00 airline
                (issue_bp smith #354 18april/10:00))
....

-------------------------------------------------------------------
```

Thus the passenger asks to book on the second flight, given that the request to book on the first has not been successful.

8.4 Agentification

The third component of Shoham's proposal is for a process of *agentification*. One way of building an agent system is to define the agents as above, and then to use a generic interpreter. This means that the intentional features of the agents (beliefs, obligations etc) are explicitly represented. However, we would like also to be able to pull lots of disparate devices into an agent framework, and these other devices are usually already implemented in some way. Can we incorporate other devices into an agent framework?

The role of the agentifier is to allow this incorporation. The key is that intentional notations can be viewed as a way of conceptualising a device, for the designer. Thus, for a design in AGENT0, there

can be an implementation in some other form. A lower-level description of the implementation can conversely be related up to a higher level intentional specification. Thus Shoham proposes that we should have a device to compute, from a description of a device in a low level process language, a description in AGENT0.

Shoham does not give details on this, simply saying that "I expect the unconstrained problem will be quite difficult". An interactive approach might be productive here.

8.5 Agentification

The third component of Shoham's proposal is for a process of *agentification*. One way of building an agent system is to define the agents as above, and then to use a generic interpreter. This means that the intentional features of the agents (beliefs, obligations etc) are explicitly represented. However, we would like also to be able to pull lots of disparate devices into an agent framework, and these other devices are usually already implemented in some way. Can we incorporate other devices into an agent framework?

The role of the agentifier is to allow this incorporation. The key is that intentional notations can be viewed as a way of conceptualising a device, for the designer. Thus, for a design in AGENT0, there can be an implementation in some other form. A lower-level description of the implementation can conversely be related up to a higher level intentional specification. Thus Shoham proposes that we should have a device to compute, from a description of a device in a low level process language, a description in AGENT0.

Shoham does not give details on this, simply saying that "I expect the unconstrained problem will be quite difficult". An interactive approach might be productive here.

8.6 Agent Oriented Programming: extensions

Shoham's proposal has had a couple of implementations, and some extensions to AGENT0 now exist. As it stands, it gives an idea of how the framework is plausible for fairly simple systems. However, we can see several ways in which the ideas can be explored further.

- *A richer mental life*: the agents could be allowed a richer set of mental attitudes. In particular, a more complex structure of motivations would give more interesting behaviours, and the addition of a planning phase, in which the agents can plan in order to find the best way to proceed, would be closer to the standard view of human agents.
- *Classes of agents*: Another feature often found in proposals for agent systems is that a group of agents form a class, with many features in common. Some support for this is needed (along the lines of object-oriented programming).
- *Persistence of mental states*: it is a hard problem to pin down what exactly this principle of the interpreter should mean in practice.
- *Belief revision*: the AGENT0 belief revision strategy when new information is acquired is naive. There are many rival approaches to this problem.
- *Beliefs about beliefs*: the simplifications made in AGENT0 rule out beliefs about beliefs in the belief database. This simplifies the interpretation, but throws away one of the attractive features of the agent approach, where agents can try to "second guess" each other.

- *Societies of agents*: many proposals for agent-like systems suggest that some of the interesting phenomena will arise on the level of the collective of *society* of agents. There could be features associated with this level, such as social *rules* which allow the agents to get on together.

  For other proposals relating to societies of agents, see [Minsky, 1985, Winograd, 1987].

8.7 Minsky's agents

The notion of agent in [Minsky, 1985] is different from that in AGENT0. Minsky is less concerned to give a formal characterisation of the way agents are specified and interact, and more concerned to argue that this is the right way to approach an understanding of mind.

Minsky says ([Minsky, 1985, p 17]):

> I'll call "Society of Mind" this scheme in which each mind is made of many smaller processes. These we'll call *agents*. Each mental agent by itself can only do some simple thing that needs no mind or no thought at all. Yet when we join these agents into societies—in certain very special ways—this leads to true intelligence.

This already introduces the idea that the individual agents should be computationally simple; the interesting behaviour is the result of the interaction.

Minsky suggests that even simple tasks involve several agents ([Minsky, 1985, p 20]).

> What kinds of smaller entities cooperate inside your mind to do your work? To start to see how minds are like societies, try this: *pick up a cup of tea!*
>
> > *Your GRASPING agents want to keep hold of the cup.*
> > *Your BALANCING agents want to keep the teat from spilling out.*
> > *Your THIRST agents want you to drink the tea.*
> > *Your MOVING agents want to get the cup to your lips.*

These sorts of agents are not engaged in the sort of interaction corresponding to message passing on the conscious level, as happened in the airline example. Still, there is some sort of interaction going on, and Minsky uses an intentional vocabulary here (the agents *want* . . .).

Minsky want to understand intelligence this way, so he deliberately wants to be sure that none of the agents is intelligent (otherwise the explanation will be circular). He points out that the capabilities of an agent can be looked at in two ways: what the agent can do on its own, and what it can do as a member of a society. Talking of an agent-based program called Builder, he says ([Minsky, 1985, p 23]):

> Let's use two different words, *"agent"* and *"agency"*, to say why *Builder* seems to lead a double life. As agency, it seems to know its job. As agent, it cannot know anything at all.

The idea is that, on its own as an isolated agent, *builder* is a simple process that turns other agents on and off. But, seen from outside, *builder* does whatever all its sub-agents accomplish, using one another's help.

8.8 Sloman on human-like agents

In various papers, Aaron Sloman has been working out ideas on more complex agents, trying to get closer to human capabilities (see [Sloman, 1996][2]). There is a toolkit for building agents, and several projects have been carried out. Sloman is interested in incorporating different sorts of sensing into an agent architecture; the agent should have rules allowing it to process incoming sensory data, and this will be an important part of how an agent reacts (so more is involved than simply receiving messages from other agents).

Sloman distinguishes agents by the sorts of control regimes they employ.

Reactive Subsystems Here the information coming in is processed in a fixed way, resulting in some output and or action. There may be some fixed conflict resolution strategy, but the processing should be like a reflex action, and not a considered response to the information at hand.

Deliberative Subsystems A reactive system is relatively inflexible; it is liable to fail when confronted with unexpected situations, and it cannot plan ahead to try out "mentally" various alternative actions to see which is the best option. An agent with the ability to form new representations, and to use them to reason hypothetically, is called *deliberative*.

In this situation, resource allocation becomes more important; in reactive systems, the response can be expected to involve some fixed amount of computation, regardless of the situation. Reasoning about hypothetical situations is potentially time consuming, and likely to involve very different amounts of computation, depending on the input.

Some learning ability is important here; solutions which are found via deliberation will be useful later on, and they can be turned into reactive components (this is like the way skills are developed in humans, where conscious problem solving for a problem class turns in to access to a known set of solutions).

A meta-management subsystem Sloman envisages a third sort of control system. The deliberative system itself is likely to suffer from some inflexibility, and more flexibility can be achieved by a monitoring system that considers the behaviour of the deliberative system, its use of resources, whether or not there is a pattern to the problems tackled, and so on, in order to redirect the efforts of the deliberative system. The motivation here is like that for meta-interpreters of reasoning systems in general; it allows an extra expressiveness, and can let us side-step some of the computational drawbacks of working with a fixed inferential mechanism.

8.8.1 Conclusion

The notion of agent as used by Minsky, Shoham, and Sloman gives some idea of the range of uses of the term. Minsky's is an earlier use, from where later work has kept the broad idea, while filling in more detail, and being more specific. Shoham takes seriously the idea that we can have a programming language by taking (a part) of the ideas involved and turning them into primitive operations in the programming language. Finally, Sloman is concerned to keep closer to Minsky's claim that complex human competences can be understood in terms of interacting agents, and he has elaborated some of the options for control in such systems.

[2] http://www.cs.bham.ac.uk/~axs/misc/agent.architecture.html

8.9 AGENT0 Programming

We look now at some examples of programming in the style of AGENT0. We will compare some pseudo-code with a Prolog implementation that we can run.

Example

Problem: Design an agent that will agree to any request, if the requesting agent is believed to be a friend, and the agent is capable of the action; and will also announce "hi there" when requested by any agent at all. Recall that agents are described by their capabilities, beliefs, and commitments; new commitments are generated by the agent's commitment rules.

Let's give the agent here a "private" action of saying "hi there".

- *Capabilities:* To say "hi there".
- *Initial beliefs:* That some agents are friends.
- *Initial commitments:* None.
- *Commitment rules:*
  If message requests some action
  and requesting agent is believed to be a friend,
  then commit to performing the action.
- *If* message requests to say hi,
  and true
  then commit to saying hi.

At the code level, this yields the following.

```
% agent name

who_am_i(greeter).

% initial beliefs
% need to "assert" for Prolog reasons
% beliefs tagged with list of
% time/truth value pairs

:- assert(b(greeter,myfriend(smith),
            [[[0,0,2002,0,0,0],t]])).

% capabilities -- need to
% repeat inform etc for each agent

can(greeter,do(Time,sayhi)).
can(greeter,do(Time,sayhowdy)).
can(greeter,inform(A,B,C)).
can(greeter,request(A,B,C)).
can(greeter,refrain(A,B,C)).

% commitment rules
% always sayhi if asked
```

```
commit(greeter,[A,request,do(Time,sayhi)],
               [],
               A,
               do(Time,sayhi)).

% always do request for requester,
% if requester currently friend

commit(greeter,[A,request,B],
               [b([Now,myfriend(A)])],
                A,
                B):-
        what_time_is_it(Now).

% private actions

sayhowdy :- write('Howdy!'),nl.

sayhi :- write('Hi there'),nl.
```

To get anything to happen with such an agent, we need another agent in the vicinity, such as the following minimal agent.

```
%  agent name

who_am_i(smith).

% no initial beliefs

% standard capabilities
  ...
% commitment rule:
% times of request & action unspecified,
% no message or mental conditions

commit(smith,[],
             [],
             smith,
             request(T,greeter,
                       do(Time,sayhi))).
```

An identical agent called `jones` is also around for testing.

Execution

This gives the following behaviour.

The interpreter can be run for successive "ticks" round the basic loop of processing messages and executing commitments.

At each tick, after the first:

- the greeter has a message from smith and from jones

- the commitments rules form 3 new commitments (2 for smith, 1 for jones)
- smith and jones form commitments to request a greeting
- the greeter performs the action 3 times
- smith and jones send new requests to the greeter.

8.9.1 Debugging

Some hints on debugging such systems, as this is a serious problem in distributed systems.

- use `clean_database/0` to tidy up beliefs, and messages in transit.
- browse the agents' beliefs with

  ```
  ?- b(agent, X, Y).
  ```

  Beliefs are tagged with a list of time and truth value pairs; agents have beliefs about the commitments they have formed, so these can be accessed in this way also.
- Instantiation of variables can happen mysteriously. It is more robust to use the current time via `what_time_is_it/1`.

8.9.2 A more complex example

Let's look at agents for the game paper, scissors, stone.

There are two agents playing the game. At each turn, the agents simultaneously play one of the three options (paper, scissors, stone).

If they make the same play, no-one wins; otherwise paper beats stone, stone beats scissors, and scissors beats paper.

We can't ensure simultaneity directly here; let's use a third referee agent to gather the play information and keep track of the scores.

The player agents: outline

- *Capability:* to pick a move – may be smart, constant, random ...
  This also forms a commitment to the umpire to inform what the move is.
- *Commitment rule:* if the umpire requests to pick a move, then do so.

Umpire agent: outline

- *Initial beliefs:* about the start state of the game.
- *Initial commitment:* to start the game.
- *Capabilities:* to score the game, by working out the winner and keeping track of the score.

- *Commitment rules:* if believe that both agents have played, then:
  – score the game
  – add beliefs that agents have *not* made move (for next round)
  – request agents to pick new moves

We can now proceed to the implementation level.

Code: player agents

```
% agent name

who_am_i(player1).

% no initial beliefs

% capabilities -- pick_move by player

can(player1,do(_Time,pick_move(player1))).

% commitment rules

commit(player1,[umpire,
                  request,
                  do(T,pick_move(player1))],
               [],
               umpire,
               do(T,pick_move(player1))).

% private actions %  -- really need a hiding mechanism

pick_move(Player) :- ....,
    add_commitment(Player,
                   umpire,
        inform(Now,umpire,
              [Now,my_move(Player,Move)])).
```

Code – umpire agent

```
% agent name

who_am_i(umpire).

% initial beliefs

:- assert(b(umpire,score(0,0,0),
             [[[1,1,2002,0,0,0],t]])).
:- assert(b(umpire,round(1),
             [[[1,1,2002,0,0,0],t]])).

% initial commitments -- % again, need to use assert
```

```
:- assert(cmt(umpire,umpire,
              request([1,1,2002,0,0,0],
                 player1,
                 do([1,1,2002,0,0,0],
                     pick_move(player1))))).

% and same for player 2
% capabilities

can(umpire,do(_Time,score_game(_,_))).

% commitment rules-- have to repeat
% conditions for each rule; the implementation
% should allow a list of actions.

commit(umpire,[],
              [b([Now, my_move(player1,M1)]),
               b([Now, my_move(player2,M2)])
              ],
              umpire,
              do(Now,score_game(M1,M2))) :-
                      what_time_is_it(Now).

% for same mental conditions

            ...
            umpire,
            inform(Now, umpire,
             [Now, not(my_move(player1,M1))]).

% Notice the umpire can inform itself --
% need a possible action here.
% initiate next round --
% again same mental conditions

commit(umpire,
          [],
          [b([Now, my_move(player1,_M1)]),
           b([Now, my_move(player2,_M2)])
          ],
         umpire,
         request(Now,
                 player1,
            do(Now,pick_move(player1)))) :-
                 what_time_is_it(Now).

% similarly for player2 % private actions
```

```
score_game(M1,M2) :-  ...
     who_wins(M1,M2,Xplus,Yplus),
     new_score(OldX,OldY,Xplus,
               Yplus,NewX,NewY),
     add_belief(umpire,
         [Now,score(N,NewX,NewY)]),
     ...
```

8.9.3 Execution

We can run this for some number of ticks (or for some time, e.g. 30 seconds, using

```
?- simulate(for,[0,0,0,0,0,30]).
```

All that is immediately visible is trace of messages being received. But *a lot* of beliefs about the state of the system are generated.

```
| ?- b(X,Y,Z).
X = player2,
Y = cmt(player2,umpire,do([1,1,2000,0,0,0],
                     pick_move(player2))),
Z = [[[2,26,2002,13,41,35],t]] ? ;

X = umpire,
Y = cmt(umpire,umpire,do([2,26,2002,13,41,35],
                   score_game(stone,paper))),
Z = [[[2,26,2002,13,41,35],t]] ? ;

X = umpire,
Y = cmt(umpire,umpire,
         inform([2,26,2002,13,41,35],umpire,
         [[2,26,2002,13,41,35],
         not(my_move(player1,stone))])),
Z = [[[2,26,2002,13,41,35],t]] ? ;
```

Getting back information

We want a judiciously chosen selection of information to get an idea of the behaviour of the system.
In this case, we can take successive scores of the game:

```
| ?- b(umpire,score(X,Y,Z),W).

W = [[[1,1,2000,0,0,0],t]],
X = 0,
Y = 0,
Z = 0 ? ;

W = [[[2,26,2002,13,41,35],t]],
X = 1,
```

```
Y = 0,
Z = 1 ? ;

W = [[[2,26,2002,13,41,36],t]],
X = 2,
Y = 1,
Z = 1 ? ;
```

8.9.4 Some comments

- **Not Forgetting:** we expect that this aspect will lead to space efficiency problems for larger systems. But *some* memorising of past beliefs can help to avoid repeating previous mistakes.
- This implementation is not distributed — but this can be achieved relatively easily; see e.g. Sicstus Prolog implementation of Prolog-Linda.
- It is very easy to “cheat” with this implementation, eg by adding beliefs directly to another agent! Of course, this would be an underhand move . . .

8.9.5 Other approaches

There is a lot of work going on in this area, with many active research groups. Some examples you may look at are:

- The Foundation for Intelligent Physical Agents – producing standards for the interoperation of heterogeneous software agents.

  `http://www.fipa.org/`

- AgentLink – European Research Consortium

  `http://www.agentlink.org/`

- Multi-Agent Systems, lots of agent resources:

  `http://www.multiagent.com/`

Chapter 9

Methodologies for Intelligent Systems Development

9.1 Introduction

Much of the development of knowledge-based systems has now moved out of the research stage. There exist many advanced techniques and tools available which are capable of tackling real-world problems. There is, therefore, a real need for a standard systems development methodology, in order to meet the increasing demand for delivering high performance systems in a cost-effective and timely manner. Ideally, such a methodology should explicitly map a prescribed application problem onto the most appropriate solution technique or a combination of suitable techniques, selected from a large and still expanding set of possibly applicable methods. A more realistic methodology, however, tends to provide guidelines about the procedures that are learned from developing past systems and hence, are known to be useful in building new systems potentially effective for a particular type of task. This helps alert the system developers to factors which they may not have considered, thereby preventing the exploration of blind alleys or, if nothing else, avoiding 're-inventing the wheel'.

There are now various working methodologies for developing knowledge-based systems. A common, and not surprising, feature of these methodologies is that the use of them is closely related to the characteristics of the knowledge available in the application domain, especially to the knowledge source and orientation. A well-known methodology is KADS (knowledge acquisition and domain structuring) [A. Schreiber and Breuker, 1993, Hickman et al., 1989, Schreiber et al., 1999], which is primarily applied to the early stages of domain problem analysis. This is briefly introduced next. Whilst the creation of a generic KBS development methodology remains as active research, a general consensus over the suitability of various problem-solving approaches has emerged with regard to what sort of knowledge is available. A brief discussion about this will be given in section 9.3.

9.2 The KADS Methodology

The KADS methodology is, perhaps, one of the most widely adopted methodological approaches to the development of conventional knowledge-based systems. It attempts to provide a guidance on obtaining (mainly empirical) knowledge, analysing it and transforming it into a detailed design for implementing a required system. The clear focus of the KADS methodology is, however, on the analysis phase. In line with this, the discussions below will concentrate on its approach to problem analysis unless otherwise stated.

9.2.1 Basic principles

KADS assumes that the following basic principles are to be followed when considering a KBS application [Breuker and Wielinga, 1987]:

- The knowledge and expertise should be analysed before the design and implementation starts.
- The analysis should be 'model-driven' as early as possible, where a model is, informally, a conceptual description of domain expertise; that is, a framework of how the knowledge is structured should be applied early on in the analysis process and used to interpret subsequent information data.
- The content of the model should be expressed at the epistemological level; i.e. an appropriate intermediate knowledge representation mechanism should be used, rather than going straight to encode the knowledge with a particular formalism before knowing how it may be best implemented.
- The analysis should cover the functionality of the prospective system, including an understanding of who will use the system and under what conditions.
- The analysis should proceed in an incremental way; new data should be elicited only when previously collected data have been analysed and their interpretations documented.

Whilst many of these principles reflect common-sense intuitions, KADS clearly recognises the significance of the use of abstract or idealised knowledge models to inform and direct the construction of a system for the problem at hand. It works by identifying the following four layers of knowledge abstraction (which reflect different knowledge orientations):

- the *domain layer* that contains the object-level knowledge, which describes the concepts, elements and their relations to each other that are specific to the application domain
- the *inference layer* that gives a kind of meta-level knowledge, which indicates how the object-level knowledge is to be organised and used in the primitive reasoning processes and hence, is declarative in nature
- the *task layer* that presents another kind of (higher) meta-level knowledge, which shows how goals and sub-goals may be reached, i.e. how tasks and sub-tasks should be performed in solving given problems and hence, is procedural in nature
- the *strategy layer* that involves a further kind of (even higher) meta-level knowledge, which provides the way of monitoring and controlling the overall execution of the problem-solving tasks

Central to this arrangement is a library of so-called *interpretation models* (or task templates [Schreiber et al., 1999]), which are generic and reusable structural descriptions of the types of task that prospective application systems may perform.

9.2.2 Interpretation models

Within the interpretation models, primitive inference functions and the elements of knowledge that the reasoning functions will operate on are explicitly described. Given some (domain) knowledge,

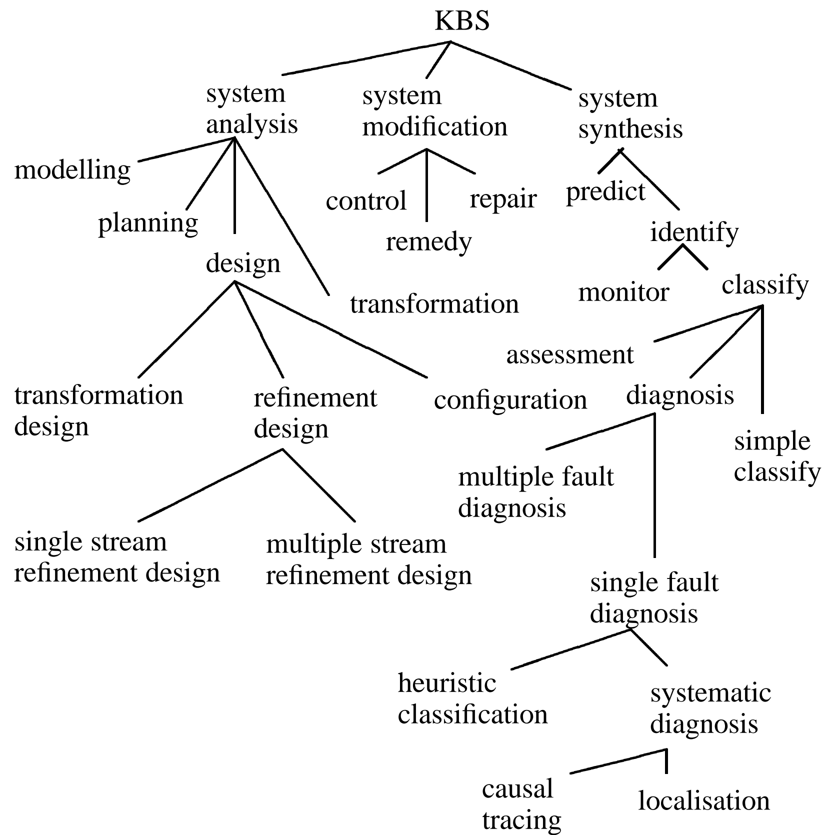

Figure 9.1: The KADS taxonomy of tasks

KADS recommends selection of an interpretation model. These interpretation models are intended to support the creation of the inference layer by providing guidance about the possible inferences, and the ordering of these inferences, that are to be involved in the application system. In organising the library of interpretation models, KADS employs a taxonomy of tasks as shown in figure 9.1.

A typical example of the interpretation models is presented in figure 9.2, which describes the idealisation of assessment tasks. Within this diagram, the unboxed words indicate the inference functions required to perform sub-tasks and the boxes represent the knowledge elements used as inputs and/or outputs to these functions. With the incremental acquisition of knowledge, interpretation models can, of course, be refined and, if necessary, combined with others to yield new ones in order to suit the given problem.

Consider a specific application example. Suppose that the task is to develop a system for assessing the suitability of banks, or building societies, for a potential investment. This task can then be matched with the interpretation model for assessment tasks given above, as follows:

- the case description matches the bank's policy statements;
- the abstract case description reflects the main interests and penalties that the bank imposes, summarised from the policy statements;
- the ideal case is a stereotypical description of "ideal" bank policies, which could be real or imagined;

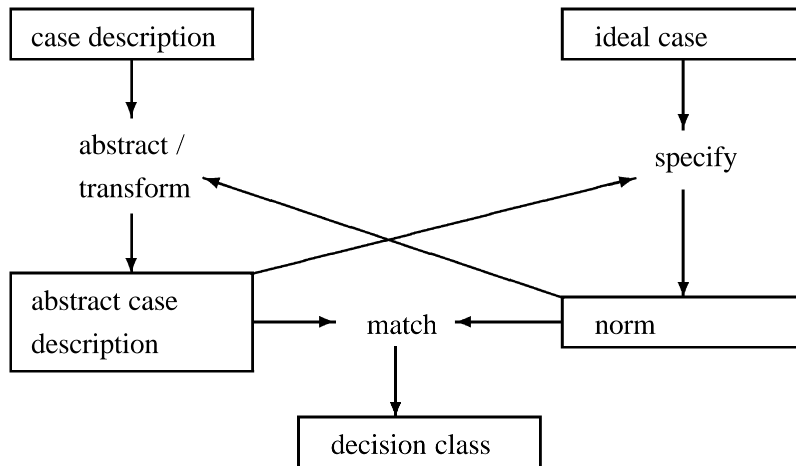

Figure 9.2: Interpretation model for assessment tasks

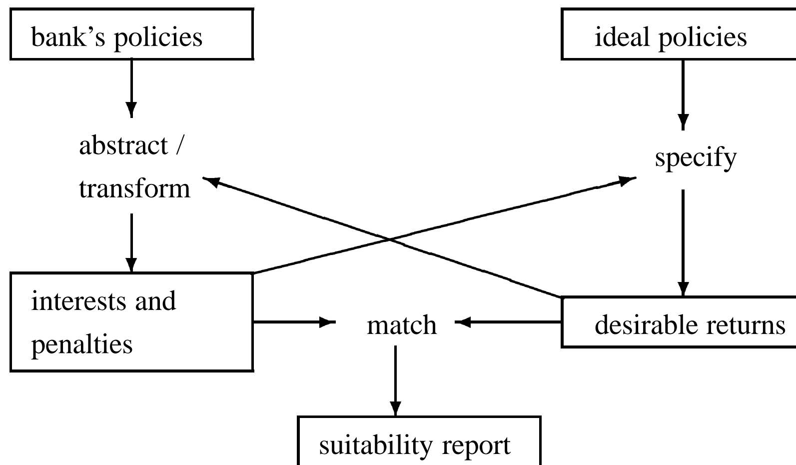

Figure 9.3: Interpretation model for investment assessment

- the norm is specified to be the desirable returns, with a minimised possibility of getting penalties; and
- the main interests and penalties are then compared against the desirable returns, resulting in a suitability report on the bank

The resulting instantiated interpretation model for assessment is shown in figure 9.3.

9.2.3 Application of pragmatic KADS for problem analysis

Before proceeding to further discussions, it is worth noting that applying the full version of the KADS methodology to problem analysis usually consumes a significant amount of time and effort, due to its inherent generality. Indeed, this may make the methodological approach too expensive. In particular, requirements like documenting progress made and decisions taken may not be immediately relevant to the needs of developing small or medium-sized applications, which are the most demanded in today's

market (although these requirements may well be necessary for a large-scale application). For this reason, the rest of this section only addresses a simplified version of KADS, the “pragmatic KADS”. For simplicity, this simplified version is named the same hereafter as the full KADS methodology. (Note that an updated, complete treatment of KADS can be found in [Schreiber et al., 1999] or at: http://www.commonkads.uva.nl/)

To illustrate how KADS offers guidance in practice, consider the development of a system for selecting undergraduate degree courses (which has been implemented for the Department of Business Studies at the University of Edinburgh). This example was first reported in [Kingston, 1992], which did not involve the strategic level analysis due to the nature of the relatively simple application task. Since the object-level knowledge was readily available in the domain, the analysis process proceeded in a bottom-up manner, i.e. starting from the domain layer, via the inference layer, and up to the task layer. Note that it could be equally plausible to follow a top-down analysis sequence. Indeed, for many other applications of the KADS methodology, the top-down approach is often preferred.

Most of the object-level domain knowledge was found in the University Calendar, which laid out the broad syllabus for each course and the other courses that were prerequisites for taking it. Other department-specific information was elicited through a few telephone interviews, uncovering additional knowledge about optional courses, which fitted well with certain degrees, and extra information about typical course combinations. From this, the creation of the domain layer was then fairly straightforward.

The construction of the inference layer involved three stages:

- Determining the type of the application task
- Identifying the generic interpretation model for that task type
- Adapting the generic interpretation model to the application domain

Since the essential task of the application system was under some constraints to construct or configure an acceptable course schedule from the collection of available courses, the task type was determined to be that of configuration. Based on this, the configuration model as shown in figure 9.4 was then selected from the library of interpretation models. This model was used as a guiding template from which to generate a suitable interpretation model for course selection.

It is clear from figure 9.4 that the model generation process includes: a) producing a list of components to be added to the configuration model selected; b) adding components one by one to the partially generated configuration model; c) verifying whether the new component fits into the configuration and, if so, whether the configuration is complete; and d) reviewing the resulting configuration and producing the final outcome. Through this generation process, the adapted application interpretation model, or the inference structure for course selection was obtained as shown in figure 9.5.

This adapted structure appears to be rather different from the original generic interpretation model. Nevertheless, it possesses the underlying principles embedded in the original. This can be observed from a number of viewpoints:

1. The course schedule box in figure 9.5 corresponds to the partial configuration box in figure 9.4.
2. Certain courses are required of all students and, hence, should be added to the course schedule first, one at a time. This is reflected by the update-1 inference function in figure 9.5, which corresponds to the instantiate-1 function in figure 9.4.
3. Students are allowed to choose further courses to add to their schedule. This is represented by the select-1 function in figure 9.5, which corresponds to the select-1 function in figure 9.4.

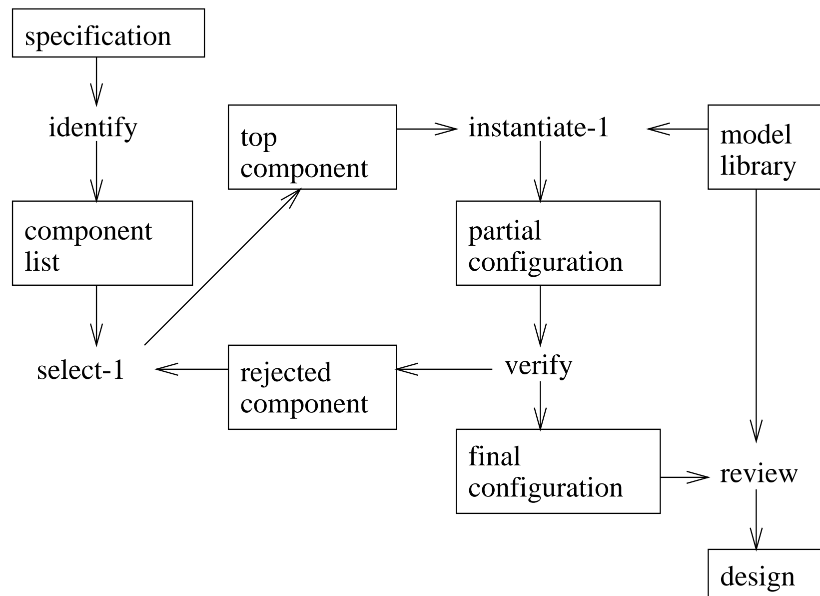

Figure 9.4: Interpretation model for configuration tasks

4. Students are allowed to continue selecting courses until their schedule is full. The check for a complete schedule is performed by the compute-1 and compare-1 functions in figure 9.5, which jointly correspond to the verify function in figure 9.4.

Thus, the original generic configuration template was adapted to a detailed inference structure suitable for the given application task. If necessary, this generation process can be performed further on. For instance, the update-2 inference function in figure 9.5 involves itself complex inferences and, therefore, could be further decomposed into that as given in figure 9.6.

Having obtained the declarative interpretation model at the inference layer, the next step is to identify the order in which the inferences required by this model should be executed. For the course selection task, this was done by devising an algorithm with procedures covering all the components of the inference structure. The resulting task layer algorithm is given in pseudocode as shown in figure 9.7. This concludes the KADS analysis phase for the present application. The next phase is to design the application system.

9.2.4 The Design Phase

In the full version of KADS this phase can be thought of as a three stage process, comprising functional, behavioural and physical designs, as briefly summarised below:

- **Functional design** is to decompose the inference structure, which was produced in the problem analysis phase, into groups of problem solving functions, storage functions and cooperation functions (in software terms). Additionally, it provides an indication of the data flow and control hierarchy in the application system. To preserve the structure of the inference layer KADS offers guidance about the decomposition, such that inference functions are normally mapped onto problem solving functions and knowledge elements onto storage functions, whilst cooperation functions permit communications between these two major types of function.

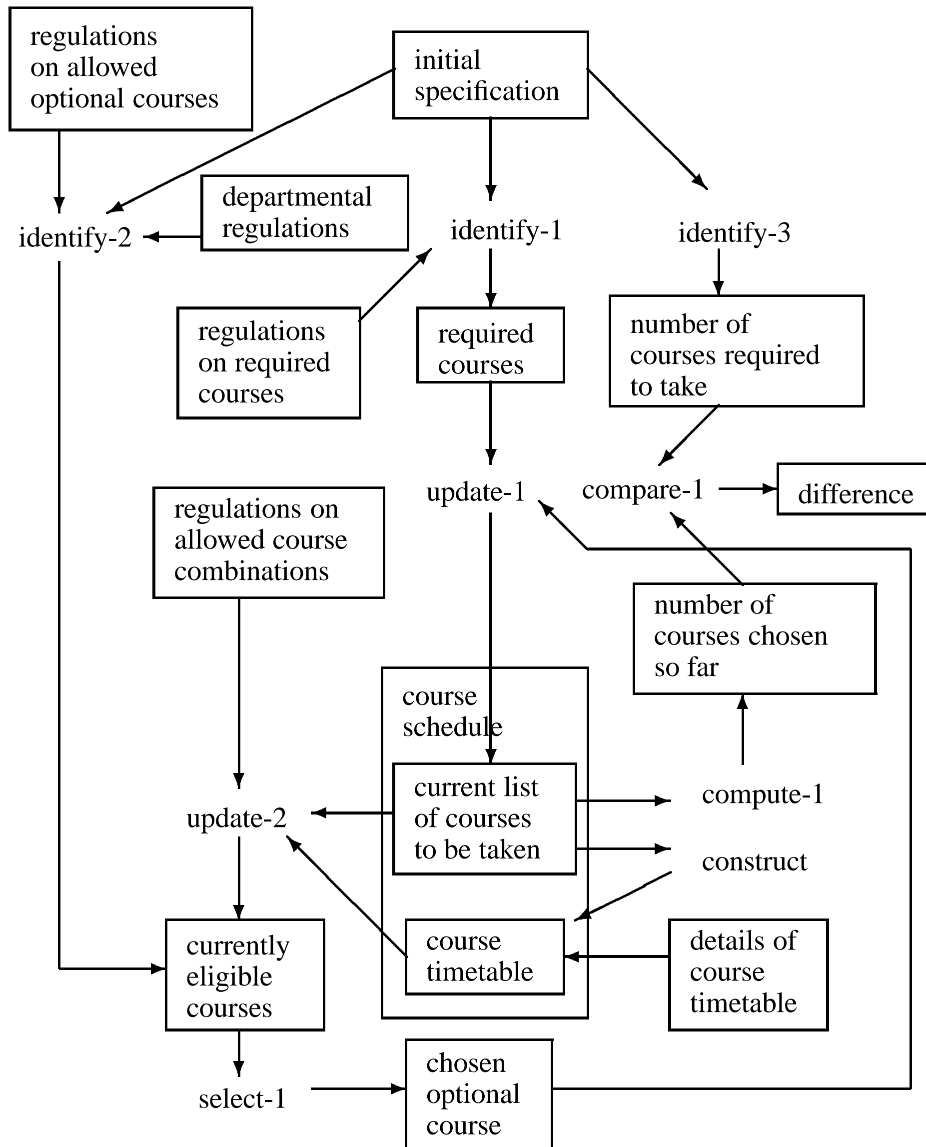

Figure 9.5: Inference structure for the course selection task

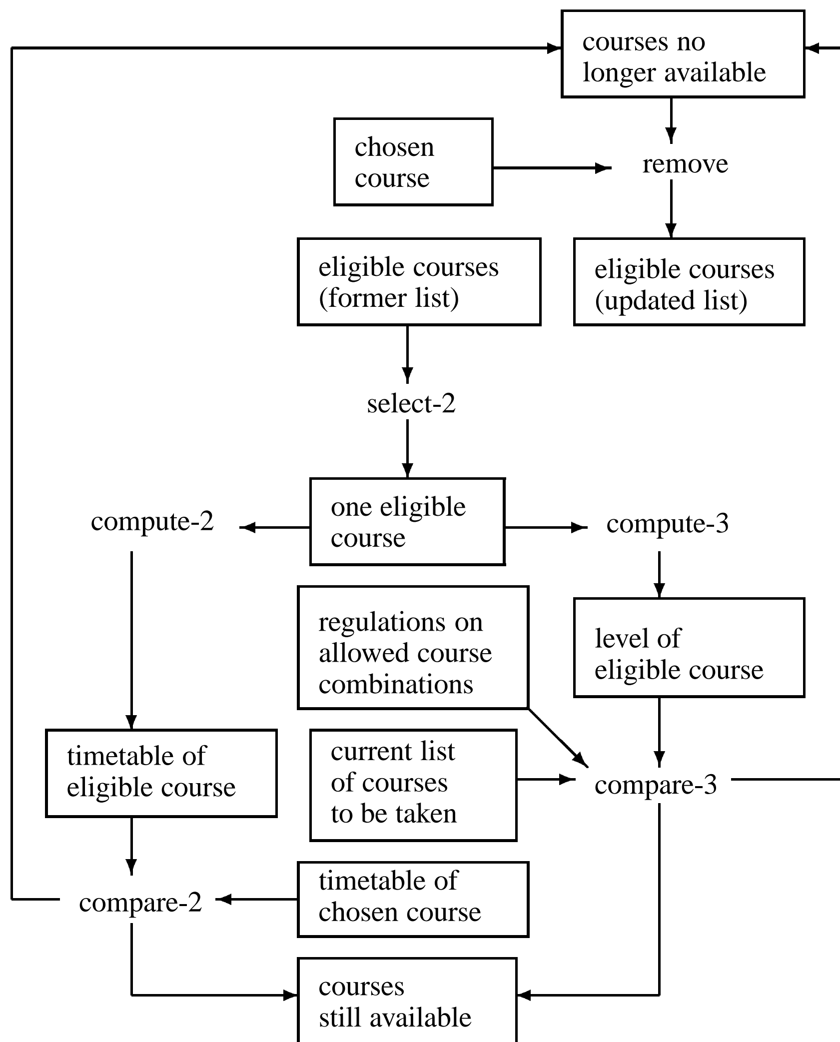

Figure 9.6: Decomposition of the update-2 inference structure

- **task**: $configure_course_schedule$
- **goal**: Assist a student to choose a set of courses complying with University regulations.
- **control terms**:
  - $required_courses$: Courses that student must take.
  - $eligible_courses$: Courses that student is permitted to take.
- **task structure**:

$configure_course_schedule : required_courses \times eligible_courses \mapsto course_schedule$
$obtain(initial_information)$
$identify(required_courses)$
$identify(eligible_courses)$
$identify(number_courses_to_be_taken)$

For each $required_course$ in $required_courses$:
  $update_1(current_course_schedule) = course_schedule_with_required_course$
end

While $number_courses_in_current_course_schedule < number_courses_to_be_taken$ do:
  $compare(number_courses_in_current_course_schedule, number_courses_to_be_taken)$
  $select_1(eligible_courses) = optional_course$
  $update_1(current_course_schedule) = course_schedule_with_optional_course$
  $compute(number_courses_chosen)$
  $update_2(eligible_courses)$
end

$assign(current_course_schedule) = course_schedule$

Figure 9.7: Task layer algorithm for course selection

- **Behavioural design** is to choose a particular kind of algorithm to implement a given functional component. KADS gives hints to focus the attention onto certain (and sometimes rather general) AI techniques, such as data-driven reasoning or best-first search.
- **Physical design** is to identify the data structures appropriate for implementing the algorithms chosen in the behavioural design stage.

9.3 More on the Methodologies

The eventual selection of which techniques/tools to implement an application system with largely depends upon what knowledge is available about the problem at hand. Although it can be difficult to devise a generic methodology for making such a selection, lessons may be learned from the existing widespread applications of AI reasoning paradigms (which include rule-based reasoning, case-based reasoning, constraint-based reasoning, and model-based reasoning). The last decade has seen considerable work on comparing and evaluating reasoning alternatives for different problem domains, and on identifying and exploiting their commonalities.

As an example, it is generally established that case-based reasoning (CBR) systems provide a number of important advantages over conventional rule-based systems, even though both approaches use empirical knowledge:

- CBR reduces the difficulty in the process of knowledge acquisition, the bottleneck in any application of knowledge-based systems. This is because the empirical knowledge used in a rule-based system is represented as a collection of association rules derived from past experience, whilst the experience is gained directly from solving problems presented in past cases. Knowledge required to construct a CBR system is, therefore, usually less difficult to acquire than that used in a rule-based system. It is easier to articulate, examine and evaluate case histories than rules. Most often, rules are formalised expertise provided by experts who are, in the first place, those people who have vast specialised experience, learned from witnessing a large variety of cases in the problem domain.
- A CBR system can remember its own past performance. If it has been successfully used to, say, diagnose a patient with certain symptoms (that is, such a case has been stored in its case memory), when facing another patient with similar symptoms the system would efficiently devise a similar treatment by retrieving, and perhaps modifying, the treatment given to the previous patient. In contrast, rule-based systems do not generally employ a memory. For the first patient it might have to fire many rules to come up with a diagnosis and treatment. When presented to another patient having the same symptoms, it has to redo the same rule-firings from scratch.
- CBR is more robust than rule-based reasoning in general. CBR systems are designed to operate in conditions where the current problem only partially matches those recorded previously. Such a system can adapt its solutions to novel problems by reasoning through analogy. Using the medical diagnosis example again, if a CBR system is presented with a patient whose symptoms appear to be different from any cases that it has successfully diagnosed, the system does not need to give up. Rather, it might try to retrieve various previous cases that were similar in one way or another and devise a treatment accordingly. Rule-based systems could not do this, however. Presenting a problem to a rule-based diagnostic system which does not match (or partially match) all the conditions of any one rule in the rule base, no solutions would be generated.

No reasoning paradigm is universally acceptable, of course. A CBR system works by relying upon the availability of an initial set of cases reflecting the typical relationships between problems and solutions. As with any knowledge-based systems, case-based systems have to encounter situations where only uncertain knowledge is present. As a result, solutions generated by a case-based system would not always be optimal or even correct. Also, questions such as how to best index cases and how to effectively and efficiently adapt cases are unresolved (see the Knowledge Representation notes for this).

Instead of making an exclusive choice of one reasoning approach, current research encourages integration of different solution techniques to maximise the use of various sorts of knowledge available. In addition to the core methodological issue of identifying categories, structures, or properties of knowledge or tasks for which different reasoning techniques are appropriate or advantageous, typical topics involved in such integration tasks include:

- Using one form of reasoning to support or guide another
- Compiling one form of reasoning experience into another form of reasoning knowledge
- Transferring successful reasoning methods from one form to another
- Switching among alternative forms of reasoning
- Investigating the interoperability of applications based on different reasoning technology

A significant amount of work exists that demonstrates practical advantages of an integrated approach for coping with real problems. For example, CBR has been combined with model-based reasoning (MBR) in [Feret and Glasgow, 1997]. This work uses the explicit structural model of the physical system under diagnosis and the fault candidates postulated by the component MBR process to index and match cases. Retrieved cases are then used to overcome errors introduced by the application of incomplete and/or incorrect system models. The integrated program allows more accurate fault isolation, without overwhelming the user with too many fault candidates as compared to using MBR alone. Incidentally, in MBR systems, including the combined ones like this, constraint-based reasoning (CSR) is often utilised to increase the effectiveness and efficiency of inference, also. This combination of MBR and CSR is natural because they both tend to exploit theoretical knowledge more than empirical knowledge.

Finally, it is worth indicating that apart from the choice of solution techniques, there are other important design decisions to make. These include selecting a language for knowledge representation, selecting software for system implementation, and refining the resulting system with respect to given objectives. Clearly, the outcomes of the latter two design stages have a lot to do with the outcome of the former, as the success of identifying a right knowledge representation language allows the most appropriate reflection of the knowledge at hand.

In practice, however, it is difficult to escape pragmatic considerations when choosing such languages. In particular, the system developers typically have to consider issues like the organisation's resources and capabilities. This is because the organisation may already have a software tool and perhaps, more importantly, knowledge of using the tool. Available resources, both technically (e.g. the computer network in which the knowledge-based systems will operate) and financially, will play a significant role in deciding what language to employ. Nevertheless, choosing a language solely based on satisfying the resource constraints is not advisable. It can be more wasteful if such a chosen language is established to be wrong after system deployment [Durkin, 1994]. It is these kinds of potentially conflicting requirements that make it necessary to follow carefully a system development methodology whenever possible.

Bibliography

[A. Schreiber and Breuker, 1993] A. Schreiber, B. W. and Breuker, J. (1993). *KADS: a Principled Approach to Knowledge-Based System Development*. Academic Press.

[Borst et al., 1997] Borst, P., Akkermans, H., and Top, J. (1997). Engineering ontologies. *International Journal of Human-Computer Studies*, 46:365–406.

[Breuker and Wielinga, 1987] Breuker, J. and Wielinga, B. (1987). Use of models in the interpretation of verbal data. In Kidd, A., editor, *Knowledge Acquisition for Expert Systems - A Practical Handbook*. Plenum Press.

[Bylander and Chandrasekaran, 1988] Bylander, T. and Chandrasekaran, B. (1988). Generic tasks in knowledge-based reasoning: The right level of abstraction for knowledge. In Gaines, B. and Boose, J., editors, *Knowledge Acquisition for Knowledge Based Systems*, volume 1, pages 65–77. Academic Press.

[Cohen and Levesque, 1990] Cohen, P. and Levesque, H. (1990). Intention is choice with commitment. *Artifical Intelligence*, 42(3):213–261.

[de Kleer, 1986] de Kleer, J. (1986). An assumption-based tms. *Arificial Intelligence*, 28:127–162.

[de Kleer, 1990] de Kleer, J. (1990). Using crude probability estimates to guide diagnosis. *Arificial Intelligence*, 45:381–392.

[de Kleer and Williams, 1989] de Kleer, J. and Williams, B. (1989). Diagnosis with behavioural modes. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, volume 2, pages 1324–1330.

[deKleer and Williams, 1987] deKleer, J. and Williams, B. (1987). Diagnosing multiple faults. *Artificial Intelligence*, 32:97–129.

[Doyle, 1979] Doyle, J. (1979). A truth maintenance system. *Artificial Intelligence*, 12:231–272.

[Durkin, 1994] Durkin, J. (1994). *Expert Systems: Design and Development*. Prentice Hall.

[Feret and Glasgow, 1997] Feret, M. and Glasgow, J. (1997). Combining cased-based and model-based reasoning for the diagnosis of complex devices. *Applied Intelligence*, 7:57–78.

[Fikes and Farquhar, 1999] Fikes, R. and Farquhar, A. (1999). Distributed repositories of highly expressive reusable ontologies. *IEEE Intelligent Systems*, 14(2):73–79.

[Gärdenfors, 1992] Gärdenfors, P., editor (1992). *Belief Revision*. Number 29 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.

[Gruber, 1993] Gruber, T. (1993). Towards principles for the design of ontologies used for knowledge sharing. Technical report KSL93-04, Stanford University, Knowledge Systems Laborator.

[Guarino and Giaretta, 1995] Guarino, N. and Giaretta, P. (1995). Ontologies and knowledge bases: Towards a terminological clarification. In *Towards Very Large Knowledge Bases*. IOS Press.

[Guarino and Giaretta, 1997] Guarino, N. and Giaretta, P. (1997). Understanding, building and using ontologies: A commentary to "Using Explicit Ontologies in KBS Development". *International Journal of Human and Computer Studies*, 46:293–310.

[Hickman et al., 1989] Hickman, F., A, and B (1989). *Analysis for Knowledge-based Systems – A Practical Guide to the KADS Methodology*. Ellis Horwood.

[Janikow, 1998] Janikow, C. Z. (1998). Fuzzy decision trees: Issues and methods. *IEEE Transactions on Systems, Man, and Cybernetics – Part B*, 28(1):1–14.

[Jennings, 1999] Jennings, N. R. (1999). Agent-Oriented Software Engineering. In Garijo, F. J. and Boman, M., editors, *Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World : Multi-Agent System Engineering (MAAMAW-99)*, volume 1647, pages 1–7. Springer-Verlag: Heidelberg, Germany.

[Kasabov, 1996] Kasabov, N. K. (1996). *Foundations of Neural Networks, Fuzzy Systems, and Knowledge Engineering*. MIT Press.

[Kingston, 1992] Kingston, J. (1992). Pragmatic kads: a methodological approach to a small knowledge based systems project. *Expert Systems*, 4.

[Langley, 1997] Langley, P. (1997). Machine learning for intelligent systems. In *Proceedings of the 14th National Conference on Artificial Intelligence*, Providence, RI.

[Lee, 1990] Lee, C. (1990). Fuzzy logic in control systems: Fuzzy logic controller - parts i and ii. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(2):404–435.

[Minsky, 1985] Minsky, M. (1985). *The Society of Mind*. Simon and Schuster, NY.

[Mitchell, 1982] Mitchell, T. (1982). Generalization as search. *Artificial Intelligence*, 18:203–226.

[Mitchell, 1997] Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill Companies.

[Quinlan, 1993] Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann.

[Russell and Norvig, 1995] Russell, S. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice Hall.

[Schreiber et al., 1995] Schreiber, A. T., Wielinga, B. J., and Jansweijer, W. H. J. (1995). The KACTUS view on the 'O' word. In *IJCAI Workshop on Basic Ontological Issues in Knowledge Sharing*.

[Schreiber et al., 1999] Schreiber, G., Akkermans, H., and Anjewierden, A. (1999). *Knowledge Engineering and Management: The CommonKADS Methodology*. MIT Press.

[Shoham, 1993] Shoham, Y. (1993). Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92.

[Sloman, 1996] Sloman, A. (1996). What sort of architecture is required for a human-like agent? Invited talk at Cognitive Modeling Workshop, AAAI96, Portland Oregon.

[Struss and Dressler, 1989] Struss, P. and Dressler, O. (1989). Physical negation - integrating fault models into general diagnostic engine. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, volume 2, pages 1318–1323.

[Uschold, 1998] Uschold, M. (1998). Knowledge level modelling: concepts and terminology. *The Knowledge Engineering Review*, 13:5–29.

[W. Hamscher and de Kleer, 1992] W. Hamscher, L. C. and de Kleer, J. (1992). *Readings in Model-based Diagnosis*. Morgan Kaufmann.

[Winograd, 1987] Winograd, T. (1987). A language/action perspective on the design of cooperative work. *Human-Computer Interaction*, 3(1):3–30.