Formal explorations of knowledge intensive tasks
Jansen, M.G.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Formal Explorations of Knowledge Intensive Tasks

Machiel Jansen
Formal Explorations of Knowledge Intensive Tasks

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de rector Magnificus
prof. mr. P.F. van der Heijden
ten overstaan van een door het college voor promoties ingestelde
commissie, in het openbaar te verdedigen in de Aula der Universiteit
op donderdag 5 juni 2003, te 10.00 uur
door Machiel Gerhard Jansen
geboren te Heemsstede
The research in this thesis has been carried out under the auspices of SIKS, the Dutch Graduate School for Information and Knowledge Systems.

SIKS
SIKS-dissertation series nr: 2003-07
# Contents

**Preface**

1 **Introduction**  
1.1 Background ................................................................. 1  
1.2 The Problem ............................................................... 3  
1.3 Approach ........................................................................ 4  
1.4 Contents of the thesis .................................................... 5  

2 **Rocky III - Round 1 A Progress Report**  
2.1 Context: Sisyphus-III ...................................................... 7  
2.2 Approach and Scope ....................................................... 8  
2.3 Data about the Knowledge Engineering Process ................. 8  
2.4 Ontologies ...................................................................... 11  
2.5 Classification Method ..................................................... 16  
2.6 Demonstrator .................................................................. 21  
2.7 Discussion ..................................................................... 23  

3 **Adapting tableaux for classification**  
3.1 Introduction ................................................................... 25  
3.2 Characterizing Classification .......................................... 25  
3.3 Tableaux ......................................................................... 27  
3.4 Tableaux for classification ............................................. 27  
3.5 Discussion ..................................................................... 33  

4 **A Spectrum of Classification Methods**  
4.1 Introduction ................................................................... 35  
4.2 Classification in terms of attributes ................................. 36  
4.3 Relaxing constraints ....................................................... 42  
4.4 Allowing missing attributes ........................................... 46  
4.5 Combining inconsistencies and missing attributes ........... 47  
4.6 Discussion ..................................................................... 48  
4.7 Acknowledgement .......................................................... 49  

5 **Using strict implication in background theories for abductive tasks**  
5.1 Introduction ................................................................... 51  
5.2 Abduction and representation ........................................ 52  
5.3 Pragmatic issues .............................................................. 53  
5.4 An alternative approach ................................................ 55  
5.5 Strict implication ............................................................. 56
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.6</td>
<td>Strict abduction</td>
<td>57</td>
</tr>
<tr>
<td>5.7</td>
<td>Analytical tableaux for abduction</td>
<td>58</td>
</tr>
<tr>
<td>5.8</td>
<td>Strict implication in T</td>
<td>59</td>
</tr>
<tr>
<td>5.9</td>
<td>Issues revisited</td>
<td>60</td>
</tr>
<tr>
<td>5.10</td>
<td>Discussion</td>
<td>62</td>
</tr>
<tr>
<td>6.1</td>
<td>A dynamic account of knowledge level tasks</td>
<td>63</td>
</tr>
<tr>
<td>6.2</td>
<td>Introduction</td>
<td>63</td>
</tr>
<tr>
<td>6.3</td>
<td>The dynamics of tasks</td>
<td>65</td>
</tr>
<tr>
<td>6.4</td>
<td>Tasks and models</td>
<td>70</td>
</tr>
<tr>
<td>6.5</td>
<td>Logical Dynamics</td>
<td>71</td>
</tr>
<tr>
<td>6.6</td>
<td>Representing knowledge dynamically</td>
<td>78</td>
</tr>
<tr>
<td>6.7</td>
<td>Describing problem solving</td>
<td>85</td>
</tr>
<tr>
<td>6.8</td>
<td>Classification criteria compared</td>
<td>92</td>
</tr>
<tr>
<td>6.9</td>
<td>Discussion</td>
<td>93</td>
</tr>
<tr>
<td>7.1</td>
<td>Towards a task ontology for knowledge-intensive assignment problems</td>
<td>95</td>
</tr>
<tr>
<td>7.2</td>
<td>Introduction</td>
<td>95</td>
</tr>
<tr>
<td>7.3</td>
<td>A basic task ontology for assignment problems</td>
<td>96</td>
</tr>
<tr>
<td>7.4</td>
<td>A basic classification of assignment problems</td>
<td>100</td>
</tr>
<tr>
<td>7.5</td>
<td>Constraints and preferences</td>
<td>103</td>
</tr>
<tr>
<td>7.6</td>
<td>Using GCNs in the assignment ontology</td>
<td>106</td>
</tr>
<tr>
<td>7.7</td>
<td>Representing preferences</td>
<td>111</td>
</tr>
<tr>
<td>7.8</td>
<td>Combining preferences: criteria and combination functions</td>
<td>114</td>
</tr>
<tr>
<td>7.9</td>
<td>Abstractions in preferences</td>
<td>116</td>
</tr>
<tr>
<td>7.10</td>
<td>Case study: Sisyphus I</td>
<td>117</td>
</tr>
<tr>
<td>7.11</td>
<td>Discussion</td>
<td>123</td>
</tr>
<tr>
<td>8.1</td>
<td>Formalizing group assignments</td>
<td>125</td>
</tr>
<tr>
<td>8.2</td>
<td>Introduction</td>
<td>125</td>
</tr>
<tr>
<td>8.3</td>
<td>Grouping</td>
<td>126</td>
</tr>
<tr>
<td>8.4</td>
<td>Abstraction</td>
<td>127</td>
</tr>
<tr>
<td>8.5</td>
<td>Formalizing abstraction</td>
<td>127</td>
</tr>
<tr>
<td>8.6</td>
<td>Partitions as formal structures</td>
<td>129</td>
</tr>
<tr>
<td>8.7</td>
<td>Constraints and abstractions</td>
<td>133</td>
</tr>
<tr>
<td>8.8</td>
<td>The nature of grouping</td>
<td>136</td>
</tr>
<tr>
<td>8.9</td>
<td>Merging and splitting blocks</td>
<td>137</td>
</tr>
<tr>
<td>8.10</td>
<td>Finding groupings for the constraints</td>
<td>138</td>
</tr>
<tr>
<td>8.11</td>
<td>Combining constraints</td>
<td>143</td>
</tr>
<tr>
<td>8.12</td>
<td>An update system for grouping</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>Discussion</td>
<td>148</td>
</tr>
</tbody>
</table>
Contents

9 Conclusions 149
  9.1 A review of the previous chapters ........................................... 149
  9.2 How do we represent these knowledge-level conceptualizations in a formal way? 151
  9.3 How can we use these representations to acquire a better understanding of the task? 152
  9.4 Synthesis and discussion .......................................................... 152

Appendix A 157
  I Statement of the sample problem .............................................. 157
  II Data on people and offices ...................................................... 157

Samenvatting 171
Preface

At this place I would like to thank those people who were at some time or other involved in the process of writing this thesis, whether they were aware of it or not.

First of all I would like to thank my promotor Bob Wielinga for the patience and understanding with which he listened to many of my unorthodox ideas. Equally important has been my co-promotor Guus Schreiber whose corrections, decisions, support and criticism have been very helpful indeed.

Annette ten Teije and Frank van Harmelen have supported me at different stages, particularly when writing chapter 4.

I would also like to thank those people who at some time or other spent time with me in the same room at SWI. When asked some of them may (falsely) claim I was never there. I thank Pascal Beys for his company and not having an outrageous accent. Also thanks to Gaston Heimeriks for not discussing the outside world, and sharing his highly objective analyses of the performances of Feyenoord. Very pleasant company has been Vania Bessa Machado. She has been very supportive by not listening to my repeated claim that Trying is the first step towards failure.

Also thanks to Remco Straatman, Michael Boelgakov, Jan Wielemaker, Homer Simpson, Barbara Dubbeldam, Edward Gibbon, Arthur Duineveld, V.S. Naipaul, Roland Groen, Thomas Pynchon, Arjen van Dijk, Erika Gasataya, William Shakespeare and all those working at SWI.
1

Introduction

1.1 Background

Over the years Knowledge Engineering (KE) has identified a number of problem types known as *tasks* such as classification, design, diagnosis, planning etc (see Puppe [71] for an overview). Each task covers a range of problems which can vary considerably from one another. As an example of a task one can think of diagnosis. Within diagnosis there are several problem variations. For example, performing abduction while assuming that there is only one fault which accounts for a malfunction, is known as *single fault diagnosis*, whereas the absence of such an assumption is characteristic of *multiple fault diagnosis*.

There is general consensus [71, 82] about distinguishing two categories of tasks: *analytical* and *synthetic* tasks. The first covers problem types such as classification and diagnosis, and involves reasoning about an 'existing system' such as the object to be classified in the case of classification, or the artefact to be diagnosed in diagnosis. In synthetic tasks the 'system' has to be *constructed*. Hence planning involves the construction of a plan, assignment involves the construction of an assignment, etc.

Conceptualizations of a task are usually given in the form of an *ontology* of the task. A task ontology provides a specification of the vocabulary of a task, which can be used to formulate problems which belong to that task. For example, a task ontology for classification provides a specification for notions as "class", "observation", "explains", "attribute" etc. These task ontologies are often presented in an informal, or semi-formal way [64] and leave considerable room for interpretation.

With each task one can associate several methods, known as *task methods* [82], together with typical domain knowledge schemas or conceptualizations. In the commonKADS methodology [82] each task has been given a *task template*. This template can be seen as a prototypical method for the task which can be taken as a starting point for the development of more specific task methods.

Ideally, when trying to solve a knowledge intensive problem, one first determines to which task it belongs. Next, one inspects the template and adapts it where needed. Finally, one instantiates the method by supplying the various types of knowledge in the proper representational format.

However, such an ideal picture is often cumbersome in practice. Therefore a better understanding of specific tasks is still needed. This thesis focuses on this subject.

Tasks are generally described as consisting of three parts: A specification of the knowledge used, the types of input and output, and a specification of the *goal* to be reached.

The task method describes how a goal is realized through a decomposition into substructures, together with some flow of control. These substructures are either tasks themselves (subtasks) or
inferences (primitive elements of which the functionality is in no need of further decomposition). A control structure describes in what order these should be executed.

The default method for a task corresponds roughly to the notion of Problem Solving Method (PSM) in the literature [25, 84, 28, 11]. Examples of PSMs are generate & test, hill climbing and propose & revise. The main difference is that PSMs are usually more general, and as such they are not directly linked to the application of a task. Task methods can therefore best be seen as instantiated PSMs [82]. Hence, a task method describes a general method for problems belonging to the task, using the vocabulary specified in the task ontology. PSMs are usually more generic and can use task-independent vocabulary. In order to instantiate a PSM one has to map the vocabulary of the PSM to the one of the task ontology.

It should be stressed that the identification of tasks, and the description and use of the methods takes place on a high level of abstraction. They are the results of experience of designing and building knowledge based systems, rather than the product of formal, theoretical investigations. In that sense PSMs are similar to design patterns in the field of object oriented programming [35]. Design patterns are presented as program structures which have proved their value in practice and which can often be identified in complex software systems. Therefore it is worthwhile to describe and classify them and make them available for future re-use.

Similarly, the motivation for characterizing and describing PSMs is usually based on pragmatic ideas. They are not finely tuned algorithms but general methods which are expected to be adapted to the peculiarities of a given problem. As such they should speed up the design of complex knowledge based systems.

Formal methods have been used to present analyses of tasks from a “top-down” perspective. In such an approach tasks are usually decomposed and then a specification for the substructures, or inferences, is given together with some ‘glue’. An example is the work of Aben [1]. His approach consists of breaking up tasks into a number of inferences first. He then presents a unified, formal framework in which all these inferences are described in detail.

Another example of an approach which aims at presenting a single, unified description of tasks, is the work on the Unified Problem Solving Language (UPML) [26]. This architecture provides one general framework for the description, development and re-use of task methods. In this project one has experimented with formal languages as a specification language for some of the notions used in the architecture. Related to this is the work of Fensel [25] which presents a unified view on PSMs from a top-down perspective.

Such a unified view on tasks is often motivated by the prospect of enhancing the development of knowledge based systems. In contrast to this, one can distinguish a more theoretical approach which is concerned with the formal analysis of individual tasks. In such an approach one analyzes tasks individually and different tasks can be described in different formalisms.

As an example of a more theoretical task-oriented approach we mention the work of ten Teijen [88] on diagnostic problem solving. Here some parameters for diagnostic methods are identified. By varying the instantiations of these parameters one can generate a spectrum of methods for diagnosis.

Also, in the same work, the formalism of approximate reasoning [79] was used, and adapted to describe characteristics of diagnostic methods. This approach differs from the ones above since it focuses on a single task only. This is an example of how an existing formalism can be employed to describe the general reasoning structure of a single task. However, the motivation behind this work consisted of the use of the analysis for the automatic configuration of PSMs in diagnosis.
Another task which features quite heavily as the subject of theoretical research is planning. As an example, we mention the planning-as-satisfiability approach of Kautz & Selman [52] and the relation of planning with linear logic [61]. Such investigations provide insights into the task which are often missing in the pragmatic approach. However, such analyses are seldom linked to the methodologies of KE, and often do not make use of task ontologies and ignore problem variations within the task.

Many of the formal descriptions of tasks within the field of KE have a rather static nature. Often, first or second order predicate logic is used to capture the functionality of the task and inferences. Attempts to provide a dynamic description of knowledge modelling have been made. An example is the ML² language [92] which provided a formal description of the KADS knowledge model. Dynamic logic was used here to describe the dynamics of the knowledge models. However, in this language it is difficult to give a clear and intuitive account of the various stages of the problem solving process.

In this thesis we explore in what ways existing formalisms can be used to analyse tasks within the framework developed by the pragmatic approach. By this we mean that we will make use of ontologies and are aware of problem variations within the task. We would like to characterize the task in knowledge-level [67] terms. That is, we give a description of which knowledge is present before, during and after solving a problem which belongs to a certain task. Hence, we will present a static as well as a dynamic account of knowledge-level problem solving. The aim is not to develop new, or faster Problem Solving Methods but to acquire a better insight into the nature of tasks.

1.2 The Problem

In general terms the problem statement of the thesis can be formulated as follows:

**HOW CAN WE GIVE A KNOWLEDGE-LEVEL CHARACTERIZATION OF THE PROPERTIES OF KNOWLEDGE INTENSIVE TASKS?**

The aim is not to develop a new logical language or formalism, but to explore how existing formalisms and approaches can be used to acquire new insights regarding knowledge level task specification.

The problem statement can be refined by introducing three related questions:

1. **WHAT DOES A CONCEPTUALIZATION OF A TASK LOOK LIKE?**
   By a "conceptualization of a task" we mean a characterization of the notions involved in problem description. The conceptualization of a task should include a task-ontology in which the task-specific vocabulary should be specified. In addition it should include some (not necessarily all) criteria for a solution to some typical problems which belong to the task. Finally, the conceptualization of a task may contain a task-template, or some general description of what problem solving for that task would look like. Note that a conceptualization of the task is often done on a high level of abstraction and is very much a knowledge-level construction.

2. **HOW DO WE REPRESENT THESE KNOWLEDGE-LEVEL CONCEPTUALIZATIONS IN A FORMAL WAY?**
The next step is to choose an adequate formal representation of these conceptualizations in order to present a thorough characterization of a task. Such a formalism forces one to be precise about the knowledge-level constructs. In addition we would like that the properties of the formalism would tell us something about the nature of the task, and that proof methods for the formalism have some relation with problem solving of the task.

3. **How can we use these representations to acquire a better understanding of the task?**
   As formal specifications of knowledge-level conceptualizations force one to be more precise, one may discover that new distinctions can be made and new variations can be formulated. For example, given some initial criteria for solutions a formal analysis may lead to a more rigorous and systematic distinction between various criteria. This can lead to the systematic generation of a spectrum of the problems which are covered by the task description.
   
   Another point of attention is that a formalism can have several proof methods associated with it. It is interesting to see whether one of these could be used, or adapted to describe task methods.

1.3 **Approach**

In order to answer the questions above we will focus on a limited selection of specific tasks. Some tasks like diagnosis [8, 21] and planning [4] have been extensively described and analyzed in the literature. We therefore focus on tasks which have been less extensively explored, like classification and assignment. We have taken care to include both an analytical task (classification) and a synthetic one (assignment). The results of the explorations of these tasks will be generalized to some extent in order to answer the questions in the previous section.

We want to stress that we want to give a knowledge-level description [67] of the task in a dynamic, formal fashion. Another point to note is the way we use logic, and formal methods in general. Our approach is aimed at the understanding of a knowledge-level specification of tasks, and formal methods are used as means to that end, not as an end in themselves. Hence, we will be quite pragmatic in the use of existing formalisms.

The steps which we will take to arrive at the answers to the research questions are the following:

- **Conduct an informal study of a knowledge-intensive task at the knowledge level.**
  As an introductory step, a task is selected and analyzed in detail, using the traditional semi-formal specification methods used in knowledge engineering. Special attention should be paid in this exercise to the nature of the ontologies and Problem Solving Methods involved. This should give adequate input to the needs for logical formalizations.

- **Show how an adequate formalism for a task is chosen.**
  The result of the previous step will serve as input for the choice of an adequate logical representation of task-specific reasoning.

- **Provide a specification of the range of problems within a task.**
  We will provide a spectrum of problems for two tasks, one *analytic*, the other a *synthetic*
task. This should provide a specification of how problems can vary within a task and how they can be elegantly described, classified and compared. This description of problem variations should be done in terms of the elements of the task ontology.

- **Provide a more general framework for the dynamic description of task-specific reasoning.**

A more dynamic approach to the description of knowledge intensive tasks will be presented. Problem solving will be described in terms of increasing knowledge about possible solutions. We will use the notions of 'knowledge state' and 'state transition' in these descriptions.

### 1.4 Contents of the thesis

The thesis will be divided into nine chapters. Chapters 2, 4, 5 are devoted to analytical tasks, whereas 7 and 8 are about synthetic tasks. Chapter 6 contains the description of a framework aimed at a dynamic description of knowledge intensive tasks based on the idea of information flow.

Finally, chapter 9 contains the conclusions and answers to the research questions formulated in the current chapter.

**Chapter 2 - Rocky III** This chapter describes the results of research performed as part of the Sisyphus III project [46]. Researchers in the knowledge engineering community were asked to develop a classification system for igneous rocks. Comparing the various approaches taken by different researchers was the research goal of the project. This chapter presents a contribution to the Sisyphus project. As part of the thesis the chapter provides an introduction into the use of traditional specifications of ontologies and problem solving methods. More specifically the task ontology and classification methods described here, can serve as input for chapter 3.

**Chapter 3 - Adopting tableaux for classification** In the chapter 2 a classification problem was looked at in detail. In this chapter we focus at the classification task itself and give a more general characterization. It is shown how classification problems can be represented in such a way that they can be solved with (manipulated) analytic tableaux. Two problem solving methods for classification are discussed: weak and strong classification. For both methods procedures in terms of tableaux are presented. The similarity between strong classification and abduction is highlighted.

**Chapter 4 - A spectrum of classification methods** In the previous chapters several variations of classification were described. This leads to the question what other problem variations are possible. This chapter describes the construction of a spectrum of classification criteria, built up from elementary building blocks which are part of the ontology of the task.

**Chapter 5 - Using strict implication in background theories for abductive tasks** Domain knowledge in abductive tasks is often represented in classical logic, using the material implication symbol ‘→’ in a specific way. We explore the objections against the use of material implication (first raised by Lewis in the 1930's [56]) and see if these affect the use
and intended meaning of the representation. As one of the possible alternatives we look at abduction with *strict* implication.

**Chapter 6 - A dynamic approach to specifying analytic tasks**  
In this chapter we provide a more general analysis of tasks in terms of knowledge about the solution. We show how problems within a task can be given a knowledge level characterization in a dynamic way. Problem solving is described as changing knowledge on the basis of new information and pro-active behaviour.

**Chapter 7 - Towards an ontology for knowledge intensive assignment problems**  
In this chapter we will provide a task ontology for assignment problems. The main emphasis is on the characterization of different forms of constraints and preferences. The ontology covers both assignment problems from the field of Constraint Satisfaction, and the family of Stable Marriage problems [44].

**Chapter 8 - Formalizing group assignments**  
We look at a subproblem of assignment known as *grouping*. This strategy is particularly interesting in combination with the use of abstractions in the domain knowledge. We explain in detail how abstractions and groupings can be treated as partitions of a set.

**Chapter 9 - Conclusions**  
This chapter contains the conclusion of the thesis and answers the research question formulated above.
This chapter was written together with A. Th. Schreiber and B. J. Wielinga and was published in the Proceedings of the Knowledge Acquisition Workshop (KAW) in 1998 [46].

This chapter describes the results of research performed as part of the Sisyphus-III project. The project started by inviting researchers in the knowledge engineering community to develop a classification system for igneous rocks. Comparing the various approaches taken by different researchers was the research goal of the project. The present chapter describes the contribution of the author(s). As part of the thesis the chapter provides an introduction into the use of traditional specifications of ontologies and problem solving methods. More specifically the classification methods described can serve as input for the next chapters.

2.1 Context: Sisyphus-III

The goal of the third Sisyphus project Rocky-III was to compare knowledge engineering methodologies and techniques in terms of effectiveness, efficiency and scalability. Participants were provided with knowledge acquisition material in three phases and were required to keep a log of their activities.

In the first phase, participants were invited to build a system that was able to classify sixteen types of igneous rock. The system also had to function as a tutorial aid in this field. Knowledge acquisition material was intended to be released in stages. The first release included a detailed problem specification together with a first set of acquisition material which was common to all participants. Future releases were planned but never materialized, due to the limited number of participants.

A second release of material was to become available later, together with an announced expansion of the problem set to be solved. Finally, a third stage was to announce a significant extension to the functionality of the system that should be incorporated if possible. In the following text we still expected later releases of knowledge acquisition material and anticipated on the the possible extension of the problem.
2.2 Approach and Scope

This chapter constitutes a progress report on the work done by the authors in Phase 1. Our work on phase 1 effectively started on June 1, 1997. The central research question in this chapter is as follows:

"Can we prove that additional work on ontology construction in the early phases of a project, pays off in later phases?"

The pay-off can be of different types, e.g. shorter development and/or maintenance time, higher quality and/or reliability.

This research question fits well with the overall Sisyphus-III aims. Operationalizing this question in terms of Sisyphus-II implies that we intended to spend more time in Phase 1 on ontology development, but that we expect that this extra work will pay off in phase 2 and 3. The work at UvA will be somewhat biased towards the ontological aspects of the problem domain.

Our practical goals in the first three months (the period on which we report here) have been twofold:

1. To develop a demonstrator application for rock classification that meets the aims of phase 1.
2. To develop a number of ontologies more or less independent of the demonstrator.

The basic idea is that in developing the ontologies we do not want to be biased too much by the current system. Of course, parts of the ontologies are used in the demonstrator, but actual ontology usage was not a *conditio sine qua non* for ontology development. The scope of the ontologies in Sec. 2.4 is therefore broader than the domain knowledge used by the demonstrator described in Sec. 2.6.

The overall knowledge engineering approach used is CommonKADS as it is documented in the draft textbook about this approach [82]. This chapter is structured as follows. First, we provide data about the knowledge engineering process, in particular on the activities performed, the knowledge-acquisition material used, and the time spent. The three following sections describe the actual work done. In Sec. 2.4 we discuss the different ontologies we developed. Sec. 2.5 describes the method for classification that we used. It turned out we could use a simple pruning method, but that we had to employ that method in a recursive way. Sec. 2.6 shows fragments of the demonstrator application. In the final section we discuss a number of issues that came up during this work.

2.3 Data about the Knowledge Engineering Process

In this section we present an overview of the activities conducted in the context of the Sisyphus project. Time is measured in days as if performed by one person. Work on the Sisyphus project was done by one person most of the time. Building the ontologies and discussing them involved two or three people. All persons were also engaged in other activities not related to the Sisyphus project.
Domain familiarization

A first preliminary study of the domain was started in June 1997. This consisted of studying the KA material provided by the Sisyphus team and examining works like Schumann [85]. The other domain texts mentioned at the Internet site of Sisyphus and the material from the Open University could not be obtained.

Instead, the Internet was used as an information source and Raymond [73] and McKenzie [59] were consulted. The information found on the Internet was often helpful but incomplete. We did not find any ontologies or software which could be helpful in the classification of rocks. We did however stumble upon a program called MINID [74], a DOS-program which assists in the identification of minerals in thin sections. We used this program as the basis for the thin sections program in the demonstrator.

Having no previous experience in the field of petrology the investigation into the domain of igneous rocks and minerals took the largest amount of time. The study was conducted by one person and took a total of 12 days of full time work.

Building ontologies

An ontology of rocks was built based on the information present in Schumann [85]. This book describes the classification of igneous rock according to the diamond-shaped Streckeisen-diagram. This diagram was not easy to describe in common ontology-representational formalisms. The inclusion of graphical representations in ontologies is a point worth discussing in more depth.

The mineral ontology could be compiled from various sources. The mineral descriptions were well described in Raymond [73] and could be gathered from the MINID program and a good introduction of investigating minerals in thin section was given by MacKenzie [59].

The texture ontology was compiled within half a day and based on information found at the Internet site of the University of British Columbia (http://www.science.ubc.ca/ geol202/). Later, corrections were made based on Raymond [73].

The ontologies were built by two persons and together took about 10 days of full-time work.

Discussing ontologies

The ontologies were discussed on a regular basis. Since the rock, mineral and texture ontologies were quite straightforward, discussions centered on the construction of the classification ontology. The appropriate representation of the diamond-shaped Streckeisen-diagram was also discussed. Discussions were mainly among the three authors of this chapter and taken together took 7 days.

Classification methods survey

A brief survey of classification methods was conducted. At this stage both Stefik [87] and Wielinga [97] proved to be very helpful. A simple prototype program for apple classification implementing the pruning method was built in order to explore this method in more detail. Some alternatives for the attribute selection step were studied, using also ideas from machine-learning algorithms. This activity took 6 days.

Building the demo-system

The demonstrator described in Sec. 2.6 was built in 7 days by one person. The thin section
part of the program was based on the MINID program. The attributes and values of this program were used and a running prototype with graphical user-interface was written within a day. Part of the program-code of the demonstrator was also used in classification of other domains.

Table 2.1 summarizes the time spent on the five knowledge-engineering activities performed.

<table>
<thead>
<tr>
<th>KE Activity</th>
<th>Time spent in person-days</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain familiarization</td>
<td>12</td>
</tr>
<tr>
<td>Building ontologies</td>
<td>10</td>
</tr>
<tr>
<td>Discussing ontologies</td>
<td>7</td>
</tr>
<tr>
<td>Classification methods survey</td>
<td>6</td>
</tr>
<tr>
<td>Building demo-system</td>
<td>7</td>
</tr>
<tr>
<td>Total</td>
<td>42</td>
</tr>
</tbody>
</table>

Table 2.1
Time spent in Round 1 on the knowledge-engineering activities performed.

2.3.1 Knowledge acquisition material

The analysis of the knowledge acquisition material was brief after it was discovered that information in the interviews was often incorrect or unreliable. An example from the first laddered grid from the Sisyphus material will illustrate this:

... these are sort of metamorphic ones which you're going to get a large grains in a fine-grained sort of matrix thing.

This is quite misleading. What the “expert” is referring to is a so-called *porphyritic texture* in which you find larger grains scattered in a *matrix* of finer grains. This texture is common in many igneous rocks and in this context has nothing to do with metamorphic processes.

The question therefore arises in what cases to use material acquired with the help of knowledge acquisition (KA) techniques and when not. Generally KA methods are a good source of information if one is after the cognitive aspects of expert reasoning. In complex domains where problems are often computational intractable KA methods are a good way of discovering heuristics.

To gather information about domains where the need for heuristic methods is not immediately apparent, the use of KA methods in the initial stage (such as in phase 1 of Sisyphus) is doubtful. The material provided by laddered grids and think-aloud protocols is not well-suited to act as as a first guide to a domain.

There are two additional points to be made here: Firstly, it is hard to interpret the correctness of the information provided by the expert if the Knowledge Engineer (KE) has no or little knowledge of the domain. Experts may make mistakes, or give false information. Mistakes of course can be very helpful in discovering heuristics or cognitive modeling. In order to make a useful interpretation of the KA material one already needs some understanding of the domain.

Secondly, experts may try to make clear to the Knowledge Engineer information that is well-documented elsewhere. For example it does not make much sense to interview an expert chess-player if one is after the rules of the game. If however one chooses to do so one has to rely heavily
on the verbal capabilities of the expert. In case of misinterpretations or omissions one might end up with the wrong set of rules. In this case it is far more efficient to read an introductory book on chess and try to play the game self first. This seems to be true for all knowledge which is objective and well-documented.

For the domain of igneous rocks similar remarks can be made. The science of rocks (petrology) is well-developed and inspection of the knowledge-acquisition material shows that experts at least try to classify rocks according to known classification schemes. In addition the KA material contains quite a few contradictions and errors. In order to get a good grasp of the domain of rock classification it seems more helpful to read additional introductory material on the subject of petrology than to dive into grids and protocols. Raymond [73] offers a systematic introduction to this field and was used to acquire an understanding of the domain. Much of the information on the rocks themselves was derived from the Dutch edition of Schumann [85].

2.3.2 A note about missing rocks

The rock adamellite was mentioned in neither Raymond [73], Schumann [85] nor MacKenzie [59]. A clear description could not be constructed from the official Sisyphus knowledge-acquisition material either. Consulting some other books on igneous rocks finally resulted into an exact description.

The rock Kentallenite could not be found in textbooks or on the Internet. This rock also puzzled some experts as a quote from the first laddered grid of the Sisyphus material shows:

Ke... (Kentallenite) I can’t even say the word, I’ve never heard of that one

A post on the newsgroup sci.geo.geology finally resulted in this quotation taken from Howell [41].

...a dark monzonite composed of approximately equal amounts of augite, olivine, orthoclase and plagioclase, with biotite, apatite and opaque oxides”

This definition however is not in line with the official UIGS classification method and can therefore not be classified with the help of our demonstrator. As there are several classification schemes in use for identifying igneous rocks by geologists, there are several rock-names which are meaningful only to a few.

2.4 Ontologies

We describe several ontologies for this application domain. These ontologies can be seen as belonging to two categories of ontologies:

Application-domain ontologies

These ontologies abstract as much as possible of the task, and are aimed at representing the static structure of a domain. In the case of igneous rocks, we expect to find descriptions of rocks and their elements.

Domain-specific ontologies, such as a rocks ontology, can also be generalized to higher-level conceptualizations, such as an ontology about object types: e.g. natural inorganic
objects. We have not done any work on such ontologies yet within this context: Instead we have concentrated on the domain-specific ontologies that stood out as useful schematic descriptions.

**Application-task ontologies**

A second category of ontologies describe the way in which we conceptualize the world from the functional perspective, i.e. the task we want to perform. In a publication about the previous Sisyphus project [83] we argued that it is useful to distinguish between at least two types of ontology within this category:

**Task-specific** ontology is an ontology which captures the knowledge structure always encountered with a certain type of task. The ontology developed for the VT domain by Gruber *et al.* [38] can be seen as an example of such an ontology for the configuration task.

**Method-specific** ontology is an ontology that contains precisely those knowledge types needed for a certain problem-solving method to work. Often, such a method-specific ontology can be represented as having a task-specific ontology as its core part to which a number of method-specific extensions are added. It can be shown that this enables knowledge-base reuse [76, 83].

In the Rocky domain we have concentrated on the task-specific ontology for classification.

This separation between domain and task ontologies lies at the heart of the PROTEGE-II approach [90]. We follow their view that the ontology of a particular application is in fact an amalgamation of ontologies from both categories, and that it is wise to keep these explicitly separate by generating an application ontology from explicit mapping relations between the ontologies [36].

---

**Figure 2.1**
Overview of the ontologies developed in the context of phase 1.

Figure 2.1 gives a graphical overview of the types of ontologies involved in this study. In this section we give a brief description of the domain-specific and the task-method-specific ontologies (depicted in the boxes on the left and the right). We briefly mention the issues involved in the ontology mappings (the central box in Figure 2.1), but this is really a subject we have not tackled yet for this application.
2.4.1 Domain-specific ontologies

Rocks This ontology contains a sub-type hierarchy of rock types. Figure 2.2 shows fragments of this hierarchy. Each rock is characterized by a number of properties (e.g. texture, grain-size and colour). A rock can have a mineral-content relation with a mineral. This relation is reified into a concept1 (the dotted line starting from the solid relation line in Figure 2.2) and can be used to represent information about mineral-content percentages. The mineral-content-constraint is an example of the use of a special CommonKADS modelling construct RULE-SCHEMA. It denotes a set of expressions about one or more other constructs, in this case about mineral-content. These constraints provide us with a way of describing the Streckeisen-diagram. For example, the instances of mineral-content-constraints for the rock syenite are represented as follows:

\[
\text{mineral-content(syenite, alkali-feldspar).percentage} \geq 65; \\
\text{mineral-content(syenite, plagioclase).percentage} \leq 35; \\
\text{mineral-content(syenite, quartz).percentage} \leq 20; \\
\text{mineral-content(syenite, foids).percentage} \leq 10;
\]

These expressions are a textual representation of the knowledge about mineral percentages in syenite represented in the Streckeisen diagram. Minerals are described in a separate ontology which is imported into the rocks ontology.

![Graphical overview of the rocks ontology. Arrows with open heads are used to denote a subsumption relation. Only a few sub-classes of rock are shown in the figure. The ontology uses the "minerals" ontology. A key concept is mineral-content. This can be used to indicate the amounts of minerals present in a certain rock. The mineral-content-constraint is an example of the use of a special CommonKADS modelling construct RULE-SCHEMA. It denotes a set of expressions about one or more other constructs, in this case about mineral-content. These constraints provide us with a way of describing the Streckeisen-diagram.](image)

1Relation reification is a powerful modelling construct in modern data-modelling. See for example the ASSOCIATION-CLASS construct in OMT [77].
Minerals  This ontology describes the minerals that may be present in igneous rocks. The attributes of the mineral classes are those attributes that can be used to identify a mineral (within a thin section of an igneous rock) under a microscope. In addition, the crystal structure and the chemical formula are also listed.

Textures  In the texture ontology all features of a rock that are related to the texture of rocks are described in a hierarchic fashion. This includes grain-sizes, grain-types, crystal shapes and habits, fabrics and textures. From this ontology one can completely describe a rock in textural terms. A graphical overview of the texture ontology is shown in Figure 2.3. The leafs of this tree serve as values for their parent-nodes.

2.4.2  Task/method-specific ontologies

An ontology for classification tasks  We have constructed an ontology for classification tasks. A graphical overview of the ontology is shown in Figure 2.4. This ontology has the status of a proposal and will typically be subject of discussion and revision.

The two central concepts in the classification ontology are object-type and descriptor. “Object-type” represents the general category to which objects that need to be classified belong, e.g. apple, rock or art object. A descriptor provides information about instances of the object type. For example, colour and grain size are descriptors of apples. Descriptors are part of a space of descriptors, termed the description-universe which consist of several dimensions. Each descriptor belongs to one dimension. These dimensions are mostly different ways of observing an object. For example, for rocks we distinguish between a macroscopic and a microscopic dimension. For apples, we could have the surface inspection and the internal inspection as two different descriptor dimensions. Each descriptor has an associated value set. The reified relation descriptor-value is used to represent a descriptor-value pair.

Each object type has a number of object classes. For the moment, we did not include a hierarchy of object classes, mainly because we have focused on capturing in this ontology the minimal conceptualizations necessary for classification. The object-instance represents the actual objects being classified, i.e. a particular apple or a particular rock. Most of the classification knowledge is represented through the rule schema class-constraint. This rule schema denotes the existence of a set of logical dependencies between object classes and descriptor-value pairs. For example, in the apple domain an instance of this rule schema could be the statement that a James Grieves apple is either green or yellow-green.

In this way one can describe a domain in terms of descriptors, objects and object-classes. Such a representation lies at the heart of several classification tasks, including rock classification. The concept of rock can be modeled as an object type which has several descriptors such as colour and grain-size. The rock type granite is represented as an object class. For granite the value for the descriptor grain-size is restricted by an instance class constraint that tells us that the value fine-grained can be excluded.

Method-specific extensions  The classification ontology should provide the core knowledge for classification problem solvers. The specific methods used in an application will typically have additional domain-knowledge demands. As we will see in the next section, the classification methods applied in our demonstrator make use of additional attribute ordering knowledge. At this moment, we have not built an explicit ontology for these method-specific extensions.
Another way of extending the classification ontology is by making use of hierarchy information in the domain. Currently the classification ontology does not represent such information. The demonstrator too does not make use of such hierarchies present in the domain.

2.4.3 Ontology mapping table

If one constructs and/or reuses different ontologies (e.g. a classification ontology and a rocks ontology) the need arises for methods that support a systematic integration of these ontologies.
such that they can be used in an application. One possible method is an ontology-mapping table. A formal ontology mapping has not been developed at the moment, but will be a central topic of interest for us in continuation of Sisyphus. The basic idea is illustrated by Figure 2.1. Every element in the classification ontology should be mapped on an element in the ontology of the domain, e.g. a rock ontology. **Object type** should be mapped on rock. **Object-classes** are those concepts in the rock ontology that are subsumed by the type rock and are themselves leaves in the subsumption hierarchy. **Descriptors** map on **properties** or **parts** of rock.

### 2.5 Classification Method

#### 2.5.1 Method selection

We selected the default classification method provided by the CommonKADS textbook [82]. The method employs a simple pruning strategy. One starts off with the full set of possible candidates, specifies a feature of interest, obtains its value and “prunes away” all candidates that are inconsistent with the incoming data. This process is repeated until there is only one candidate solution.

The specification of this method is shown in Figure 2.5. The first **while** loop generates the set of candidate solutions. The second **while** loop prunes this set by obtaining new information. The method finishes if one of the following three conditions is true (see the condition of the second **while** loop):

1. A single candidate remains. This class becomes the solution.

2. The candidate set is empty. No solution is found.
psm prune-candidate-set;
  can-realize: classification;
  decomposition:
    functions: generate, specify, match, obtain;
  roles:
    intermediate: candidate, attribute, new-feature,
      current-feature-set, truth-value;
  output: candidate-set;
  control-structure:
    while more-solutions generate(→ candidate) do
      candidate-set := candidate union candidate-set;
    end while
    while new-solution specify((candidate-set → attribute)
      and length candidate-set > 1 do
      obtain(attribute → new-feature);
      for-each candidate in candidate-set do
        match(candidate + current-feature-set → truth-value);
        if truth-value = false;
          then candidate-set := candidate-set subtract candidate;
        end if
      end for-each
    end while
  end psm prune-candidate-set;

Figure 2.5
Pruning method for classification.

3. No more attributes remain for which a value can be obtained. A partial solution is found in the form of the remaining set of candidates.

Figure 2.6 shows the corresponding inference structure. Three inferences are used in this method plus a transfer function for obtaining the attribute value:

**Generate candidate** In the simplest case, this step is just a lookup in the knowledge base of the potential candidate solutions.

**Specify attribute** There are several ways for realizing this inference. The simplest way is to just do a random selection. This can work well, especially if the "cost" of obtaining information is low. Often however, a more knowledge-intensive attribute specification is required.

**Obtain feature** Usually, one should allow the user to enter an "unknown" value. Also, sometimes there is domain knowledge that suggests that certain attributes should always be obtained as one group.
Match  This inference should be able to handle an “unknown” value for certain attributes. The default approach is that every candidate is consistent with an “unknown” value for a certain attribute.

2.5.2 Method customization

The pruning method was customized for use in Rocky by providing three alternative strategies for the attribute-selection inference.

Random choice of attributes
A random choice will work if the information is easily obtainable and the number of attributes is not too large. In this case the user will have to supply far more values to attributes than is needed.

Order attributes
A more sophisticated method is to explicitly order the attributes and turn them into a decision tree. In this case one uses domain knowledge of the form If grain-size is coarse-grained ask about colour. Such explicit attribute ordering information can very often be replaced by a domain-independent method like:

Find an attribute that is present in as many of the classes that are among the possible solutions.
Find most informative attribute
Another way is to choose the most informative attribute. The amount of information of an AV-pair reflects the measure of reduction of possible solutions. The way to calculate this amount is to count the number of binary decisions needed to come to this reduction.

The amount of information of an AV-pair, $I(AVi)$, can be computed as follows:

$$I(AVi) = 2 \log \frac{1}{p(Vi)} = -2 \log p(Vi)$$

In order to compute the amount of information of an attribute (instead of an attribute value-pair) we simply compute the amount of all possible AV-pairs for this attribute and divide it by the number of possible values. This is different from the entropy formula as known in information theory:

$$I(s) = \sum (s_i)I(s_i)$$

This formula weighs the probabilities as chances and is useful in cases similar to flipping an unbalanced coin or learning. The same idea is also used in ID3 [72]. Note that in our case we take the distribution of an AV-pair as it appears in the set of candidate solutions.

All three methods were explored and implemented in the demo-program in a domain-independent way.

2.5.3 Nesting classification methods

The classification of igneous rocks takes place on two levels. First, information from a macroscopic level will be obtained. If this suffices to prune away all but one of the known rock classes there is no need for a microscopic investigation. However a problem arises when one is not sure about some features of the rock that are necessary for a good classification. This problem often occurs when classifying small-grained volcanic rocks. A few quotes from the first interview taken from the Sisyphus material illustrate this issue:

Assuming you’ve got a coarse grained rock, the mineralogy would be very easy to identify (…)
And from your identification of the mineralogy you’ll be able to arrive at a basic chemistry
Now the difference here is going to be in, you can’t really identify the minerals very easily cause they’re fine grained and the minerals could be only a twentieth of a millimetre across in these
That’s going to be difficult to tell in hand specimen cause you can’t identify the minerals so well.

2

2Taken from Structured Interview I - Sisyphus KA Material
Classification of minerals is a science in itself and many of the features described in the interviews and books regarding mineral classification, apply to fully developed (crystallized) minerals. The problem with identifying minerals in igneous rocks, particularly when they are small-grained, is their lack of crystallization. To overcome this difficulty minerals can be identified in thin slices of rock under the microscope.

Therefore the classification of igneous rocks involves nesting methods of classification. Instead of simply obtaining a value for the selected attribute it could be decided that the value itself is subject to classification. This is likely to occur when the attribute is not directly observable and can only be inferred from other observables at the microscopic level.

We illustrate this process with an example:

Identification of the mineral quartz in igneous rocks with a coarse-grained texture is not very difficult. Quartz is common in many rocks, often abundant in lightly coloured ones. It is often colourless or grey and it does not show clear-cut cleavages: it splinters when broken.

Such information doesn’t constitute a classification in itself. It serves more as an explanation of the attribute quartz-presence. In fine-grained rocks the process is far more complicated. Individual grains cannot be seen, even with the help of a small hand lens. It’s here that an investigation of the thin section is necessary. Of course one could also resort to the investigation of thin sections in case of a rock with a coarse-grained texture.

![Diagram](image)

**Figure 2.7**

Nested classification. In order to establish a rock type it may be necessary to provide information about minerals present in the rock. A recursive classification process can be employed to classify a mineral in the rock.

Figure 2.7 depicts the nesting of classification in a graphical way. Instead of obtaining a value for an attribute, one recursively starts another classification process. The attributes that will trigger the recursive call are typically the presence of certain minerals. In order to identify these, the classification procedure will start from the beginning. A set of possible candidates will be
generated (these will all be minerals now), an attribute is specified, and so forth, until the mineral class is established.

2.6 Demonstrator

A demo-program has been built which can assist in classifying igneous rocks. The program makes use of the pruning-strategy described earlier. It looks up all the rocks known to the system and prunes away those candidates which are inconsistent with the attribute information provided by the user. If there is only one candidate left a picture of the rock is shown. Figure 2.8 shows a snapshot of the interface of the demonstrator.

![Figure 2.8 Opening window of the Rocky III demo.](image)

At the top a number of text items are shown that allow selecting values for the main rock attributes such as colour, texture and grain size. The left list-browser will show the list of candidates consistent with the current data. The list-browser on the right will give the user a hint about the attribute to fill in next. This is the result of the attribute ordering inference described in the previous section. This ordering is dependent on the problem-solving method. The default-value for this option is “tree” (most common attribute for remaining candidates). The other options are “info” (using the most informative) and “prune” (random selection).

Pressing the Help button launches Netscape with a local HTML-file. Here an explanation of the descriptors and classification method is given. As long as no solution has been found, Sisyphus himself is shown in full labour.

According to the International Union of Geological Sciences (UIGS) igneous rocks are classified based on the (relative) percentages of certain minerals. The user can submit this information to the system by making use of sliders (see Figure 2.9).
Initially the values of all sliders are set to -1, which means that the values are unknown. Between some sliders constraints are defined, which make it impossible for the user to enter incorrect or inconsistent data. For example, setting the slider for quartz to a non-zero value will result in a foid percentage of zero, because the two minerals are never found together in a rock. The reverse also holds: presence of foids excludes the presence of quartz.

Within the system every rock is represented as a class together with certain attributes and their possible values. In case of the mineral percentages a lower and upper bound is stated for each rock.

In case of medium or fine grained rocks it is often very difficult to determine the minerals contained in the rock. For this purpose the minerals themselves are subject of classification in a thin section of the rock. To start the program to classify minerals in thin section the user simply pushes the Thin Section button in the main program.

The window for Thin Sections shows a number of attributes with each a restricted number of values. In order to explain each of the attributes the user may click on the labels. This will result in the execution of Netscape which then shows a page explaining how to obtain a value for the attribute.

The classification of minerals is similar to the case of igneous rocks. Inconsistent candidates are pruned away. After the nature of the minerals has been determined the user may close the Thin Section program and continue with the classification of rocks.

New attributes, minerals and rocks are quite easy to add to the program. All are kept in separate modules in easy to understand code. Minerals are, like rocks, represented as classes with lists of
attribute-value pairs. Attributes are represented as a list with the range of values. Minerals and rocks can be added by editing the appropriate files and inserting code. Adding attributes works similarly. No changes to the graphical interfaces will have to be implemented. The interface is built up dynamically and changes automatically as the attributes change.

At this moment the program does not make use of the ontologies written in CML. The representation of rocks and minerals is however very similar to the ones used in the rock- and mineral ontologies and future versions are likely to be extended by making use of a CML-API. The program also does not make use of the hierarchy of igneous rocks and minerals. It uses a flat list of classes instead. The information in the texture ontology is incorporated as attribute information and therefore less explicitly present than in the ontology. The classification ontology is not used at all in this demo.

The system was written in SWI-Prolog and XPCE, and consequently runs on both UNIX and PC (Windows) systems.3

2.7 Discussion

The first round of the Sysiphus III project certainly was a useful and interesting exercise. The domain of rock classification shares many characteristics with other real world domains: it is rich, much terminology is standardized but individual experts use idiosyncratic terms, the procedures for classification depend heavily on the type of information available and on the viewpoint and expertise of the person who performs the classification (amateur, astronomer, petrologist). As a consequence, much effort in the early knowledge-acquisition phases concerned familiarization with the domain, scoping of the knowledge types, studying and evaluating the source material and obtaining additional information. One lesson that we learned in this KA process is that in well-understood and well-documented domains a good textbook is preferrable above interviews with

3See http://www.swi.psy.uva.nl/software.html for details on SWI-Prolog and XPCE.
experts or other results of elicitation activities.

The strategic approach that we have taken focused on ontology construction. Problem-solving methods for classification tasks are widely available and quite well understood, although their formal properties are not always clear. Thus, PSM’s have not been a major topic of our efforts so far. We have reused classification PSM’s that were developed for other purposes [97]. The main challenge of the Sysiphone III task for our group was to find a rich and coherently partitioned representational framework of the domain knowledge.

The knowledge of rock classification was partitioned along several dimensions. A first dimension concerns the task dependency. The classification ontology that describes the structure of objects, classes and their descriptions is independent of the domain of igneous rocks, but specific for classification tasks. In fact a very similar ontological structure is used in the domain of classification of art and antiques object in the GRASP project.

The second dimension is domain dependent and concerns distinctions of materials (rocks and minerals) and their various attributes. Domains that involve perceptual attributes as well as semi-quantitative attributes require separate ontologies for the attributes and their potential values. Much of the knowledge involved in domains like rock classification and art and antiques concerns the determination of the relevant properties of the object at hand. A clear separation is required between properties in the various domains: phenomenological properties (colour, grainsize, texture), thin-section properties, physical properties (hardness, specific weight) and chemical properties.

The coupling of the various ontologies in the demo application requires a mapping. This mapping can be viewed as an ontology in itself. Mapping ontologies are a powerful tool to reuse existing knowledge bases and ontologies, but are poorly understood so far. More work has to be done to understand the mapping knowledge at a generic level.

In summary, we have focussed on the groundwork for the future phases of the Sisyphus-III project rather than on an application for Round-I. However, a demo system could be built in a short period of time using parts of the ontologies and existing implementations of Problem Solving Methods for classification.

Acknowledgements We are grateful to Nigel Shadbolt and his colleagues at the University of Nottingham for the work they put in setting up the Sisyphus-III experiment. We also acknowledge the contributions of Arno Stam and Mark Verkerk in discussions we had on the Rocky domain.
3.1 Introduction

Knowledge-engineering research has delivered an abundance of problem-solving methods (PSMs) for classes of tasks such as classification, diagnosis, and configuration (see e.g. Benjamins, [9]). These PSMs are used in practical knowledge systems. To get a better understanding of the different PSMs work has been done on the formalization of PSMs, [27, 93, 97]. In this chapter we describe another approach, in which we adapt semantic tableaux to formalize PSMs. We show that this provides us with a technique for studying the logical properties of PSMs. In particular, we show that this approach allows us to model non-deductive style reasoning. This was a problem with previous formalizations based on first-order logic. We use the classification task as an example.

3.2 Characterizing Classification

We can characterize a knowledge-intensive task by defining three aspects: (1) the goal of the task, (2) the ontological commitments, and (3) the solution criteria.

Goal

The goal is typically an informal description of what the task attempts to achieve. In the case of classification, the goal is to identify to which class a certain object belongs. Example classification tasks are apple classification, rock classification, and art-object classification.

Ontological commitments

Ontological commitments describe our assumptions on the representation of the task domain. They provide us with a vocabulary. We can use this vocabulary to define what we mean with
classification. Together, the ontological commitments form an ontology for a task. An example of such a task ontology is the configuration-design ontology [38]. For classification we base the task ontology on the descriptions of classification given by Stefiak [87] and Schreiber [82].

We define six basic ontological types, namely attribute, object, value, class, feature, and observation. A is a (finite) set of attributes, each of which is associated with a list of possible values. A feature is an admissible attribute-value pair. Objects that need to be classified are described by a finite number of attribute-value (AV) pairs. These AV-pairs are called observations. The set of observations for a particular object is called Obs.

By definition, we assume that an attribute can have only one value at the time. So, if colour is an attribute and \{red, yellow, blue\} is its list of possible values, then an object description can never contain the AV-pairs colour = red and colour = yellow simultaneously. Every attribute with more than two values can be transformed into several attributes which all have binary values. This transformation is performed as follows: For every such AV-pair we take a new attribute with possible values true and false. The new attribute has the value true if the original AV-pair holds, and false otherwise.

Note that after having applied such a transformation the exclusion of multiple values for an attribute is no longer guaranteed. In order to maintain this principle every new attribute has the original attribute as its type. Now we can say that exclusion of multiple values holds for binary AV-pairs of the same type. If one binary attribute has the value true, then all other attributes of the same type have the value false. In this way, each multi-valued attribute can be represented as a set of atomic propositions.

Classes can now be represented as follows:

\[ c \rightarrow (a_1 \lor \ldots \lor a_n) \land \ldots \land (b_1 \lor \ldots \lor b_n) \]

The class name is represented by the proposition \( c \) and implies its features (i.e. AV-pairs). Features are here represented as atomic propositions with an index for ease of representation. \( a_1 \) represents the feature where \( a \) designates the attribute and the index \( i \) a certain value.

Domain knowledge can be represented by assigning meaningful names to the atomic propositions occurring in such a sentence. For example:

\[
\text{blackbird} \rightarrow \\
(plumage = \text{black} \lor plumage = \text{brown}) \land (\text{bill} = \text{yellow})
\]

The domain theory \( DT \) consists of a conjunction of class definitions.

Solution criteria

There are several alternative criteria one can formulate with respect to the goal of the classification process. We define two criteria:

**Weak classification** In weak classification (WC) a candidate solution must be a class \( c \) which is consistent with the domain theory \( DT \) and the observations \( Obs \) made thus far. Formally, this set of candidate solutions \( S \) can be expressed by:

\[ S = \{ c \mid DT \cup \{ c \} \cup Obs \not\models \perp \} \]

1 In Sec. 3.4.3 we introduce a third form of classification, namely composite-solution classification.
**Strong classification** In strong classification (SC) a class $c$ is a member of the set of candidate solutions $S$ iff the domain theory together with $c$ explains all observations. That is, we want candidate solutions to be classes which actually possess the properties that have been observed. Formally, the criterion for SC is:

$$S = \{c | DT \cup \{c\} \models Obs\}$$

The criterion of strong classification is stronger in a logical sense than weak classification. If a class is a candidate solution according to SC it is also according to WC. This follows as SC can be formulated as an extension of WC.

### 3.3 Tableaux

Semantic tableaux were developed in the 50's by Beth [10] and Hintikka [40]. Like resolution they form a *refutation* system. In a tableau proof a tree is constructed where nodes are labeled with formulæ.²

In order to test whether a certain formula $\varphi$ follows from a set of premises $\Theta$ a tableau tree is constructed for $\Theta \cup \{\neg \varphi\}$. Constructing such a binary tree can be seen as checking for (in)consistency of the theory. It is built using reduction rules which determine how the tree is branched. If in any branch, a formula and its negation appear the branch is said to be closed. If all branches close the theory is inconsistent.

$$\frac{\neg \neg Z}{Z} \quad \frac{\alpha}{\beta} \quad \frac{\beta^1}{\beta^2}$$

**Table 3.1**

Rules for the tableau trees.

Table 3.1 shows the rules for constructing the tableau tree. The first rule indicates that double negations are redundant. All propositional formulas containing binary connectives can be divided as belonging to two types: True conjunctive formulas ($\alpha$-type) and true disjunctive formulas ($\beta$-type). For example $p \rightarrow q$ can be rewritten to $\neg p \lor q$ and so is a $\beta$-formula. The rule for $\alpha$-type formulas indicates that the conjuncts have to be placed on the same branch of the tree. The $\beta$-rule however indicates a branching of the tree.

To see how this works we give an example. Let $\Theta$ be $\{p \rightarrow q\}$ and $\varphi = q$. We will check whether $\Theta \models \varphi$. The resulting tableau is depicted in Fig. 3.1. Since the tableau is open $\varphi$ does not follow from $\Theta$. Note that the open branch corresponds to a counterexample for $\Theta \models \varphi$.

### 3.4 Tableaux for classification

The general method to use propositional semantic tableaux as a proof procedure for classification proceeds as follows. The solution criterion for the classification task is translated into a consistency formula. Subsequently that formula is expanded into a semantic tableau. If the tableau

²We limit ourselves here to the description of propositional tableaux since this seems sufficient for the description of classification. We follow Fitting [30] in this description.
can be closed or remains partially open, a conclusion can be derived about the solutions of the classification problem. The precise nature of the conclusion depends on the nature of the solution criterion.

The closing rule for tableaux for classification can be specialized on the basis of the ontological commitments described in Sec. 3.2. Since all features that occur in the domain theory are considered to be typed propositions and since two distinct propositions of the same type exclude each other, any branch that contains two different propositions of the same type can be closed. This is equivalent to the addition of formulae of the following type to the domain theory for each feature:

\[ a_1 \rightarrow (\neg a_2 \land \ldots \land \neg a_n) \]

Building this ontological commitment into the proof procedure retains all properties of the general procedure such as soundness and completeness, but is more efficient.

3.4.1 Weak classification

In weak classification we assume that the domain theory and the observations together are consistent. This assumption is implicit in the way knowledge is represented and can be viewed as an additional ontological commitment.

Because of this consistency a semantic tableau for $DT \cup Obs$ will have open branches. Fig. 3.2 shows a tableau for the domain theory \{c_1 \rightarrow a_1 \land d_2, c_2 \rightarrow a_2 \land d_3\} and the observation $d_3$. The observation is added to the leaves of the tree for the domain theory.

Now, in order to check which classes are consistent with the observations made up to this point, each class must be individually added to the tableau. If the tree closes the resulting theory is inconsistent and the class is not a candidate. If a new observation is made it must be added to the tree and a check for all remaining candidates has to be made again. Fig. 3.2 shows class $c_1$ to be inconsistent with the domain theory together with the observation $d_3$, as its addition to the tableau would close the tree.

The trouble with this representation is that it leads to a rapidly expanding tree. Notice that since class definitions are $\beta$—formulae each class definition leads to branching of the tree. In classification, observations are typically added incrementally. This means that classes that at some point are proven to be inconsistent will remain part of the tree. This can be be remedied by putting
Adapting tableaux for classification

Figure 3.2
Initial tableau for weak classification.

the observations at the root, instead of adding them to the leaves of the tableau for the domain theory. The result is a much smaller tree. Subsequent observations can also be added to the root instead of the leaves. This proof strategy is depicted in Fig. 3.3. The example shows a general property of tableau proofs: the length of a proof typically depends on the order of application of the expansion rules.

Figure 3.3
Alternative tableau for weak classification.

This procedure for weak classification with semantic tableaux can be summarized as follows:

Procedure WC-1:
1. Construct a tableau for the observations and the domain theory (in this order)
2. FOR each possible candidate class c DO
   IF c (and c alone) is added to the tableau
AND the tableau closes
THEN c is not a possible candidate
ELSE c remains a possible candidate

3. When new observations are made:
   a. add observation(s) to top of the tableau
   b. redo step 2

It is interesting to view weak classification in terms of what is actually deduced during the process. In order to prove that a formula follows from a theory with the help of semantic tableaux one has to prove the inconsistency of the negation of the formula together with the theory. In the case of weak classification the aim is to prove the inconsistency of a class together with the domain theory and observations. In terms of consequence this means proving the negation of a class from the domain theory and observations. More formally:

\[ DT \cup Obs \cup \{c\} \vdash \bot \iff DT \cup Obs \vdash \neg c \]

In other words: weak classification only provides negative information about classes. The method is actually more about ruling out candidates than looking for candidates which can explain the observations. This is left to the stronger criterion of strong classification.

The procedure WC-1 is still inefficient since it generates the tableau for the entire domain theory. The ontological commitments about the structure of the knowledge base, i.e. that class definitions are conjunctions of disjunctive feature sets, allow us to specialize the procedure even more. A branch in the tableau tree that contains no literals involving classes, can never be closed by adding other parts of the domain theory that do not concern the class under investigation. This leads to the following procedure:

Procedure WC-2:
1. Construct a tableau for the observations
2. FOR each possible candidate class c DO
   2a. add the formulae that concern c from DT to the tableaux
   2b. IF c (and c alone) is added to tableau AND the tableau closes
       THEN c is not a possible candidate
       ELSE c remains a possible candidate
3. When new observations are made:
   a. add observation(s) to top of the tableau
   b. redo step 2 for those classes that were possible candidates

This procedure is much more efficient than WC-1, but given the ontological commitments still sound and complete. In fact, this proof procedure explains why the "test" part in implemented problem solving methods can remain so simple. It is also clear from this procedure that generate-and-test methods that do not test for all classes are sound, but not complete. In step 3b we make use of the fact that addition of new observations will not make classes that have been ruled out on the basis of earlier observations, candidates again.
3.4.2 Strong classification

We now proceed to describe a procedure for testing whether a candidate class fulfills the SC requirement. If it can be shown for a certain class $c$ that $DT\cup\{c\}\cup\{\neg o_1 \lor \ldots \lor \neg o_n\}$ is inconsistent, the criterion of SC is met with respect to $c$. In order to show this, first a tableau for $DT$ and the disjunction of negated observations is built. Since this theory is consistent the tree will have open branches. If $c$ is added to the tree and closes the tableau, $c$ is a SC candidate, otherwise it is not. Here we make use of the ontological assumption that $DT\cup\{c\}$ is consistent. An example is given in Fig. 3.4.

![Tableau for strong classification](image)

**Figure 3.4**
Tableau for strong classification.

This procedure for SC can be summarized as follows:

Procedure SC-1:
1. Construct a tableau for the domain theory
2. Add the disjunction of the negation of each element of Obs to the leaves of the tableau for the domain theory
3. FOR each possible candidate class $c$ DO
   IF $c$ (and $c$ alone) is added to the tableau AND the tableau closes THEN $c$ remains a possible candidate ELSE $c$ is not a possible candidate
4. IF new observations become available THEN redo step 2 and 3

SC is in general more complex to compute than WC. To infer the observations from the domain theory and a candidate class, the disjunction of all negated observations should be added to the tableau. In contrast to WC, each observation will give rise to a branching of the tree.

The same specializations of the proof procedure that we have described for WC apply to SC.
3.4.3 Classification and abduction

The procedure SC-1 is identical to the way abduction is performed with tableaux as described by Aliseda-Llera [3]. This is intentional. As defined above, comes down to an abductive method. Abduction is often linked to a style of reasoning which produces causal explanations for observations. Classification is rarely, if ever, considered to be an abductive task. Still the criterion of SC is formally in line with the definition of abduction, as for example put forward by Kakas et al. [50].

Aliseda-Llera [3] defines five different styles of abductive reasoning. Given \( \Theta \) (a theory) and \( \varphi \) (a sentence), \( \alpha \) is an abductive explanation (abducible) if:

- **Plain**: \( \Theta, \alpha \vdash \varphi \)
- **Consistent**: \( \Theta, \alpha \models \varphi \) and \( \Theta, \alpha \) is consistent.
- **Explanatory**: \( \Theta, \alpha \vdash \varphi \) and \( \Theta \nvdash \varphi \) and \( \alpha \nvdash \varphi \)
- **Minimal**: \( \Theta, \alpha \models \varphi \) and \( \alpha \) is the weakest such explanation.
- **Preferential**: \( \Theta, \alpha \models \varphi \) and \( \alpha \) is the best explanation according to some given preferential ordering.

Interestingly enough, SC displays four of these properties of abduction. The "plain" property follows directly from the solution criterion. Classification is a restricted form of abduction: the only abducibles allowed are atomic class propositions. Since it is an ontological assumption that class definitions are individually consistent with the domain theory (i.e., \( DT \cup c \) is consistent), it follows that solutions found by the SC-1 method are consistent. SC is minimal since we restrict the form of abducibles to single classes (atomic propositions) only.

Weak classification does not exhibit any of the properties of abduction, since no formula (\( \varphi \)) is assumed to be entailed by the theory (\( \Theta \)) and the abducibles (\( \alpha \)). Intuitively, WC generates a formula and tests if it is consistent with the current domain theory and observables, but it does not try to explain anything. This makes WC a very different task from SC. This is in line with our earlier observation that WC is a ruling-out task rather than an explanation task. From a logical point of view, one could argue that the two forms of classification are rather different ways of reasoning, even though procedurally they are very similar. One could even go as far as considering WC not as a classification task, but as a refutation task.

For the property of minimality, there is a difference between abduction and strong classification. If we decide to allow not only single classes as abducibles but conjunctions of classes as well, we end up with a different style of classification. In that case we would get a form of classification in which more than one class explains the observations and counts as a solution. Thus, a conjunction of classes can act as a solution candidate. This is known as composite-solution classification (CSC) [87].

The solution predicate can be formulated as follows:

\[
S = \{ c_1 \land \ldots \land c_n | DT \cup \{ c_1 \land \ldots \land c_n \} \models Obs \}
\]

Note however that we can no longer assume the property of consistency (\( DT \cup \{ c_1 \land \ldots \land c_n \} \)), as the simultaneous addition of two or more classes (e.g., a black bird and a white bird) to the
tableau of the domain theory may lead to inconsistency. Therefore, the procedure will have to test explicitly for consistency. For example, if \( DT = \{ c_1 \rightarrow a_1, c_2 \rightarrow a_2 \} \), adding the composite solution \( \{ c_1 \land c_2 \} \) makes the theory inconsistent, since \( a_1 \) and \( a_2 \) are regarded as exclusive.

If one allows composite solutions, one could still prefer single solutions. In this case CSC is defined as preferential abduction in the above sense.

3.5 Discussion

Even though classification is one of the simplest knowledge-intensive tasks in the knowledge-engineering domain, it has been quite hard to prove that certain computational methods satisfy logical competence theories. Similarly, it has been difficult to transform logical competence requirements into an operational method. Problems encountered include: the abductive nature of classification reasoning, the incremental nature of observation gathering and the mapping of logical theories onto different computational strategies in classification methods [97]. In this chapter we have presented some steps forward towards solving these problems.

A first insight is that strong classification is a special case of abduction. The solution of a classification problem is considered as an abducible of some domain theory and the observed facts. Theories of abduction provide several types of abduction that can be mapped onto different variants of the classification task. However, classification is more specific than abduction in the sense that it restricts the vocabulary of abducibles to a predefined set of classes and that it assumes a particular structure of the domain theory.

Second, it appears that the semantic tableau proof method has some features that makes it suitable to model various forms of classification reasoning. Tableaux provide a natural way of handling incrementally growing theories as they often occur in knowledge-based systems, where new facts are incrementally obtained from a user. In classical logical approaches that attempt to formalize classification reasoning, this problem is not easily solved [97]. Tableaux also provide a way of thinking about the search space of possible inferences in a formal context. As we have shown, the ontological commitments of the task restrict the possible expansions and closures of the tableaux. These restrictions can be translated into the proof procedure itself, thus reducing the space of formulae to be processed. This is precisely where knowledge-based systems derive their power from, when compared to general theorem-proving approaches. It can be proven that the specialized proof procedures are equivalent to the normal proof procedure when the ontological commitments are added as axioms to the domain theory.

The third result of our investigation is that PSMs for classification that have been published (by Steifik [87] and Wielinga [97]) can be mapped onto proof procedures. For example, procedure WC-1 and WC-2 formalize the "pruning" method, which can therefore be characterized as both sound and complete. WC-1 is a "pure" logical method, but computationally not very efficient. In tableau terminology it generates a much larger tableau than WC-2. WC-2 is an optimized method which is in fact close to operational methods used in classification systems. Procedure SC formalizes a generate-and-test method for classification. Here, we can see from the formalization that if the method would terminate after having found one solution (which is often the case in operational methods), the method is sound, but not complete. In short, this type of mapping of PSMs used in knowledge-based systems onto specific tableau proof procedures provides a powerful way of establishing the competence of these methods in logical terms.
A Spectrum of Classification Methods

This chapter is submitted for publication.

We present a systematic survey of criteria for knowledge intensive classification problems. Criteria are build up from some simple definitions and combined into more complex ones. Orderings will allow the description of some basic preferential criteria. By relaxing some fundamental constraints other preferential criteria are obtained. It will be shown that all such criteria are interrelated and can be ordered by set-inclusion of their solution sets. Such a systematic overview may be helpful as an indexing scheme for Problem Solving methods (PSMs) for classification. It can facilitate finding a criterion on the basis of a goal description.

4.1 Introduction

The goal of a classification task is to identify an object as belonging to a certain class. The object is described in terms of a (possible incomplete) set of observations. Identifying a bird, flower, rock or an object of art on the basis of some (possible incomplete) set of observations are all examples of classification problems.

An overview of several classification criteria and associated methods is given by Stefik [87]. By a classification criteria we mean a predicate which is true of a class when the class is a solution to the problem at hand, and false otherwise. The choice of a Problem Solving Method (PSM) depends on the criteria one chooses, and these may differ between different instances of classification problems. With each criterion one can associate a set of classes which are regarded as solutions. These can be ordered by set-inclusion. The resulting ordering makes it possible to compare criteria and the functionality of their associated PSMs.

Recently, attempts have been made to capture PSMs in libraries and facilitate their automatic retrieval on the basis of a problem specification [9] [65]. As a test-case classification was chosen as a task to be made operational. We feel that a systematic description of criteria which we offer here, may add to the construction of such libraries.

In this chapter we present a systematic overview of possible criteria for classification problems. We do this by describing simple criteria from basic definitions and use these in the construction of more complex ones.

A similar overview of criteria for diagnostic criteria was given by Console and Tarasso [21], and TenTeije and van Harmelen [89]. At least intuitively there seems to be a relation, and even a
possible overlap between diagnosis and some forms of classification. However, classification tasks are often regarded as much easier to define, and making use of much less sophisticated methods of problem solving.

4.2 Classification in terms of attributes

In classification tasks classes as well as observations are typically described in terms of attribute-value pairs (av-pairs).

**Definition 4.1**: Let $A$ be a finite set\(^1\) of attributes. Each attribute $a \in A$ has a set $V_a$ (a domain) of possible values associated to it. An **attribute-value pair** is an ordered pair $(a, v)$ where $a \in A$ is an attribute and $v$ a value from its associated domain $V_a$.

Observations form a set of av-pairs and they represent an object that should be classified. Each av-pair in the set of observations will be referred to as an observation. Classes are here also represented as non-empty, finite sets of av-pairs, where the name of a set represents the name of the class. An alternative definition, used in the previous chapter and encountered in the literature (see e.g. Wielinga [97]) is that a class-name implies the conjunction of av-pairs. The relation between such a representation and the one used here is quite straightforward.

Within each class and within the set of observations each attribute can have only a single value. When this is the case then the class, or the observations are said to be **internally consistent**. In the following internal consistency is assumed for all classes and the set of observations.

Not all attributes have to occur as part of some av-pair in a class or the observations. If an attribute does not occur in a class or the observations it is called **missing** (with respect to the class or observations).

Instead of allowing attributes to be missing, an alternative representation [87] is to give the value **unknown** to an attribute. In this case all attributes have to be mentioned in all class definitions and the observations.

In order to check whether the object, which is described by the observations falls under a particular class, the class must be compared to the observations. It is assumed here that in both the descriptions of observations and classes the same attributes and values are being used. However, this assumption may easily be given up. A **match** between two attributes can be defined as:

**Definition 4.2**: Two attributes (one from a class and one from the set of observations) **match** if there is a predefined mapping between the two values.

Matching two av-pairs can be as simple as a test for syntactical identity. Class and observation attributes may be distinct and a mapping might be needed to compare their values. Another option is the use of an interval as a value for an attribute. Hence, there is a certain degree of freedom in the implementation of matching.

The important notion of **consistency** between an av-pair from a class and one from the observations can be defined as follows:

**Definition 4.3**: An attribute $a_c$ of a class $c$ is **consistent with the observation** $o$ iff it does not occur in $o$ or it matches an attribute in $o$. This relation is symmetric in the following sense: an

---

\(^1\)In what follows all sets are finite.
attribute $a_o$ of an observation is consistent with a class $c$ iff it does not occur in $c$ or it matches an attribute in $c$.

One could also allow disjunctions of attribute-value pairs. For example one could state that the value for the attribute colour for the class blackbird is either brown or black. In a representation in terms of sets of av-pairs this would mean that some attributes will have multiple values, which should be interpreted as disjunctions. This means that the notion of internal consistency of a set of attribute value-pairs can no longer be computed.

Note that the notion of consistency between class and observation attributes is in line with the use of disjunctive attributes.

### 4.2.1 The attribute level

Given a set of observations (as a set of av-pairs) we must determine which class can be regarded as a solution. In order to formulate a criterion we explore what possibilities arise when an av-pair belonging to a class is compared to one belonging to the observations. We limit our focus to the attribute level and focus on attributes without considering their values. There are four basic options:

1. The attribute occurs in both the class and the observation.
2. The attribute does not occur in the class but does in the observation.
3. The attribute occurs in the class but not in the observation.
4. The attribute occurs in neither the class nor the observation.

Next, we consider each of the above options and take the values of the attributes into consideration when needed. In the first case, when an attribute occurs in both the class and observations, there are two possibilities:

1. The two values of the attribute do not match. In that case the two av-pairs are inconsistent. (For short we say that the attribute itself (with respect to the class and observations) is inconsistent.)
2. The two values of the attribute match. Then the av-pair in the observation is said to be explained by the matching av-pair in the class. Similarly, the av-pair in the class is said to explain the matching av-pair in the observation.

In the second case the attribute is present in the observation but not in the class. In this case there is no inconsistency, but there is no matching and the attribute in the observation is said to be unexplained.

In the third case, the attribute present in the class is said to be an absent explanation of the absent attribute in the observation. Figure 4.1 illustrates these terms.

The fourth case represents an empty statement. The attribute is neither observed nor mentioned in the class-definition.
4.2.2 Universal criteria for classification

These options on the attribute level can be generalized to class-level by universally quantifying over av-pairs. Referring to this quantification we call the resulting criteria universal criteria for classification. The well-known criteria for weak and strong classification are among them [87].

**Definition 4.4**: *Weak classification*: A class $c$ is a weak solution for an observation $o$ iff all the av-pairs in $o$ are consistent with all the av-pairs in $c$.

Note that this definition merely involves a generalization of the attribute level criterion of consistency to class-level.

**Definition 4.5**: *Strong classification*: A class $c$ is a strong solution for an observation $o$ iff all the av-pairs in $o$ are explained by all the av-pairs in $c$.

Weak and strong classification are perhaps the best known criteria. However, they are not the only ones. Following up on the notions described earlier, another criterion would be:

**Definition 4.6**: *Explanative classification*: A class $c$ is an explanatory solution for an observation $o$ iff all the av-pairs in $c$ are explanations for all the av-pairs in $o$.

In addition, the last two criteria can be combined by the use of conjunction:

**Definition 4.7**: *Strong explanatory classification*: A class $c$ is a strong explanatory solution for an observation $o$ iff all the av-pairs in $o$ are explained by all the av-pairs in $c$, and all the av-pairs in $c$ are explanations of an av-pair in $o$.

Likewise disjunction can be used to produce yet another criterion:

**Definition 4.8**: *Covered classification*: A class $c$ is a covered solution for an observation $o$ iff all the av-pairs in $o$ are explained by all the av-pairs in $c$, or all the av-pairs in $c$ are explanations of an av-pair in $o$.

This last criterion does not add anything extra as far as individual classes are concerned, since a class which is a covered solution is either a strong or an explanatory solution. However the criterion is relevant when we associate with each criterion a set of solutions:

$\text{WEAK} = \{ c \mid c \text{ is a weak solution} \}$

$\text{STRONG} = \{ c \mid c \text{ is a strong solution} \}$

$\text{EXPL} = \{ c \mid c \text{ is an explanatory solution} \}$

\footnote{The name covered is chosen because either all attributes in the class or in the observation are explanations or explained respectively.}
These sets can be partially ordered by set-inclusion as is shown in figure 4.2.

\[
\text{COVERED} = \{ c \mid c \text{ is an explanatory or a strong solution} \}
\]
\[
\text{STR-EXPL} = \{ c \mid c \text{ is a strong explanatory solution} \}
\]

Figure 4.2
The ordering of classification criteria by set-inclusion. An upward line represents set inclusion: e.g. STR-EXPL $\subset$ STRONG.

From this partial order it follows that every strong explanatory solution is also a strong solution. Every strong solution is also a weak solution. And both explanatory and strong solutions are covered solutions. Every strong explanatory solution is also an explanatory solution, which in turn is also a weak solution.

4.2.3 An analysis in terms of attribute sets

The ordering of classification criteria can be further analyzed by looking at the attributes occurring in class and observations. Again, we do not consider the values of the attributes.

**Definition 4.9:** Let $c$ be a class and $A_c$ be the set containing all attributes having a value in $c$: $A_c = \{ a \mid (a, v) \in c \}$. Similarly let $A_o$ be the set containing all attributes having a value in the observation $o$: $A_o = \{ a \mid (a, v) \in o \}$. When these two sets are compared the following cases may occur:

1. $A_c = A_o$
2. $A_o \subset A_c$
3. $A_c \subset A_o$
4. $A_o \cap A_c = \emptyset$
5. $A_o \cap A_c \neq \emptyset$ (This subsumes 1-3).

These possible relations between the attributes in the observation and class can be used to get some insight into the nature of the above mentioned criteria.
Theorem 4.1: Suppose \( A_c = A_o \) for some class \( c \) and some set of observations \( o \). Then the five criteria coincide for \( c \): If, and only if \( c \) is a solution according to one of the criteria, it is a solution according to all others.

Example 4.1: The following (simplified) class definition was taken from a system built to classify igneous rocks [46].

\[
\text{granite} = \{(\text{Grain size, Coarse grained}), (\text{Colour,Light}), (\text{Quartz, 20}), (\text{Quartz, 80})\}.
\]

The attribute \( \text{Quartz} \) is an interval attribute. Any observation of \( \text{Quartz} \) with a value within the interval \( 20 - 80 \) matches it.

The set \( A_{\text{granite}} \) contains all attributes occurring in the class \( \text{granite} \): \( A_{\text{granite}} = \{\text{Grain size, Colour, Quartz}\} \). If the set of observations defines values for all (and no more) of these attributes, then \( \text{granite} \) is either a solution according to all the criteria defined thus far, or to none at all.

Theorem 4.2: If for a class \( c \) \( A_0 \subseteq A_c \) holds, then the notions of weak and strong classification coincide for \( c \). In addition, \( c \) cannot be an explanatory solution and hence not a strong explanatory solution either.

Theorem 4.3: Suppose that for a class \( c \) \( A_c \subseteq A_o \) holds, then the notions of explanatory and weak classification coincide for \( c \). Also, \( c \) cannot be a strong solution and hence not a strong explanatory solution either.

Theorem 4.4: Suppose \( A_o \cap A_r = \emptyset \) holds for class \( c \). This means that the class \( c \) and the observations have no attributes in common. \( c \) is a weak solution with respect to \( o \) but evidently not a covered, strong, explanatory or strong explanatory solution.

Upon closer inspection we can define the aforementioned criteria, shown in figure 4.2, in terms of the attribute sets \( A_o \) and \( A_r \). By doing this it becomes clear why the above theorems are true.

Consider a class \( c \) which is regarded as a solution according to weak classification. When \( A_o \cap A_r \neq \emptyset \) holds, then all attributes which the class and observations have in common should match, otherwise an inconsistency occurs. Attributes which are missing in either \( A_o \) or \( A_r \) are allowed; they do not disturb consistency. Even if the class and the observations have no attributes in common \( (A_o \cap A_r = \emptyset) \) is \( c \) a consistent solution with respect to \( o \).

Formally we can express this as:

\[
\text{\( c \) is a weak solution } \Leftrightarrow \forall a : (a \in A_o \cap A_r \wedge match(a)) \lor (a \notin A_o \cap A_r)
\]

The other criteria can be defined by a set of attributes as follows: We define a set of attributes whose elements are required to match. This set is constructed by taking the intersection of the set of attributes \( A_c \) occurring in the class \( c \) which is regarded as a solution, and the set of attributes occurring in the observations \( A_o \).

1. Strong classification requires that a class \( c \) is regarded as a solution when \( A_o \subseteq A_c \) holds, and all elements of the intersection \( A_c \cap A_o = A_o \) match.

2. Explanatory classification requires that \( A_c \subseteq A_o \) holds and all elements of the intersection \( A_c \cap A_o = A_c \) match.
3. Strong-explanative classification requires that \( A_0 = A_c \) holds and that all elements in the intersection (either \( A_o \) or \( A_c \)) match.

4. Covered classification requires that either \( A_0 \subseteq A_c \) or \( A_c \subseteq A_o \) holds and that all attributes in the subsumed set match.

In between covered and weak classification another criterion can be defined. To the definition of weak classification we can add the constraint that \( A_o \cap A_c \neq \emptyset \) holds. We call this criterion **fortified classification**. It excludes the possibility that a class is solution if it has no attributes in common with the observations. This possibility can be seen as an (albeit absurd) criterion on its own and we refer to as NIL. It states that a class \( c \) is a solution if \( A_c \cap A_o = \emptyset \), and it’s use is obviously futile.

The partial ordering of universal criteria can be extended by these two criteria in the way depicted in figure 4.3.

![Figure 4.3](image-url)

The extended ordering for universal classification criteria.

From this ordering the aforementioned theorems are easily validated. For example the first theorem that the criteria (except NIL) coincide when \( A_o = A_c \) can be easily proved as follows: suppose \( o \) is a set of observations and \( c \) a candidate solution. In addition assume that \( c \) and \( o \) have the same attributes, so \( A_o = A_c \). If all attributes match, then \( c \) is a strong-explanative solution. All other criteria (except NIL) subsume strong-explanative classification, so \( c \) is a solution according to these as well. If not all attributes match there is an inconsistency and \( c \) is not a solution according to any of the criteria. In a similar way the other theorems can be proven.
4.2.4 Preferential criteria

The ordering by set-inclusion of universal classification criteria can also be seen as a preferential structure for possible solutions. Instead of selecting a criterion for a particular classification problem, one could determine for each class the strongest criterion which it satisfies. Some classes may be only weak solutions, others strong (and hence weak) but not strong-explanatory etc. All classes will then be partially ordered along the several criteria and the most preferred can then be selected. The result is a universal preferential criterion for classification.

Note that the decision of what to consider as the most preferred class can still be a matter of choice. Whether to prefer strong to explanatory solutions, or vice versa is not determined by the ordering.

Such preferential criteria for classification can be extended further. One could allow some attributes to be inconsistent and prefer the class(es) with the least inconsistencies as a solution. Or one could count the number of explained attributes and give a preference to the class which has the most. Such criteria are not universal: they are existential, in the sense that they can be described with the help of an existential quantifier ranging over the attributes.

4.3 Relaxing constraints

The universal criteria defined above impose constraints on all attributes which occur in class and observations. There are situations in which one would like to relax these constraints. This can be done in two ways: One could relax the constraint concerning the presence of attributes, and one could allow (some) inconsistencies. We first discuss the latter option.

4.3.1 Allowing inconsistencies

Consider a case where all classes fail the constraint that all their attributes match the observations. In this case one could still prefer the class with the least inconsistencies.

Allowing inconsistencies seems particular useful in preferential criteria: the class with the least inconsistencies is to be preferred. By relaxing the consistency constraint for each universal criterion a preferential structure emerges. To illustrate this construction we show this for strong-explanatory classification.

Strong-explanatory (SE) classification is the strongest criteria in the sense that it puts the most constraints on the set of attributes that should match. Notice that if a class is a solution according to the strong-explanatory criterion there are no absent explanations or unexplained attributes and hence \( A_n = A_c \) holds for any solution class \( c \) with respect to the observations \( o \). All attributes in either of these sets are required to match, and hence no inconsistencies are allowed.

By gradually relaxing this last constraint we can construct a preferential structure for SE. Meanwhile we do not weaken the constraint that no absent explanations and unexplained attributes are allowed. Therefore we demand that \( A_o = A_c \) holds as before, and relax the constraint that all the elements from one of these sets should match.

Let \( A_o = \{a_1 \ldots a_n\} \) be the set of attributes whose elements have a definite value in a given observation set \( O = \{a_1 = v_1 \ldots a_n = v_n\} \). For a class \( c \) we define the set of attributes whose elements each match an av-pair in the observation set \( O \): \( M_c = \{a | matches(a) \text{ and } a \in A_o\} \).
We call such a set the matching-attribute set of class \( c \).
We assume that attributes occurring in the observations and the class are the same (their values may be different). Note that under this assumption a matching attribute set is always a subset of $A_o$. Therefore all possible matching-attribute sets (for all possible classes $c$) are given by the powerset of $A_o$. We can order all these sets by set-inclusion. The result is a lattice. An example lattice is shown in figure 4.4.

\[
\begin{array}{c}
\{a_1 \ldots a_n\} \\
\{a_1 \ldots a_{n-1}\} \\
\vdots \\
\{a_1\} \\
\{a_n\} \\
\emptyset
\end{array}
\]

Figure 4.4
The lattice of possible matching attributes sets. The figure shows all possible subsets of the set $\{a_1 \ldots a_n\}$ ordered by set-inclusion. An upward line indicates subset inclusion. Not all subsets are shown. Dashed lines and dots indicate the presence of sets not shown.

This lattice can be interpreted in two ways. First, as a preferential structure: for each class $c$, the set of matching attributes $M_c$ can be computed (given an observation) and its place in the lattice be determined. To decide which class contains the fewest inconsistent attributes, can now be decided by an answer to the question whose class's matching-attribute set is closest to the top of the lattice. At the top of the lattice is the set $A_o = \{a_1 \ldots a_n\}$. If a class has $A_o$ as it's matching-attribute set then there are no inconsistencies. On the other hand if it's attribute-matching set is $\emptyset$ then there are $n$ inconsistencies. All other possible matching attribute sets are in between these extremes.

Another view on the same structure is to see every node in the lattice as a demand that the mentioned attributes are required to match. For example, the node $\{a_1 \ldots a_k\}$ can be interpreted as follows: $c$ is a solution if (and only if) all the attributes in $\{a_1 \ldots a_k\}$ match. Then the lattice is no longer viewed as a preferential structure. Every node represents an absolute criterion and the lattice as a whole represents the space of possible criteria relative to strong-explanative classification.

### 4.3.2 Absolute criteria

The lattice of matching attributes under the assumption that the attribute-sets of class and observations are the same, can be seen in two ways. First, as a preference structure for strong-explanative classification with possible inconsistencies. Given an observation with attributes $A_o$ we can determine for each class $c$ with attributes $A_c$ and $A_c = A_o$ the subset of attributes $M_c$, whose elements
each match an attribute occurring in the observations. The place in the lattice of the matching-attribute set determines the preference order of the class.

Second, with each node of this lattice we can also associate a new absolute criterion for classification. Each set of attributes \( M_i \subseteq A_0 \) can be taken as a constraint saying that these attributes should match. For example the set at the top of the lattice contains all attributes by definition. Demanding that these attributes should match is equal to the criterion of strong-explanative classification.

The set \( M_c = \{ a_1, \ldots, a_{n-1} \} \) then corresponds to the criterion in which all the attributes except \( a_n \) should match. In this case the attribute \( a_n \) is given a special status: it is regarded as unnecessary.

Similar to the construction of the universal criteria, we can define a set of classes which are covered by these new criteria. (Remember we have assumed \( A_c = A_0 \)). With each node of the lattice of matching-attribute sets we associate a set of classes. Each class \( c \) is a member of the associated set if its matching-attribute set \( M_c \) is equal to the set mentioned at the node:

Let \( v(A_0) = \{ A_0, A_1, \ldots, A_n, \emptyset \} \) be the powerset of \( A_0 \).

\[
E_{0se} = STR-EXP = \{ c \mid M_c = A_c \}
\]

\[
E_{1se} = \{ c \mid M_c = A_1 \}
\]

\[
\ldots
\]

\[
E_{nse} = \{ c \mid M_c = A_n \}
\]

\[
E_{\emptyset se} = \{ c \mid M_c = \emptyset \}
\]

Hence with every node in the lattice in figure 4.4 we associate a set \( Ei_{se} \) which contains the classes whose matching attribute sets are equal to the set which forms the node. As said before each node can be interpreted as a criterion that the given list of attributes should match, and the associated set of classes contains classes which satisfy this criterion. Moving up the lattice (by following the lines) means moving to stronger criteria. The classes associated with each node also satisfy the criteria with lower, subsumed nodes.

These sets of classes, or criteria are closed under conjunction. For example, when one wants the set of classes who match attributes \( a_i, \ldots, a_j \) and \( a_k \ldots a_l \) then the answer can be found as follows. Identify the nodes corresponding to both sets and follow lines upward in the lattice to the first node at which they meet. The result is the union \( a_i, \ldots, a_j \cup a_k \ldots a_l \). This set is part of the lattice and so is the associated set of classes.

However, the disjunction of criteria is not yet defined. In fact the matching attribute lattice is defined on the attribute level, and when generalizing to sets of classes the disjunctive closure should be added. For example, consider the criterion where at most one inconsistent attribute is allowed. This can be defined by a disjunction of all the criteria which allow one particular element to be inconsistent. This criterion is not an element of the sets of classes defined in the above construction.

By adding all sets associated with the disjunctions of criteria, the disjunctive closure of all the criteria in the matching attribute lattice is obtained. This results in a structure which is closed under conjunction and disjunction. Again this forms a lattice of criteria whose elements are all variants of strong-explanative classification.
We refer to these criteria as existential criteria because they allow for some attributes to be inconsistent. Universal strong-explanative classification can now be extended by a lattice of existential strong-explanative criteria, of which the universal criterion is the bottom element.

### 4.3.3 Comparing existential criteria

The same construction which produced new existential criteria for strong-explanative classification can be used to relax the other universal classification criteria.

To summarize the construction procedure: for each universal criterion list all the possible sets of attributes which are required to match the observations. In fact, all these possible sets are given by the powerset of the attribute set defined by the universal criterion. Ordering all elements of this powerset by set-inclusion results in a lattice. The top element of this lattice will allow no inconsistencies (it’s the universal criterion itself), the bottom element will allow all elements to be inconsistent (the empty set), and all the other sets express that some particular elements are allowed to be inconsistent.

Next, associate with each node a set of classes in such a way that the matching attributes of each class are exactly those in the node. Of all these sets take the disjunctive closure. All these sets of classes are now closed under disjunction and conjunction and form another lattice. This lattice is the inverse of the one it is constructed from. The top element contains all classes that may have no consistent attributes, the bottom element is the universal criterion that one has chosen at the beginning of the construction process.

Hence, instead of strong-explanative classification we can relax the criterion of strong classification by assuming $A_o \subset A_c$ instead of $A_o = A_c$. The line of reasoning is completely similar to the above construction. The only difference between all the existential strong criteria, as opposed to the existential strong-explanative criteria is that the former allows some attributes in the class to be absent explanations, which is expressed by $A_o \subset A_c$. Under all strong classification criteria, whether we allow inconsistencies or not, the difference of the sets $A_c$ and $A_o$ (written as $A_c \sim A_o$) is the set of absent explanations. Note that there are no unexplained attributes under all strong classification criteria. Similarly, all explanatory criteria allow unexplained attributes, characterized by the set $A_o \sim A_c$. There are no absent explanations.

This means that if we allow inconsistencies, we can still order criteria on the basis of explanatory attributes and explanations. If we relax weak and fortified classification by allowing inconsistencies, similar lattices are obtained. The only difference between all the pairs of new criteria is that the fortified ones drop the possibility that $A_o = A_c$.

Hence, every universal criterion can be extended by a lattice of existential criteria of which it is the bottom element. Furthermore, every existential criterion which allows an attribute to be inconsistent is related to the other existential criteria which expresses the same fact, but differs under the assumption of the relation between $A_o$ and $A_c$. For example: Strong classification which allows attribute $a_1$ to be inconsistent subsumes strong-explanative classification which allows $a_1$ to be inconsistent. This existential strong-explanative criterion is still stronger than its strong counterpart in that it does not allow attributes failing to be explanations. All these inter-relationships lead to the ordering of the criteria by set-inclusion shown in figure 4.5.
4.4 Allowing missing attributes

Instead of relaxing the constraint that all attributes from a certain set should match by allowing some of them to be inconsistent, one could also allow certain attributes to be missing from either the class or set of observations.

For each universal criterion a similar construction procedure as was used for allowing inconsistencies can be used to allow missing attributes. Instead of listing the possible sets of allowed inconsistent attributes for each criterion, we list the sets of allowed missing attributes.

For example, for strong classification enumerating all possible sets of attributes which are allowed to be missing in the observation, can be done as follows. Again, let $A_o$ be the set of attributes in the observations: $A_o = \{a_1 \ldots a_n\}$. Let $\wp(A_o) = \{\emptyset, A_1, \ldots, A_m, A_o\}$ be the powerset of $A_o$. $A_c$ denotes the set of attributes occurring in the class $c$. For strong classification $A_o \subset A_c$ holds and hence $A_o \cap A_c = A_o$. We enumerate all subsets of the powerset of this set.

\begin{align*}
A_o \cap A_c &= A_o \\
A_o \sim \{a_1\} \cap A_c &= A_o \sim \{a_1\} \\
&\vdots \\
A_o \sim \{a_1 \ldots a_i\} \cap A_c &= A_o \sim \{a_1 \ldots a_i\} (0 \leq i \leq n) \\
&\vdots \\
A_o \sim A_o \cap A_c &= A_o \sim A_o \Leftrightarrow \emptyset \cap A_c = \emptyset
\end{align*}

The same can be done for explanatory criteria. Here we $A_c \subset A_o$ holds and hence $A_o \cap A_c = A_c$. Hence, by reading $A_o$ for $A_c$ and vice versa in the above enumeration, one obtains the possible sets of missing attributes for explanatory classification. For strong-explanative criteria a similar
construction $A_0 = A_c$ holds and the following enumeration can be performed:

$$A_0 \sim \emptyset = A_0$$
$$A_0 \sim \{a_1\}$$
$$\ldots$$
$$A_0 \sim \{a_1 \ldots a_n\} = \emptyset$$

Similar to the construction of inconsistent criteria, with each of these sets a set of classes which fulfill the associated constraints can be associated. Each set of classes then represents a criterion. If we take the disjunctive closure of these criteria, we end up with all possible criteria for missing attributes.

There are a few things to note here. Allowing all attributes to be missing, comes down to the universal criterion NIL, as defined in section 2. Allowing no attributes missing in either the class or observations is of course universal strong-explanative classification.

Allowing no unexplained attributes equals strong classification, demanding all attributes to be explanation equals covered classification. Expressing that at most two attributes may be unexplained, comes down to the disjunction of all the constraints which mention two particular attributes to be unexplained.

These missing-attribute criteria do form a lattice with as top the criterion NIL. This is not in line with the ordering of universal criteria. For example, we can relax the constraint of strong classification that each attribute in $A_o \cap A_c$ should match, further and further until $A_o \cap A_c = \emptyset$, which is the criterion NIL. However, in the definition of the universal criteria this option was explicitly forbidden, and NIL does therefore not subsume any universal criterion.

### 4.5 Combining inconsistencies and missing attributes

The last type of criteria we discuss are those that allow a combination of missing and inconsistent attributes. If the attributes that are allowed to be missing according to one criterion are disjoint from those of the one which allows them to be missing, the criteria can be combined by both conjunction and disjunction.

However, the situation is different for those criteria which share attributes. First, note that an attribute cannot be missing and inconsistent at the same time. We will show that inconsistency is the stronger notion of the two.

Intuitively, if a class is allowed to have one particular attribute inconsistent then it should also be allowed to have this attribute missing. Missing criteria seem to subsume inconsistent ones, defined for the same attributes.

One could argue one step further: if an attribute is missing in either the observations or the class, we do not know its value. This means that it could be inconsistent. If we allow a class with one particular attribute to be missing, we also allow the possibility that it is inconsistent. Therefore if a class $c$ with one particular attribute missing is regarded as a solution, so must a class $c'$ with the same inconsistent attribute. This view would entail that inconsistent and missing attributes amount to the same. We do not agree with this view and argue against it:
It may be true that if we allow a class with one particular attribute to be missing, we also allow the possibility that it is inconsistent. But we only disqualify a class as a solution if we are certain of an inconsistency. So if a class is allowed to have an attribute missing, we trust it will not be inconsistent.

Hence, missing criteria subsume inconsistent ones (for the same attributes), but not vice versa. This means that instead of saying that the set of classes which satisfy the constraint that an attribute is allowed to be inconsistent or missing, equals the set of classes which satisfy the constraint that it is allowed to be inconsistent, which is the stronger notion.

In summary, combining missing and inconsistent criteria for the same attributes can only be performed by disjunction and yields the inconsistent ones. If attributes of both criteria overlap, the ones who are allowed to be missing should be retained. The other attributes can be combined by disjunction and conjunction into new criteria.

4.6 Discussion

The previous treatment of classification criteria is not meant to be an exhaustive description of the task. The abstraction of features, the hierarchical ordering of classes and the refinement of solutions, have not been treated here. Also, we have not indicated how to solve any of the classification problems associated with a criterion.

We have shown how classification criteria can be described and be compared to each other in a systematic way. Starting with some simple criteria, one could build up more complex ones by simple constructions and weakening constraints. Any partial ordering of criteria can be viewed as a preference structure which in itself can be used as another classification criterion.

Describing a task like classification this way may facilitate the construction of libraries of problem solving methods for the task. Attempts are being made for the automated selection of a problem solving method after some initial goal specification by a user [7]. In order to select such a method one has to have some systematic account of how to describe the competence of the PSMs and their relationships.

Describing criteria for a task also allows for more complex ways of problem solving. Instead of identifying a criterion and try to establish a solution accordingly, one could go one step further. If some idea of what a solution may look like is available, one may search for the criterion which approaches this idea in an optimal way. This involves moving through the ordering of criteria, until the constraints for a solution set are optimally realized.

As an example, consider classification criteria which produce too many, or too few solutions. Strengthening, or weakening may lead to a more appropriate solution set. Other examples involve the search for criteria that should at least produce certain predefined classes as solutions, or should meet some predefined constraints. It is this kind of 'meta problem solving' which is also involved in the selection of a PSM on the basis of a goal description.

Another point for discussion involves the representation of the ontological elements like classes, attributes, values etc. We have chosen here for a set-based representation instead of one in (predicate)logic. Although one could easily translate one of these representations into the other, the logical one seems to be an interesting candidate for linking criteria to PSMs.

An an example we mention the criteria for classification with the least number of inconsistent attributes. At least intuitively there seems some correspondence between these criteria and the
'approximate reasoning' logic of Schaerf and Cadoli [79]. Ten Teije [88] uses this logic to describe diagnostic criteria.

This is an example of how one can associate with each criterion a consequence relation. Problem solving can then be described as performing deduction or abduction according to this consequence relation. The relation between criteria and different styles of logical reasoning will be a topic of future research.

4.7 Acknowledgement

This chapter originated after a discussion of the main ideas in [65] which can be found at http://kmi.open.ac.uk/projects/ibrow/documents.htm. We thank Enrico Motta for reviewing an earlier draft of this chapter. We also would like to thank F. van Harmelen en A. ten Teije from the Free Univeristy of Amsterdam for discussing the main ideas.
Using strict implication in background theories for abductive tasks

This chapter was published in the workshop on abduction at IJCAI-2001 [47]. Abduction is usually defined in terms of classical logical consequence. In this paper we substitute this 'inferential parameter' by the notion of strict implication. By doing so we hope to put more of the intended meaning of the abductive explanatory relation into the background theory. By using strict rather than material implication static domain knowledge for abductive tasks can deal with some limitations of the truthfunctional nature of material implication. It can be proved that by using strict instead of material implication the same explanations can be computed according to a strict version of abduction.

5.1 Introduction

In knowledge engineering it is common practice to separate the representation of static \(^1\) domain knowledge from the procedural method for solving a knowledge intensive task. The reason for this is that such separation facilitates the reuse of the domain knowledge. When constructing domain models one has to realize that they might also be looked at outside the scope of the task they were designed for.

In abductive tasks the static domain knowledge is usually represented as a logical theory, called the background theory. Abduction itself is often defined in terms of logical consequence and can therefore be considered a meta-logical notion. This meta-logical construct is not directly available at the object-level on which the background theory is formulated. As a result the background theory can be interpreted in two ways. First, since it is a classical logical theory it can be interpreted in the standard truth-functional way. Second, since it is input to an abductive procedure it has an 'explanative' interpretation. There is a discrepancy between the two interpretations.

When a domain model for an abductive task is written in propositional logic, the intended interpretation of this logical theory is the one in terms of possible explanations and observations. When the same theory is looked at outside the scope of the abductive task is has a standard truth-functional meaning.

\(^1\)By 'static knowledge' is meant knowledge that will remain invariant during the problem solving process.
Object-level characterizations of abduction do exist [19] and show that the semantics of background theories is really that of completed theories, (in the sense of Clark-completion [18]).

Our approach is to leave the meta-logical nature of abductive inference intact, and instead to attempt to put more of the intended meaning on the object-level by changing the inferential parameter which occurs in the definition of abduction. By doing so we hope to solve some problems that may hamper the interpretation of domain theories outside the scope of the task. As an example of a different inferential parameter we will look at strict implication and define a notion of strict abduction accordingly.

This paper is structured as follows. First abduction is defined in general terms and the problems regarding the semantics of the domain knowledge are explained. The following section mentions a number of issues regarding the use of material implication that are at odds with the intended meaning of the representation. Next we introduce the notion of strict implication and define strict abduction in terms of it. It will be shown how strict implication deals with the aforementioned issues. A method for computing strict abduction is given in terms of analytical tableaux for the modal language T. Although this method is more complicated than that of a tableau method for classical abduction, for some theories the results are equivalent. Finally, a summary and general comments on our approach are given in the form of a discussion.

5.2 Abduction and representation

Abduction can be generally defined as follows [3]:

Definition 5.1 : Given a logical theory \( \Theta \) (a background theory) and a formula \( \beta \) (an observation), \( \alpha \) is an abductive explanation for \( \beta \) iff the following properties hold:

Implication: \( \Theta, \alpha \Rightarrow \beta \)

Consistency: \( \Theta, \alpha \Rightarrow \beta \) and \( \Theta, \alpha \) is consistent.

Explanation: \( \Theta, \alpha \Rightarrow \beta \) and \( \Theta \not\Rightarrow \beta \) and \( \alpha \not\Rightarrow \beta \)

Minimality: \( \Theta, \alpha \Rightarrow \beta \) and \( \alpha \) is the weakest such explanation.

(\( \alpha \) is the weakest explanation if \( \Theta, \alpha \Rightarrow \beta \) and for all other formulas \( \circ \) such that \( \Theta, \circ \Rightarrow \beta \), \( \circ \not\Rightarrow \beta \).

Here \( \Rightarrow \) is a meta-logical symbol denoting an 'inferential parameter' [2] which relates the background theory and explanation to the observations. This can be classical consequence or any other consequence relation, or any other 'implicative' relation. In the case that by the inferential parameter is meant classical consequence we will refer to the abductive variant as classical abduction.

It is important to note that this definition of abductive explanation is a meta-logical one. In fact, the relation abductive explanation itself can be seen as a consequence relation, see for example Flach [32].

In an abductive task, domain knowledge should somehow link explanations to observations. This static domain knowledge is represented in the background theory, which is part of the object-level language. The abductive explanation relation itself is not directly available at this level.

A consequence relation (for a logic \( L \)) is a meta-logical notion as well but it can be expressed on the object-level as implication, if the deduction theorem holds in \( L \):
\[ \Theta, \alpha \vdash L \beta \iff \Theta \vdash L \alpha \rightarrow \beta \]

As a result the domain knowledge necessary to link explanations to observations can be represented by the implication symbol of \( L \). As a result, when compiling knowledge for background theories it is tempting to treat the implication symbol as the actual abductive explanatory relation itself.

In the process of representing domain knowledge for an abductive task the use of implications is often taken to represent the meta-logical abductive explanatory relation. But when we look at classical abduction the semantics of material implication does not reflect the intended meaning of its use. In fact there is a discrepancy between the meta-logical notion of abduction and the semantics of the object-level language.

### 5.3 Pragmatic issues

In order to illustrate that there is divergence between the object-level representation of the domain knowledge and the meta-logical construct of abductive explanation we will take a closer look at some issues regarding classical abduction and material implication.

These are inspired by pragmatic considerations about the representation of static domain knowledge. These issues also reflect the wish to put more of the intended meaning in the semantics of the object-level representation.

In the following subsections by 'abduction' 'classical abduction' is meant, by 'implication', 'material implication' and by 'consequence', 'classical consequence'.

#### 5.3.1 Negation

By the above definition of abduction \( \alpha \) is not an explanation if one of the mentioned properties fails to hold. For example, suppose for some \( \alpha \) it is the case that \( \alpha, \Theta \not\vdash \beta \). To express this explicitly in the background theory cannot be done by simply negating the implication \( \alpha \rightarrow \beta \). The reason for this that \( \neg (\alpha \rightarrow \beta) \vdash \alpha \land \neg \beta \), which is clearly not what is intended.

However, one could view the background theory as a closed world in the sense that \( \alpha \) is not an explanation for \( \beta \) if it can not be proved that \( \alpha \) is an explanation of \( \beta \). In such theories every abducible is either an explanation of an observed phenomenon or not.

Formally we can define this as follows:

**Definition 5.2**: Completeness assumption for background theories: Given \( \Theta \) (a background theory) and \( \beta \) (an observation), \( \alpha \) is not an abductive explanation for \( \beta \) iff \( \Theta, \alpha \not\vdash \beta \) and \( \Theta \cup \alpha \) is consistent.

The idea that \( \alpha \) is not an explanation of \( \beta \) by failure to proof that it is, might seem over-constrained. For many explanatory models this completion assumption seems to be too strong. Console et al. [20] circumvented this objection by the use of incompleteness-assumption symbols. Let \( A \) be such a symbol, then \( A \rightarrow \beta \) expresses that \( A \) is an unknown explanation of \( \beta \). This construction leaves room for explicitly stating that the logically completed theory is not complete in a pragmatic sense.
5.3.2 Problems with conditionals

There are a number of problems regarding the use of material implication in conditionals which are well documented (for an overview see Veltman [94]). The main issue that concerns us here is that the truth-functional meaning of conditional statements may lead to confusion when they are used to represent background knowledge for abductive tasks.

Any (material) conditional statement $\alpha \rightarrow \beta$ can be reformulated as $\neg\alpha \lor \beta$. This last formulation makes an interpretation of the implication as explains cumbersome.

The conditional problems with material implication are perhaps best illustrated by what are known as the 'paradoxes' of material implication: Consider the following two sentences, both theorems of propositional calculus (PC):

(1) $p \rightarrow (q \rightarrow p)$

(2) $\neg p \rightarrow (p \rightarrow q)$

The first one can be said to mean that if a proposition is true any proposition implies it. The sense of the second is that anything is implied by a proposition which is false. From these the following tautology can be derived:

(3) $(p \rightarrow q) \lor (q \rightarrow p)$

Calling this a paradox might suggest that this is a fallacy, but it is not. The above sentences merely reflect the truth-functional meaning of material implication.

From these it follows that the semantics of material implication is indeed different from that of abductive explanation. For example theorem (3) would say that for every two phenomena one is the explanation of the other. The contrary, that two phenomena are each not to be considered as an explanation of the other, can therefore not be expressed. Clearly, this is not in line with the definition of abductive explanation.

5.3.3 Necessity and possibility

In more complex domains often expressions are needed which express the necessity or possibility of explanatory relations. Classical logic offers no standard way of modelling constructs like $A$ possibly causes $B$ and $A$ necessarily causes $B$. Several solutions using classical logic have been proposed though.

Poole [69] advocates an approach where possible and necessary causal implications are represented by two different sets. One, named $H$, contains general or open hypotheses and the other $F$, closed formula or facts. The interpretation is that $H$ contains implications which should be interpreted as possible, whereas elements of $F$ should be interpreted as necessary implications.

Another approach is the above described use of assumption symbols in Console et al[20]. Here the fact that $A$ MAY cause $B$ is represented as $A \lor \alpha \rightarrow B$ where $\alpha$ stands for an hypothetical assumption the truth of which is first assumed but which may be rejected if their evaluation gives rise to unwanted results.

However, the modalities possibility and necessity are not directly available at the object-level. Using a modal instead of a truth-functional semantics would solve this issue.
5.3.4 Non-deterministic explanation

By an non-deterministic explanation is meant here that some phenomenon $\alpha$ can explain observations $\beta$ or $\gamma$. As $\alpha$ occurs $\beta$ or $\gamma$ will result but it isn’t known which one. The non-deterministic aspect indicates a choice between $\beta$ and $\gamma$.

When representing knowledge for an abductive task, one could make use of a conditional of the form:

$$\alpha \rightarrow \beta \vee \gamma$$

However it is not clear what such a conditional should mean. If we take the viewpoint that implication is interpreted as an explanatory relation its intended meaning would be something like: $\alpha$ explains $\beta$ or $\gamma$.

However when performing abduction and $\beta$ is observed this conditional will not lead to the conclusion that $\alpha$ is an explanation.

$$\alpha \rightarrow \beta \vee \gamma, \alpha \doteq \beta \vee \gamma$$

And $\beta \vee \gamma$ does not imply $\beta$.

This problem occurs frequently in classification tasks. Consider the following representation of the fact that a blackbird is either black or brown:

blackbird $\rightarrow$ (colour = brown $\vee$ colour = black)

The attribute colour has as possible values $\{\text{black, brown}\}$. Observations are represented as attribute-value pairs where only one value per attribute is allowed. Therefore if the observation colour = black is made, it follows that colour $\neq$ brown.

At first sight blackbird seems to explain this observation, but closer inspection learns that it does not. Though the observation is consistent with blackbird, it is not implied by it.

5.4 An alternative approach

The issues raised in the previous section were meant to draw attention to the fact that the semantics of the object-level language in which the background theory is formulated does not reflect the intended meaning of the abductive explanatory relation.

One solution is to approach the notion of abductive explanation as an object-level notion itself. This has been described by Console et al. [19] and Konolige [54]. From the first [19] it has become clear that for cycle-free background theories which contain only definite clauses, their semantics in an abductive setting is really that of completed theories.

The problems raised in the previous section can be dealt with in this manner. For example, it becomes clear that to express explicitly in a background theory that some $\alpha$ is not to be regarded as an explanation, has a strong resemblance to negation as failure. Possibility can be expressed in completed theories as well, see Console et al[20].

Here we take a different approach. Instead of bringing the notion of abduction to the object-level we try to bridge the gap between observation and explanation by looking at another logical system. So instead of substituting classical consequence for the inferential parameter in the general definition of abduction given above, others can be tried.

There is however one potential drawback to this approach. A decision procedure for an inferential parameter $\Rightarrow$ can be adapted in order to compute abduction with. (An example method
in the form of analytical tableaux will be described later.) Another notion of logical consequence in the definition of abduction, will generally lead to a different method to compute abductive explanations. In order to avoid complicating this procedure care must be taken when choosing an inferential parameter. If the resulting logic has no or an arduous decision procedure this will complicate the abduction procedure considerably.

Here we will describe one alternative inferential parameter: a system for strict implication, (strict implication will be denoted by the symbol \( \rightarrow \)). It is important to note that if in a propositional logic we replace every occurrence of \( \rightarrow \) by \( \rightarrow \) the result is a weaker logic, in the sense that every theorem of the latter is also one of the former (but not vice versa). Strict implication is certainly not the only candidate. Another option would be to use intuitionistic logic, which we will leave for future research.

5.5 Strict implication

The notion of strict implication was first put forward in modern times by Lewis, [56]. The intention was to come up with a different notion of implication which would not lead to the paradoxes of material implication, mentioned above.

The intuitive meaning of strict implication can be formulated as follows: A sentence of the form \( \alpha \rightarrow \beta \) is true in a given situation \( s \) iff there is no possible situation \( s' \) such that \( \alpha \) is true in \( s' \) and \( \beta \) is false in \( s' \). Lewis' distinction between strict (or necessary) and material implication marked the birth of the development of modern modal logic.

In modern notation strict implication (\( \rightarrow \)) can be defined as
\[ \Box(\alpha \rightarrow \beta) \text{, or alternatively } \neg \Diamond (\alpha \land \neg \beta). \]

The modal operators \( \Box \) (necessity) and its dual \( \Diamond \) (possibility) now facilitate to express directly the distinctions between three kinds of propositions: The tautologies or necessary true propositions (represented by \( \Box \alpha \)), the contradictions or impossible propositions (represented by either \( \neg \Box \neg \alpha \) or \( \neg \Diamond \neg \alpha \)) and the contingencies which are neither contradictions nor tautologies (represented by either propositional variables or \( \Diamond \alpha \)).

The weakest system to capture these ideas is the modal logic T.\(^3\) Stronger systems like S4 and S5 could be used but these are extensions of T. Furthermore the decision procedure for a logic can often be adapted in order to perform abduction with. For example Aliseda Llera [3] uses semantic tableaux. Modal tableaux procedures for T and S4 exist [31, 37] but not for S5. Not surprisingly the procedure for S4 is more complicated than the one for T. This general tradeoff between complexity and expressiveness is the main motivation to opt for a weak alternative for classical abduction.

However, as said above just replacing material by strict implication yields a weaker propositional logic, whereas T is stronger than classical propositional logic: it contains PC. The reason for choosing T in preference to the weaker strict implication logics, is that the latter do not come with a decision procedure, and so computing abduction in them is not evident.

Syntactically T contains PC and in addition has the following two axioms:

\[ \Box \alpha \text{ can be defined as } \neg \Diamond \neg \alpha. \text{ Similarly } \Diamond \alpha \text{ can be defined in terms of } \Box. \]

\[ \text{Lewis originally came up with weaker systems S1 and S2. These do not make use of the modalities of necessity and possibility. T is the weakest normal system. For a discussion of Lewis' systems and the appropriateness of T, the interested reader is referred to Hughes & Creswell [43].} \]
Using strict implication in background theories for abductive tasks

(K) \( \Box(\alpha \rightarrow \beta) \rightarrow (\Box \alpha \rightarrow \Box \beta) \)

(T) \( \Box \alpha \rightarrow \alpha \)

and the following extra rule:

(Necessitation) If \( \alpha \) is a theorem of \( T \) so is \( \Box \alpha \).

Note that the deduction theorem, mentioned above, holds for \( T \). The semantics of the modal logic \( T \) can be characterized by two key notions: possible worlds and accessibility relation. A Kripke-model is built up from several possible worlds, representing as many possible states of affairs. In these models truth-values are always connected to possible worlds rather than formulas. In determining the truth-value of a purely propositional formula in a world other worlds play no role. Only if \( \Box \) occurs may it be necessary to involve other worlds.

Possible worlds are connected by means of an accessibility relation which together constitute a frame. Worlds may be accessible from or visible for each other. A formula \( \Box \alpha \) is true in world \( w \) if \( \alpha \) is true in all worlds \( w' \) accessible from \( w \). A formula is valid if it is true under any interpretation in every world.

The semantics of \( T \) differs from that of other modal systems in that its frames (characterized by the \( T \)-axiom) are reflexive. For a more formal treatment of modal logic semantics the reader is referred to Hughes and Creswell [43].

The notion of strict implication makes use of the modality necessity. Hence, using strict implication instead of material implication means making use of a modal instead of a truth-functional semantics.

5.6 Strict abduction

The purpose of this exercise is to see if we can use strict instead of material implication in abductive background theories. In order to do this we define the notion of strict abduction. This comes down to substituting strict implication for the inferential parameter in the general definition of abduction described in section 2. In addition it should be clear that all reasoning should be done in the system \( T \).

Definition 5.3 : Given \( \Theta \) (a background theory) and \( \beta \) (an observation), \( \alpha \) is a strict abductive explanation for \( \beta \) in the system \( T \) iff:

Implication: \( \Theta \vdash_T \Box(\alpha \rightarrow \beta) \)

The rest of the properties can be defined in a similar way.

Note that strict abduction is not the same as abduction in \( T \). \( T \) contains PC and performing abduction in it would lead to a more complicated procedure for computing abductive explanations. Just replacing strict for material implication in a background theory does not lead to a more complicated procedure, as will be shown below.

Let \( \Theta \) be a purely propositional (consistent) background theory consisting of purely propositional implications of the form \( \alpha \rightarrow \beta \). \( \alpha \) is here an abducible atom, a disjunction or conjunction of atoms.
Let □Θ be the set \(^4\) where each material implication in Θ has been replaced by a strict implication. Furthermore let β be an observation. Now the following holds:

**Proposition 1** \(α\) is a *classic* abductive explanation for \(β\) according to \(Θ\) iff \(α\) is a *strict* abductive explanation for \(β\) according to □Θ.

**Proof:** It suffices to show that

\[ \vdash Θ \rightarrow (α \rightarrow β) \iff \vdash □Θ \rightarrow □(α \rightarrow β) \]

(\(\Rightarrow\)) Since \(T\) contains PC: \(\vdash T Θ \rightarrow (α \rightarrow β)\) then by necessitation and the K-axiom it follows that \(\vdash T □Θ \rightarrow □(α \rightarrow β)\).

(\(\Leftarrow\)) This proof depends on the fact that each thesis of \(T\) has a PC-transform which is valid in PC. The PC-transform of a formula \(α\) is formed by rewriting it in a form containing only \(\sim, \land, \lor\) and \(\Box\) and then removing every occurrence of \(\Box\). For the proof see Hughes and Creswell [42]. It can be verified that the PC-transform of \(□Θ \rightarrow □(α \rightarrow β)\) is \(Θ \rightarrow (α \rightarrow β)\).

This means that there is no difference between strict and classic abduction if the only difference between the domain theories is that the strict one contains strict implications where the classic one contains material implications.

In fact it also shows that in background theories for classical abductive tasks the implications which occur in the background theory can be safely interpreted as strict implications. They can also be interpreted as possible implications.

When it is clear that the implications in the background theory should be interpreted as necessary implications there is no need for a special procedure for strict abduction.

However, when combining the necessary and possible explanations such a procedure is needed. In the next section we describe such a method in terms of analytical tableaux in \(T\). As an introduction to these ideas we first give a quick introduction into the method of analytical tableaux for propositional logic, and propositional classical abduction. Those familiar with the tableau method can skip the next section.

### 5.7 Analytical tableaux for abduction

Decision procedures for classical logic like resolution and analytical tableaux can be used in a 'reverse manner' for abductive reasoning. To illustrate the ideas presented in this paper analytical tableaux will be used. The choice over other methods is purely a pragmatic one. Tableaux are a quite popular method in automated theorem proving [30] and their application in the field of abduction is well-documented [3] [62].

In a tableau proof a set of formulae is transformed into binary tree by means of reduction rules. These rules are depicted in table 1.

The first rule indicates that double negations are redundant. The second and third rule deal with the atomic constants \(⊤\) and \(⊥\). All propositional formulas containing binary connectives can be divided as belonging to two types: True conjunctive formulas (\(α\)-type) and true disjunctive

\(^4\) Although the domain theories are defined as sets, they are sometimes used as a conjunction of their elements. Note that □ can be distributed over conjunctions: □(α ∧ β) → □α ∧ □β.
formulas (β-type) [30]. The rule for α-type formulas indicates that the conjuncts have to be placed on the same branch of the tree. The β-rule however indicates a branching of the tree.

If at any branch a formula and its negation appear, the branch is said to be closed. If all branches close, the tree is said to be closed. A tableau tree \( T(\Theta) \) for a theory \( \Theta \) has the following two general logical properties: Every open branch in a tableau \( T(\Theta) \) corresponds to a verifying model. If \( T(\Theta) \) is closed, \( \Theta \) is inconsistent.

In order to test whether a certain formula \( \alpha \) follows from a set of premises \( \Theta \) a tableau tree is constructed for \( \Theta \cup \{\neg \alpha\} \). Only if the constructed tableau closes does \( \alpha \) follow from \( \Theta \).

The procedure for performing plain abduction with tableaux is as follows: The tableaux \( T(\Theta \cup \{\neg \beta\}) \) is generated. Assuming that \( \beta \) does not follow from \( \Theta \) alone, this results in an open tableau. Any formula \( \alpha \) which closes the tableau when added to it is an explanation.

The various styles of abduction can then be described as follows: Let \( \Theta \) be the domain theory, \( \beta \) the observation, then \( \alpha \) is an explanation if:

**Plain:** \( T((\Theta \cup \{\neg \beta\}) \cup \{\alpha\}) \) is closed

**Consistent:** Plain +
\( T(\Theta \cup \{\alpha\}) \) is open.

**Explanatory:** Plain +
\( T(\Theta \cup \{\neg \beta\}) \) is open and \( T(\alpha \cup \{\neg \beta\}) \) is open.

**Minimal:** Plain +
\( \alpha \) is minimal.

In addition abducible should be part of the vocabulary of the domain theory and the observation, and be either literals or conjunctions or disjunctions of literals. For minimal abduction literals should be checked first. If none of these close the tableau conjunctions of literals should be checked for, etc.

### 5.8 Strict implication in T

Tableaux in T (or T-tableaux) can be constructed with the help of the reduction rules for PC and two additional rules for the modal operators, shown in table 2.\(^5\)

Where \( \square \Sigma \) is a set of boxed formulae, \( \Sigma_1 \) a set of non-boxed formulae and \( \alpha, \Sigma, \Sigma_1 \) is short for \( \{\alpha\} \cup \Sigma \cup \Sigma_1 \).

The interpretation of these rules is different from those for PC. After application the formulae on the top side of the line must be deleted from the current branch. The reason for this is that the

\[
\begin{array}{ccccccc}
\text{\( \neg \neg Z \)} & \text{\( \rightarrow Z \)} & \text{\( \neg \bot \)} & \text{\( \bot \)} & \text{\( \alpha \)} & \text{\( \beta \)} \\
\text{\( \bot \)} & \text{\( \top \)} & \text{\( \alpha^1 \)} & \text{\( \beta^1 \)} & \text{\( \alpha^2 \)} & \text{\( \beta^2 \)} \\
\end{array}
\]

**Table 5.1**
Rules for the tableau trees.

\(^5\)A more elaborate description of tableaux for modal systems can be found in [31] and [37].
two modal rules mimic the transition to another world. Boxed formulae are then stripped from a box and unboxed formulae disappear, as they are local with respect to the world they occur in.

With the help of semantic tableaux for \( T \) strict abduction can now be performed. Let \( \Theta \) be the domain-theory, \( \beta \) the observation and \( \alpha \) an explanation. The goal is to derive \( \Box(\alpha \rightarrow \beta) \) in \( T \) for some formula \( \alpha \). Using tableaux this can be done as follows:

Generate the \( T \)-tableau \( T(\Theta \cup \{ \neg \Box(\alpha \rightarrow \beta) \}) \). Then check which formulae when substituted for \( \alpha \) result in closure of \( T \).

Apart from the fact that \( T \)-tableaux instead of PC-tableaux are used, a procedure for strict abduction differs in two ways from a normal abductive procedure. First, not the negation of the observation is added to the tableau of the domain theory, but instead the negated strict implication from explanation to observation. Second, explanations are not those formulae which close the tableau when added to it, but when substituted for \( \alpha \).

### 5.9 Issues revisited

If instead of material implication strict implication is used, the issues mentioned before can be treated directly in the language of the background theory.

**Negation.** To express that \( \alpha \) does not explain \( \beta \) can be expressed by \( \neg \Box(\alpha \rightarrow \beta) \). In fact the semantics of this statement compared to the material conditional \( \neg(\alpha \rightarrow \beta) \) is much closer to the intended meaning. As the last conditional means \( \alpha \) and not \( \beta \), the strict version has the meaning that it is possible that \( \alpha \) and not \( \beta \).

Note that if \( \neg \Box(\alpha \rightarrow \beta) \) is contained in the background theory \( \alpha \) is indeed not an explanation for \( \beta \) since the implication property in the definition of strict abduction does not hold.

**Conditional problems.** The notion of strict implication was developed out of dissatisfaction with the paradoxes of material implication. These do not hold for strict implication. In that sense strict implication is to be preferred over material implication. However, strict implication is not the ideal candidate for solving all problems concerning conditional statements.

Like material implication \( \neg \Box \) is still a connective and does not really express the \( \text{connection} \) between explanation and observation. This objection can be put in the form of 'paradoxes of strict implication':

\[
\begin{align*}
(p \land \neg p) & \not\rightarrow q \\
qu & \not\rightarrow (p \lor \neg p) \\
\Box \neg p & \rightarrow (p \not\rightarrow q) \\
\Box q & \rightarrow (p \not\rightarrow q)
\end{align*}
\]

\[
\frac{\Box \neg \alpha, \Box \Sigma, \Sigma_1}{\neg \alpha, \Sigma} \quad \frac{\Box \alpha, \Sigma}{\Box \alpha, \alpha, \Sigma}
\]

**Table 5.2**

Additional rules for \( T \)-tableaux.
All these sentences are theorems of T. The meaning of each of them can be expressed respectively as follows: a contradiction strictly implies everything. Every proposition strictly implies a tautology. Every impossible proposition strictly implies everything. When a proposition is necessary it is strictly implied by everything. Similar to the paradoxes of material implication these sentences may give rise to unwanted interpretations.

**Possibility and necessity** The modal semantics of T now facilitates expressing possibility and necessity directly in the object-level language. However, this would mean that the notion of abduction should be adapted in order to make the distinction between possible and necessary (strict) explanations.

In fact *possible* abduction can easily be defined as strict implication, except that each occurrence of $\Box$ is replaced by $\Diamond$. For possible abduction a similar result as that of proposition 1 can be proved.

**Proposition 2** $\alpha$ is a classic abductive explanation for $\beta$ according to $\Theta$ iff $\alpha$ is a possible abductive explanation for $\beta$ according to $\Diamond \Theta$.

$$(\Rightarrow) \quad \Theta \rightarrow (\alpha \rightarrow \beta) \text{ holds in PC and therefore in T. So:}$$

$$
\begin{align*}
\vdash_T & \neg(\alpha \rightarrow \beta) \rightarrow \neg \Theta \Leftarrow \\
\vdash_T & \Box \neg \Theta \rightarrow \Box \neg(\alpha \rightarrow \beta) \Leftarrow \\
\vdash_T & \Box \neg \neg(\alpha \rightarrow \beta) \rightarrow \Box \neg \neg \Theta \Leftarrow \\
\vdash_T & \Diamond(\alpha \rightarrow \beta) \rightarrow \Diamond \Theta \quad \text{6}
\end{align*}
$$

$$(\Leftarrow) \quad \text{Since } \Diamond \text{ can be defined as } \neg \Box \neg, \text{ the PC-transform is the same as that of the boxed version.}$$

This result shows that material implication can be interpreted in two ways: As possible or necessary explanation. Using both at the same time would mean performing strict abduction in T. This would lead to the more complicated abductive procedure (described above) than the standard case.

**Non-determinism and abduction** Suppose $\alpha$ explains either $\beta$ or $\gamma$. If $\beta$ is observed will an abductive procedure produce $\alpha$ as an explanation?

The an answer partly depends on the representation of the explanatory rule. Just using material implication would give us $\alpha \rightarrow (\beta \lor \gamma)$. Then clearly $\alpha$ does not imply $\beta$. However one could opt for a *weak* variant of abduction where $\alpha$ is an explanation only if it is consistent with the observation. Although this would give the desired result, this procedure will often lead to numerous unwanted explanations as well.

Another problem with such a weak variant of abduction is that the relevance of the explanation for the observation is still lower than for classic abduction.

Strict implication does not provide for an ideal solution to this problem. $\alpha$ is not a strict explanation for $\beta$ if the explanatory rule is represented as $\Box (\alpha \rightarrow (\beta \lor \gamma))$.

However making a distinction between possible and necessary causal relations does offer a solution. Non-determinism expresses a choice, or possibility, between alternatives. The non-deterministic causal rule could thus better be represented as: $\Diamond (\alpha \rightarrow \beta) \land \Diamond (\alpha \rightarrow \gamma)$.

---

6Note that $\Diamond$ can be distributed over conjunctions: $\Diamond (\alpha \land \beta) \Rightarrow \Diamond \alpha \land \Diamond \beta$. 
Using this representation $\alpha$ is not a strict but a possible explanation for $\beta$.

However this solution still does not completely explain some cases. Consider the example where one disease has as symptoms red skin and another either skin-rash or fever. Observing skin-rash would mean the first disease to be a necessary, the second a possible explanation. Since possibility is the weaker notion this suggests that the the necessary one is to be preferred. The question remains if this is really the desired result.

5.10 Discussion

By using strict implication and defining abduction in terms of it, the semantics of the object-level representation changes from a truth-functional to a modal one. Still the same explanations can be computed on a strict background theory compared to the classical case. The interpretation of background theories in terms of strict implication has a number of advantages in that it deals with some pragmatic issues regarding the intended meaning of the knowledge representation.

We do not claim that strict implication is the only notion that could be used to solve problems like the ones discussed. In fact intuitionistic logic seems to be a good candidate as well. The semantics of this logic can be formulated in terms of information states, which is interesting from an abductive point of view. Furthermore this logic is weaker than classical logic.

The discrepancy between the semantics of the representation of static domain knowledge and meta-level construct is of interest for the field of knowledge representation. Abduction can be seen as a (general) problem solving method operating upon static knowledge representations. As such it interprets the domain knowledge in a partial way. As a result this 'procedural semantics' of the domain knowledge differs from the semantics of the representation language itself.
A dynamic account of knowledge level tasks

We present a framework for describing problem solving for knowledge intensive tasks in terms of dynamic change of knowledge. The aim is to provide a knowledge level description of what an agent knows before, during and after problem solving. Knowledge is taken here as a semantic notion related to the information-theoretic analysis of Dretske [23]. The formal layer of our representation is formed by update semantics [95]. We use (and interpret) this formalism to represent knowledge about possible solutions by means of knowledge states. Problem solving is described in terms of knowledge states, updates and epistemic tests. We distinguish between reactive and proactive updates, the use of memory and the specification of problem solving steps. Finally, we provide an example of how three classification criteria can be represented and compared within this framework.

6.1 Introduction

Knowledge intensive problems can often be classified as belonging to a certain type or task. Alternatively, some problems first have to be decomposed into several subproblems which each can be classified. The different tasks found in the literature [71, 87, 82] have surfaced over the years in knowledge engineering research.

With each task a set of Problem Solving Methods (PSMs) can be associated. Like tasks, PSMs are pragmatic rather than formal constructs. They can be seen as generalized methods of problem solving for a range of problems, belonging to the same task. Building knowledge intensive systems can benefit considerably from using available PSMs after the problem type has been established.

Since PSMs are generalized methods they often have to be adapted to the peculiarities of the problem at hand. From a formal point of view this pragmatic nature of PSMs and tasks leaves considerable room for vagueness. Logical properties of different problem types are often ignored. This becomes an issue when one tries to specify PSMs in a more precise way. For example, attempts have been made, and are still under development [7] to systematically compile the precise descriptions of problems that are covered by a task description. The aim of those endeavours is to categorize problem solving methods into libraries and make them available for large-scale reuse.

Both the goal of a task and its PSMs are usually described using vocabulary defined in a task ontology. The methods of a task operate on domain knowledge which is defined in a separate domain ontology. In addition a method has a typical control structure which defines its data flow.
Problem Solving Method

Task ontology
Domain ontology
Control structure
Dynamic knowledge
Static knowledge

Epistemic change
States
Tests
Updates
Reactive updates
Pro-active updates

Figure 6.1
A partial mapping between problem solving methods and epistemic change is shown. Problem solving methods will be described in terms of epistemic change. Knowledge states will capture both the dynamic and static knowledge of an agent. The flow of knowledge will be described in terms of updates and tests on these knowledge states.

The description of tasks in our framework will assume the existence of both task and domain ontologies. The task ontology will consist of formal definitions of the concepts used in the descriptions of the task. The domain ontology should do likewise for the domain knowledge. To this we add the distinction that some knowledge remains invariant or static during problem solving, whereas some will be subject to change and is dynamic. The main focus will be on the dynamic characterization of the knowledge flow.

Problem solving will be treated as an attempt to acquire knowledge about possible solutions. Our view on knowledge is in line with the information theoretic notion as described in Dretske [23]. It is also similar to the one described in Fagin et al. [24] for multi-agent systems.

The information theoretic account of knowledge links the amount of knowledge to the number of possible states of affairs. A knowledge-level agent has no knowledge if all possibilities are open. His knowledge increases when the number of possibilities decreases. We present a semantic framework of epistemic change, based on an information theoretic view on knowledge, and use it to describe problem solving behaviour.

The formal heart of our framework is formed by update semantics as described by Veltman [95]. It allows for standard as well as non-standard reasoning to be cast in terms of information states, updates and epistemic tests. However, we present a particular interpretation of update semantics. We will use information states as knowledge or solution states. They should capture the knowledge an agent has during each stage of the problem solving process. State transition takes place by processing new information in the form of updates. Epistemic tests enable one to verify properties of a state without state change.

Hence, we use update semantics to describe changing knowledge during problem solving. For this purpose we introduce the distinction between reactive and proactive updates. Reactive updates describe changes in knowledge of the agent induced by the environment. Proactive updates reflect decisions made by the agent itself and make use of a memory. Problem solving can then be described as a series of updates and tests on successive knowledge states. This flow of knowledge will be illustrated by diagrams.

The focus of our description will not be on the specification of efficient methods, or on the complexity of problems, but rather on a description of what knowledge an agent has before, dur-
ing and after solving a problem. Specifications will be given in terms of knowledge-level [67] constructs, such as knowledge state and knowledge update.

The framework presented here provides a semantic description of tasks in terms of epistemic change. We take a "model-based" approach in our characterization of an agent's knowledge about the solution of a problem. Model-based approaches to reasoning have been proposed by several people [49, 51, 53].

A knowledge base (KB) is here presented in terms of models. In its most elementary form we think of a KB as the set of all models under which it is true. Computationally this set is usually too large. However, computational inviability is not inherent to a model-based approach. Research has been directed to keep the set of models of a KB as small as possible. We refer to Khardon and Roth [53] for a detailed description of this. Their approach will be briefly described in section 6.5.1.

6.2 The dynamics of tasks

The framework presented here will be model-based in the sense that the search for a solution will somehow be linked to finding one or several models which pass some epistemic test. This epistemic test is the criterion of the problem, which describes when a solution has been found.

The knowledge of an agent is related to the possibilities it considers at a given moment in time. Problem solving can be described in terms of increasing knowledge about candidate solutions. The amount of knowledge an agent has about the solution is inversely proportional to the number of candidate solutions it considers. As the number of possible solutions decreases the amount of knowledge about solutions increases. In this light problem solving is a process of acquiring knowledge about possible solutions. This means that one could ask at any stage during the problem solving process of an agent: what does the agent know about the solution of the problem? The answer will be a characterization of which solutions will be possible and which are not.

6.2.1 Problem solving in terms of changing knowledge

An example taken from van Benthem [91] may clarify what kind of problem solving we have in mind.

Example 6.1: Consider a game of Master Mind where the purpose of the game is to guess the positions of coloured pegs.

<table>
<thead>
<tr>
<th>Guess</th>
<th>Answer</th>
<th>Open options</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>• o</td>
<td>24</td>
</tr>
<tr>
<td>red, orange, white</td>
<td>• o</td>
<td>6</td>
</tr>
<tr>
<td>white, orange, blue</td>
<td>• o</td>
<td>2</td>
</tr>
<tr>
<td>blue, orange, red</td>
<td>o o</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.1
A game of Master Mind.

In each round the player is allowed one guess, after which he gets feedback by way of a number of open or closed dots. An open dot indicates a right colour at a wrong position, a
closed one a right colour at a right position. Table 6.1 shows a game with three pegs and 4 colours (red, white, blue and orange). The arrangement to be guessed is (red, white, blue). The column on the right shows the number of open options after each round. This style of reasoning can be described as an interplay of incoming information and changing knowledge about solutions.

A knowledge level description of a problem and the process of solving it, involves a characterization of what the player knows before, during and after having solved the problem. In general we give an account of the problem solving process of a knowledge-level agent, in the sense used by Newell [67].

In case of the Master Mind game, before the start of the problem solving process the player, or agent, knows a number of things: he (assuming the player is male) knows the rules of the game, the possible colours of the pegs etc. Also the player knows that only one configuration can be a solution, and that there is no sense in considering the conjunction of several configurations. All such knowledge will remain invariant during the problem solving process; it is not subject to change during the problem solving process.

The dynamic or variant knowledge the player has is mainly about the solution of the problem he faces. It is safe to say that before problem solving the player has no knowledge about the solution. In other words: before problem solving any configuration of the pegs could be a solution. After having solved the problem this situation is changed and the player knows which configuration is the solution. During problem solving the number of possible configurations has steadily decreased, as is evident from table 6.1. In other words his knowledge about the solution has increased.

This view on knowledge is elaborately explained and refined by Dretske in his book Knowledge and the Flow of Information [23]. In this view knowledge and information are related notions and are strongly linked to the reduction of possible states of affairs. We adapt and use this theory here in order to present a knowledge level account of problem solving.

6.2.2 Knowledge and Information: states and updates

We describe problem solving in terms of knowledge in a dynamic way. By this we mean that we give an account of how knowledge changes while the problem is being solved. In order to do this we introduce knowledge states and state transitions. A more formal account of the notions introduced here will be given later.

A knowledge state is intended to capture what an agent knows. However, as we will see, not everything which would qualify as knowledge will be represented in a knowledge state. For example, procedural knowledge will not be part of an agent's knowledge state.

When some sentence is accepted in a certain knowledge state then the agent knows the sentence. By "accepting" we mean that the knowledge state does not contain anything which refutes the sentence. A more formal definition will be given later.

A knowledge state consists of two types of knowledge: the domain knowledge which will remain invariant during problem solving, and knowledge regarding the possible solutions to the problem. Hence a knowledge state will have a static and a dynamic part.

As remarked above, knowledge is strongly linked to possibilities. Logically, possibilities will later be described as models. From this perspective there are two knowledge states which are of special interest. The first one is the state in which everything is possible. This one is called the
minimal knowledge state because if everything is possible then nothing is known. The other knowledge state is the one in which nothing is possible, it is empty or logically inconsistent. This state is known as the absurd state.

The domain knowledge associated with a problem will be part of any state during the problem process. However, the dynamic part of the state changes. Since we use the dynamic part of the knowledge state to keep track of the possible solutions to the problem, we will refer to it as the solution state.

A state transition gives an account of how one state may change into another one. This will be done by means of updates. We distinguish two types of updates: reactive or informational updates (called 'updates' for short), and proactive updates. The first ones intend to capture passive state transitions on the basis of incoming information. For example, knowledge in diagnostic problem solving may change after observations have been made. These observations can be seen as containing new information which affects the knowledge of the agent about possible malfunctions of the artefact to be diagnosed. In other words: the knowledge state is updated with new, observed information.

The reason why we call such updates 'passive' is that we will only deal with expanding knowledge states, not with revisions. Any knew knowledge state which is the result of a state transition will contain at least as much knowledge as its predecessor. (In terms of possibilities the states contract rather than expand, since increase of knowledge means decrease of possibilities.) In the belief revision literature [34] belief state revisions are described when incoming information is inconsistent with beliefs held by the agent. In this case some beliefs should be given up or adapted to conserve consistency of the belief state.

Our framework will be limited in that we only deal with states of expanding knowledge and not with revisions. When information is inconsistent with the knowledge of a state this will lead to the transition to the absurd state. Informational updates are passive because they give rise to purely deterministic changes of knowledge states.

We also make use of proactive updates. These updates reflect the rational capabilities of an agent, they often involve the selection of a certain element, a guess or a move. Proactive updates are state transitions which are initiated by the agent itself, rather than a deterministic reaction to information from the environment.

Proactive updates are indeterministic and therefore it is often necessary to return to previous states. In other words: the agent needs some kind of memory to backtrack on the choices it made. Like informational updates we restrict proactive updates by only allowing that a state will have as least as much knowledge as its predecessor.

In the Master Mind game example the guess of the agent at each round can be represented as a proactive update. By guessing a configuration the player ignores all other candidate solutions and changes to a solution state where only this guess is a solution. It's important to note that this move is done on the initiative of the agent itself, and is therefore proactive. The indeterministic aspect of proactive updates is given by the fact that a guess can be right or wrong. When the guess is wrong the agent can, by using it's memory, move back to his previous state. When his guess is affirmed to be correct a solution has been reached.
6.2.3 Epistemic test and criteria

Similar to updates are epistemic tests [95]. Epistemic tests (or ‘tests’ for short) do not lead to state transition. Instead they are hypothetical updates which are used to test which effect an update would have on the current knowledge state.

As an example (which will be treated in detail later) consider the question whether a given expression is consistent with a certain knowledge state. The answer to this question is either ‘yes’ or ‘no’ and there is no need for a transition to a new state. This test for consistency is an example of an epistemic test.

Another example is the test for acceptance of some expression with respect to a given state. (We will present a formal meaning of this notion later.) By this we mean the question whether the expression can be derived from the given knowledge state. Here again we get a boolean answer (yes/no) as output of the epistemic test.

An important use of epistemic tests in our framework is as a specification of a final state, i.e. a state wherein a solution to the problem has been reached. This is done by defining the criterion of the problem as an epistemic test. As mentioned in chapter 4 the criterion of the problem is an expression which states when a candidate solution is in fact a solution.

In the case of the Master Mind game the criterion is that a solution should match the target configuration. This can be formulated as an epistemic test as follows: The test succeeds for a given solution state if it contains only one configuration which matches the target configuration. Otherwise it fails.

Epistemic tests are also used here as boolean constructs in the description of control flow. Problem solving is described as a series of tests and updates together with some flow of control. The flow of control describes the order of updates and test together with possible iterations.

As an example we will give a description of the Master Mind game in terms of a diagram. In figure 6.2 a pictorial representation is given in a flow-chart notation. States are depicted as rectangles, updates and epistemic tests as ovals. The tests are followed by a choice symbol. The diagram should be read from left to right, top to bottom, which can be emphasized by numbered arrows.

The player begins with an initial knowledge state in which all configurations of pegs are possible. We have already remarked that the player will only consider single solutions. Hence the solution state which contains all possible single configurations will be the initial state of a Master Mind player. The player then starts with guessing a configuration. This is modelled as a proactive update, as was explained above. We have not indicated how the player comes to his guess.

The result of the update is a solution state in which only this guessed solution is present. The other candidate configurations are, at least for the moment, disregarded and the transition to a solution state with only the guessed configuration has taken place. The proactive update is recorded into memory for backtracking purposes. This is not shown in the figure. Next, it is tested whether the new state, containing the guessed configuration, meets the criterion. The result of the test is a boolean value and a choice what to do next for each value is given. If the test for this guessed solution succeeds this selection embodies his knowledge about the solution. The problem is solved.

If the test fails the agent has to return to all other candidate configurations. The memory associated with proactive updates, is used to return to the previous state. In addition, the negative feedback has resulted in some new information about the solution. First, the player received the
information that the guessed configuration can safely be discarded. Second, the open and closed dots contain information about which colours should, or should not be at certain positions. Hence, other possible solutions can be discarded as well. This line of reasoning is typical of informational updates. The solution state contracts after processing the information given by the feedback, in the sense that it contains less possible solutions. It is then tested whether this new solution state is not empty. If it is and contains no possible solutions then the procedure stops.

If there are still possible solutions to be considered then the procedure repeats itself until a solution has been found. Hence, knowledge states move progressively to more knowledge about solutions and only backtracking to previous states is provided for proactive updates.

The general line of reasoning in this example is a select-and-test method augmented with a capacity for pruning. The feedback the player receives enables him to disregard candidate solutions. If we keep this feedback to a minimum and only inform the player whether his guess was right or wrong (without given him information in the form of open and closed dots) the result would be a select-and-test method Figure 6.2 can therefore be seen as a general knowledge-level description of select-and-test. Note, that the figure is an informal, high-level description of problem solving and is not intended to be complete.
6.3 Tasks and models

The approach we take in this chapter associates a solution to a problem with one or more models. This “model-based” approach is intuitively very suited for tasks in which a set of candidate solutions is given at the start of problem solving. Other problems, like those of tasks in which the solution should be constructed rather than selected from a predefined list, are more difficult to represent.

The distinction between selecting and constructing solutions is reminiscent, but not identical, to the distinction between analytic and synthetic tasks [82].

- **Analytic tasks** are those tasks in which the artefact or “system” one reasons about is given and the goal of the task is to identify the solution(s) which meet some criterion. Examples of analytic tasks are classification and diagnosis.

- In **synthetic tasks** a “system” or artefact has to be constructed from more primitive elements. Planning and assignment are examples of synthetic tasks. In planning one constructs a plan, in assignment problems an assignment.

In analytic tasks like classification and diagnosis one is given a set of candidate solutions, from which one (or several) must be selected which meets a given criterion. In synthetic tasks like planning one constructs a plan as a series of actions to reach a given goal. However, it is sometimes possible (and even feasible) to construct all possible solutions first, and then to select the best “constructed” solution. Hence a constructive problem can sometimes be solved by methods typical for analytical tasks.

In the framework we describe the solution is associated with one or more models, and states represent which possible solutions an agent considers at a given time. Incoming information and pro-active behaviour induce state change and eliminate possibilities. Finally the agent reaches a state in which a criterion is met, and the problem has been solved. This final knowledge state then contains a solution to the problem.

Intuitively, this approach suits tasks, in which the solution is primitive, very well. We will give an example of how several classification criteria can be described later in this chapter. Such an example for synthetic tasks will not be given here. A brief description of an update system for grouping problems is given in chapter 8.

However, model-based approaches have been used for planning. As an example we mention Kautz et al. [52]. They describe a formal model for planning problems based on the notion of satisfiability rather than deduction. As a consequence they identify a solution to a planning problem (a plan) with a model. In addition the representation of domain knowledge is done in such a way that any model of the domain theory corresponds to a given plan. This is similar to the approach we take here and a description of planning in our framework would closely follow the one given by Kautz et al..

Our approach can be seen as an extension or generalisation of such a model-based approach. The differences are two-fold. First, we use update semantics as the underlying formalism. This allows for a dynamic description of the model-finding process. We also make use of the distinction between passive and pro-active updates to characterize state changes. This enables us to characterize state change based on the nature of the information an agent receives.

Second, our representation is not limited to sets of models. As we will see, states can be structured in many ways, and hence one can make use of preferential, or minimal reasoning. As
a consequence our approach is not limited to satisfiability. In fact, satisfiability is one example of the more general notion of epistemic test.

In addition to these formal differences, our motivation also differs from those of Kautz et al. One of the main motivations for the "planning-as-satisfiability approach" is to be able to construct fast and efficient methods for planning problems. Here, however we look merely at the functionality of methods, and their description in terms of knowledge-level constructs. Having said that, the results of computational model-based approaches, as described by Kautz et al. [51] and Khardon et al. [53], can be used to make our approach computationally viable. We will describe this after having explained the underlying formalism of update semantics.

6.4 Logical Dynamics

We will proceed by formalizing the ideas explained above. We do this by first explaining an update system described by Veltman [95]. ¹ The main idea behind this system is that it gives a semantic and dynamic account of reasoning with states and updates. We will extend and adapt this system later for our purposes of describing knowledge level tasks, and now proceed with the treatment of the logical aspects.

In logic the dynamics of information flow has attracted considerable attention. Various formalisms have been developed which can be characterized as evolving around the notion of information rather than that of truth. For an overview and discussion of this subject we refer to van Benthem [91].

The central idea behind update semantics [95] is the following. Whereas according to the classical view, the meaning of a sentence is given by the conditions under which it is true, in the dynamic account the meaning of a sentence is given by the change it induces to an information state.

This view on meaning is more attractive from a ‘cognitive’ perspective. One can think of an agent, possessing a state, dealing with incoming information and making changes accordingly. However, from a formal point of view this change of perspective may not necessarily offer something new. Classical logic can be given a dynamic interpretation quite easily, as will now be shown.

6.4.1 Propositional logic dynamified

Consider a propositional language $L$ with propositional letters $p, q, r \ldots$ and the usual connectives. We associate with $L$ the powerset $W$ of atomic sentences. An information state $\sigma$ is defined as follows:

**Definition 6.1 :** Let $W$ be the powerset of the set $A$ of atomic sentences in a language $L$. An information state $\sigma$ is any subset of $W$. In symbols: $\sigma \subseteq \wp(A)$. The elements of a state, being sets of atomic formulas, are called (possible) worlds.

The definition of an information state may change when one describes richer update systems. The notion of knowledge state will be used as a higher level concept. We use it to express the knowledge an agent has at a given moment in time. The notion of information state will be used in a more rigorous, logical sense.

¹Readers familiar with Veltman’s work can skip all but the latest subsection of this section.
If $\sigma$ is a state and $\Phi$ a sentence we write $'\sigma[\Phi]'$ to denote the update of $\sigma$ with $\Phi$. Here $[\ ]$ is a function which assigns to each sentence $\Phi$ an update operation $[\Phi]$.

For each sentence the update function can be defined as follows $^2$:

- **Atoms**: $\sigma[p] = \sigma \cap \{ w \in W | p \in W \}$
- **Negation**: $\sigma[\neg \Phi] = \sigma \setminus \sigma[\Phi]$
- **Conjunction**: $\sigma[\Phi \land \Psi] = \sigma[\Phi] \cap \sigma[\Psi]$
- **Disjunction**: $\sigma[\Phi \lor \Psi] = \sigma[\Phi] \cup \sigma[\Psi]$

A sentence $\Phi$ is **acceptable** in state $\sigma$ iff $\sigma[\Phi] \neq \emptyset$. A sentence $\Phi$ is **accepted** in state $\sigma$ iff $\sigma[\Phi] = \sigma$. If $\Phi$ is accepted by $\sigma$ we write $\sigma \models \Phi$. When $\Phi$ is accepted in every state, we write $\models \Phi$. An update is **informative** if $\sigma[\omega] \neq \sigma$.

It is not difficult to discern the classical semantics of propositional logic in this formalism. A state is a set, whose elements, or worlds, are sets of atomic propositions. Each world corresponds to a valuation in the following way: atomic propositions occurring in the world are given the truth value ’true’, those which are absent ’false’. Hence, a state can be seen as a collection of valuations or models, and $W$ as the collection of all models of $L$. $\wp(A)$ is denoted by $0$ and is called the **minimal state**. $\emptyset$ is called the **absurd state** and is denoted by $1$. Note that all formulas are accepted in the absurd state, hence the name. Logically, $0$ and $1$ can be thought of as the symbols $\top$ and $\bot$ respectively.

**Example 6.2**: Consider a language $L$ with three propositional atoms $p, q, r$. The powerset $W = \wp(\{p, q, r\})$ is given by $\{\{p, q, r\}, \{p, q\}, \{p, r\}, \{q, r\}, \{p\}, \{q\}, \{r\}, \emptyset\}$. The elements of the set are called worlds, which can be interpreted as models. The set $W$ is itself an information state, called the minimal state $0$, containing all models over $L$.

**Updating the minimal state** $0 = \wp(\{p, q, r\})$ with $p$ gives us a new state which we call $\sigma$:

$$\sigma = 0[p] = \{\{p, q, r\}, \{p, q\}, \{p, r\}, \{q, r\}, \{p\}, \{q\}, \{r\}, \emptyset\}$$

**In this new state** $p$ is accepted ($\sigma \models p$) because $p$ occurs in every world of $\sigma$.

If we start with the powerset $W$ as our initial state than updating sentences to this state is equivalent to adding them to a set of hypotheses. So $W[p] \models p$ is the same as $p \models p$ in classical, propositional logic. And the truth of a sentence $\omega$ is equivalent to $\omega$ being **accepted** in $W$. In this sense the dynamic variant does not offer anything extra compared to the classical, static, truth-functional logic.

Instead of $0[\omega]$ we will sometimes write $\|\omega\|$. Propositional sentences can now be given a static meaning as follows:

- **Atoms**: $\|p\| = \{w \in W | p \in w\}$
- **Negation**: $\|\neg \omega\| = W \setminus \|\omega\|$
- **Conjunction**: $\|\omega \land \nu\| = \|\omega\| \cap \|\nu\|$
- **Disjunction**: $\|\omega \lor \nu\| = \|\omega\| \cup \|\nu\|$

We will say that $\|\omega\|$ is the set of worlds in which the proposition $\omega$ holds. When one thinks of worlds as models (truth assignments of the atomic formula), $\|\omega\|$ is the set of all models which

---

$^2$ $\setminus$ denotes the difference between the sets.
make \( \phi \) true. For example, the static meaning of an atomic formula \( p \) (written as \( \|p\| \)) is equal to the set of worlds in which \( p \) occurs.

Hence, such an update system is no different from a truth-functional propositional logic. However, this parallel between dynamic semantics in terms of states and updates with static, truth-functional semantics changes when the dynamic language is extended. The framework of update semantics facilitates the definition of other operators. For example the language can be enriched with an operator for an 'epistemic test'.

6.4.2 Epistemic tests

The propositional system described above can be extended by introducing an epistemic test. As we remarked above, unlike updates, tests do not lead to state change. Instead updates are done hypothetically to see if the desired result would occur. In fact, acceptance \(|\vdash p|\) is an epistemic test as well.

We enrich the system described above with a test for consistency. The propositional language is extended with an operator \( \lozenge \), \(^3\) so that propositions like \( \lozenge \phi \) can be formed. Informally its meaning is as follows. If an update with \( \phi \) to the state \( \sigma \) does not lead to the absurd state \( 1 \) then the test \( \lozenge \phi \) succeeds, otherwise it fails. Its dynamic semantics can be defined as follows:

\[
\begin{align*}
\sigma[\lozenge \phi] &= \sigma \text{ if } \sigma[\phi] \neq \emptyset \\
\sigma[\lozenge \phi] &= \emptyset \text{ otherwise}
\end{align*}
\]

Hence the epistemic test \( \lozenge \phi \) consists of the hypothetical update of \( \phi \) to a state \( \sigma \). If the result is absurd, then in truth-functional terms \( \phi \) is inconsistent with the information contained in \( \sigma \), otherwise it is consistent with it.

An alternative formulation would be:

\[
\begin{align*}
\sigma[\lozenge \phi] &= \sigma \text{ if there is a non-empty state } \sigma' \text{ such that } \sigma' \subseteq \sigma \text{ and } \sigma[\phi] = \sigma' \\
\sigma[\lozenge \phi] &= \emptyset \text{ otherwise}
\end{align*}
\]

From this it follows that a sentence of the form \( \lozenge \phi \) is accepted in a knowledge state \( \sigma \), \( \sigma \models \lozenge \phi \), iff its update with \( \phi \) would not result in the absurd state \( 1 \).

It is clear that by introducing \( \lozenge \) the parallel between acceptance and the classical notion of truth is disturbed. A sentence like \( \lozenge p \) can be accepted but will not be true in the classical sense. To see this, consider a state where \( \lozenge p \) is accepted, then update it with \( \neg p \). In the resulting state \( \lozenge p \) is no longer accepted. Hence acceptance is no longer monotonic.

Not mentioned by Veltman is the fact that adding this epistemic test for consistency to a propositional update system, results in a system which is equivalent to the modal logic S5. This logic is often used for knowledge representation. For example, in Fagin et al. \([24]\) it is used to represent problem solving in a multi-agent setting.

S5 is defined as the modal system in which all propositional tautologies hold together with the following axioms:

- \( \Box(\phi \rightarrow \psi) \rightarrow (\Box \phi \rightarrow \Box \psi) \)
- \( \Box \phi \rightarrow \phi \)

\(^3\)Veltman uses might instead of \( \lozenge \).
• \( \Diamond \phi \rightarrow \Box \Diamond \phi \)

In addition the following rule, called necessitation, holds: If \( \vdash \phi \) then \( \vdash \Box \phi \).

\( \Box \) is defined as the dual of \( \Diamond \). So \( \Box \phi \) is equivalent to \( \neg \Diamond \neg \phi \). Van Benthem [48] mentions how the update system described above can be seen in relation to S5. Alternatively, one could prove that the S5 axioms and the necessitation rule hold in the above update system.

We regard it as a nice feature of update semantics that by extending one system with an extra operator, one can move from one logic to another quite naturally. Adding an epistemic test for consistency to a propositional update system allows one to go from propositional logic to S5. Note that the semantics of the two systems are very similar. Compare this to the "standard" semantics of propositional logic (truth-values) and S5 (possible worlds), which are much harder to compare.

Further extensions and changes are possible as we will now show.

### 6.4.3 Preferential reasoning

The update system presented here can be further extended. Various non-classical ways of reasoning can be modelled. Veltman in [95] describes a default system in terms of the operators \textit{normally} and \textit{presumably}. Adding such operators also involves a change to the definition of information state.

The idea of incorporating defaults can be put as follows. Sentences like \textit{normally} \( \phi \) can be used to express what normally is the case. In [95] they are called default rules. Suppose that an agent has a set of sentences \( P \) of default rules. Every world in the agents knowledge state could be a model for some of these sentences. Worlds in a state \( \sigma \) can be ordered with respect to the set \( P \) as follows. We describe the order \( \leq \) between worlds \( v \) and \( w \) from \( \sigma \) as follows: \( w \leq v \) if \( w \) is a model for all the sentences from \( P \) which hold in \( v \) and maybe some more.

To explain this in other words: The set \( P \) contains sentences which describe what the agent knows normally to be the case. A world (seen as a truth-functional model) in which every sentence in \( P \) holds represents the situation in which everything is normal to the agent. The world in which no sentence of \( P \) holds represents the situation in which nothing is normal.

Each world can be seen as a description of a possible state of affairs. Some worlds are more normal than others and hence worlds can be ordered along the relation 'more-normal-than' with respect to the set \( P \) of sentences which are to be regarded normally to be the case. A state then becomes an ordered set of worlds.

This means that the notion of information state is extended with a partial ordering on worlds. This ordering can then be used as a preference structure. Updates with sentences of the form \textit{normally} \( \phi \) to a state \( \sigma \) affect the ordering of the worlds in \( \sigma \). A new epistemic test in the form of \textit{presumably} \( \phi \) can now be defined to see whether \( \phi \) holds in the most preferred, or optimal world, being the one highest in the ordering. Other updates work exactly as before and may eliminate worlds, leaving the remaining ordering intact.

Such an example shows that the notion of information state can be changed in the framework of update semantics to allow the description of preference structures and non-monotonic reasoning. Furthermore it allows the description of changing preference structures during reasoning.
### 6.4.4 Formalizing defaults

Consider a propositional language $L$ which we extend to $L'$ with operators $n$ and $p$ such that if $\phi$ is formula in $L$ then $n(\phi)$ and $p(\phi)$ are formulas in $L'$.

By $W$ we denote the powerset of the set of atomic propositions $A$. The elements of $W$ are denoted by $w, v, \ldots$ and are called worlds. An expectation pattern $\varepsilon = \langle W, \leq \rangle$ is a partially ordered set (poset) on $W$. If $w \leq v$ and $v \leq w$ we write $w \equiv v$, for worlds $w, v$.

The use of expectation patterns is as follows. The agent has some knowledge of state of affairs which it expects normally to be the case. This is represented in the form of sentences of the form $n(\phi_1) \ldots n(\phi_n)$. The set of all sentences which the agent considers normally to be the case is denoted by $P$. $\langle w, v \rangle$ is an element of the expectation pattern $\varepsilon$ iff every proposition in $P$ which holds in $v$, also holds in $w$. (We say that the sentence $\phi$ holds in world $w$ if $w \in \|\phi\|$). That is, if $w$ can be interpreted as a model for $\phi$.

Hence all the worlds are partially ordered relative to the set $P$ of sentences which are normally to be the case. The world $w$ such that $w \leq v$ for all $v \in W$, is called a normal world. Hence, $w$ is a world in which everything considered normally to be the case, does actually hold. In the following we assume that at least one such a world exists for states. In truth-functional terms this means that the set of sentences that express what is normal, is consistent.

**An information state for an update system with default rules** can now be defined as a pair $\sigma = \langle \varepsilon, s \rangle$ where $\varepsilon$ is a pattern on $W$ and $s$ captures the knowledge of the agent, $s \subseteq W$. The minimal state is given by $\langle W \times W, W \rangle$, the absurd state by $\langle \{ \langle w, w \rangle \mid w \in W \}, \emptyset \rangle$.

As an example of how this all works consider figure 6.3. The figure contains a graph of a state $\sigma = \langle \varepsilon, s \rangle$ for a language of three atoms. The eight worlds are denoted by $w_0 \ldots w_7$.

![Figure 6.3](image)

A knowledge state with eight ordered worlds. If $w < v$ then there is a path to the left from world $w$ to world $v$. If $w \equiv v$ $w$ and $v$ are placed in the same oval (taken from Veltman [95]).

Worlds are placed in the same oval if they belong to the same equivalence class defined by $\equiv$. If for worlds $w, v$ we have $w < v$, then there is a rightward path from the oval containing $w$ to the one containing $v$. In such a case $w$ is more normal than $v$.

---

$p$ can be read as presumably, $n$ as normally, as Veltman does. However we prefer this more abstract notation. A more abstract reading of these operators is to view $n$ as a default rule. $p$ can be read as the test whether the default holds in a state.
In this situation depicted in figure 6.3 there are two normal worlds: \(w_0\) and \(w_5\). The set \(s\) captures the knowledge the agent has about the world. It is pictured as a triangle and contains \(w_0, w_3\) and \(w_4\). Hence, the agent knows that the other worlds are no longer possible. Since both normal worlds \(w_5\) and \(w_0\) do not belong to \(s\), the agent knows that not everything is normal. However it may have a preference for worlds in \(s\) that are 'as normal as possible'.

Such worlds are called optimal in the following sense: \(w\) is optimal in \((\varepsilon, s)\) iff \(w \in s\) and there is no \(v \in s\) such that \(v < w\). In figure 6.3 both \(w_3\) and \(w_6\) are optimal in the pictured state.

Optimal states play an important role when knowledge is incomplete. The expectation pattern orders worlds along the agent's knowledge of what is normal. The set \(s\) captures the knowledge of what is, and what is not, possible. An optimal world then captures the knowledge of what is to be expected given the knowledge of what is still possible. Optimal worlds will come into play when one considers sentences of the form \(p(\phi)\). Such a sentence is accepted if \(\phi\) is true in all optimal worlds.

Three kinds of updates (or tests) can now be distinguished.

- **Propositional updates**: updates of sentences not containing \(p\) or \(n\). These work exactly as in the propositional update system, and affect \(s\), while leaving the pattern \(\varepsilon\) untouched.

- If \(\phi\) is of the form \(p(\psi)\) then \(\sigma[\phi]\) is given as:
  
  Let \(m\) be the set of optimal worlds in \(\sigma\), then \(\sigma[\phi] = \sigma\) iff \(\sigma[\psi] = m\). Otherwise \(\sigma[\phi] = 1\).

  In words: \(p(\psi)\) is accepted in \(\sigma\) if and only if \(\psi\) holds at all optimal worlds in \(\sigma\).

- If \(\phi\) is of the form \(n(\psi)\) then an update will result in a change of the pattern \(\varepsilon\), on the condition that there is at least one normal world in which \(\psi\) is true. If there is no such normal world \(\sigma[\phi] = 1\).

  Otherwise the pattern \(\varepsilon\) is refined. A pattern \(\varepsilon'\) is a refinement of \(\varepsilon\) if \(\varepsilon' \subseteq \varepsilon\). And a pattern \(\varepsilon\) is a refinement with the proposition \(e, \varepsilon \circ e\) if \(\varepsilon \circ e = \{(w, v)\} \text{ if } w \in e \text{ then } v \in e\).

  The update of \(\sigma[n(\psi)]\) can now be defined as being equal to \(\varepsilon \circ ||\psi||, s\).

Note that an update with a sentence of the form \(p(\phi)\) is actually an epistemic test: it does not change the information state.

As an example of how this update system works, consider figure 6.4. Here four worlds are represented. \(W = \{w_0, w_1, w_2, w_3\}\), and \(w_0 = \{p\}, w_1 = \{q\}, w_2 = \{p, q\}\).

The figure starts with the minimal state given by \((W \times W, W)\). Hence at this point the agent has no knowledge of what is normally the case. Then this state is updated with \(np\). This induces an ordering, and puts worlds containing \(p\) to the left of those which do not contain it. Notice that the knowledge of which worlds the agent considers possible, given by \(s\), has not changed by this update.

It is quite easy to establish that \(p(p)\) holds in this new state. Updating this state with the proposition \(q\) eliminates two worlds \((w_0\) and \(w_1\)) from \(s\). \(w_3\) is both optimal and normal in this state. It can be readily verified that in addition to \(p(p)\), \(p(q)\) now also holds.

An alternative to the last update is shown by the vertical arrow. Here the second state is updated with \(nq\), affecting the ordering of the worlds. \(w_3\) has become the only normal world in

\[\text{Here the proposition } e \text{ is the set of worlds in which the sentence expressed by the proposition holds. So if } \phi \text{ is the sentence, } ||\phi|| \text{ is the proposition expressed by the sentence.} \]
6.4.5 Another epistemic test

The above preferential system was described by Veltman in [95]. We add a new epistemic test to it which will be used later on in the description of classification criteria.

First we look at the test \( p \phi \) which we used above. It can be defined as follows:

\[
\begin{align*}
\text{Let } W_o \text{ be the set of all the optimal worlds in a state } \sigma. \\
p: & \quad \sigma[p(\phi)] = \sigma \text{ iff } W_o \cap ||\phi|| = W_o. \\
& \quad \text{Otherwise } \sigma[p(\phi)] = \emptyset.
\end{align*}
\]

In words: \( p(\phi) \) holds in \( \sigma \) if it holds in the state formed by all optimal worlds in \( \sigma \). Veltman uses this as to express presumably. A sentence presumably \( \phi \) is accepted in a state \( \sigma \) when it holds in all those worlds which are considered normal in \( \sigma \).

It is easy to think of a weaker version of this test. It is a test for consistency in the state formed by all optimal worlds in \( \sigma \).

Here follows its definition:

\[
\begin{align*}
\Diamond p: & \quad \sigma[\Diamond p(\phi)] = \sigma \text{ iff } W_o \cap ||\phi|| \neq \emptyset. \\
& \quad \text{Otherwise } \sigma[\Diamond p(\phi)] = \emptyset.
\end{align*}
\]

Note that \( \Diamond p(\phi) \) can also be defined as \( \neg p(\neg \phi) \). In order to make use of this operator the language \( L' \) has to be extended to allow sentences of the form \( \Diamond p(\phi) \).
6.5 Representing knowledge dynamically

The idea of an information state as it occurs in update semantics will here be used to capture the knowledge an agent has at a given moment. When used in this sense we will use the term knowledge state. If an agent starts with the minimal state 0 as its initial state, this represents the fact that the agent has no knowledge at all, (hence the notion of minimal state). After successive updates, the knowledge of the agent will increase as its state will contract. The state will expand in terms of knowledge. The amount of knowledge and worlds in a state are each others duals.

6.5.1 Models, methods and tableaux

Before we apply the logics to our framework we would like to emphasize the semantic nature of this exploration. Update semantics can not be seen as a computational viable method for reasoning. Building a propositional theorem prover which begins with a powerset of all atoms is not a good idea.

The analysis presented here is not aimed to be a computational analysis, targeted at the development of fast or new PSMs. Instead we provide a functional, knowledge-level, rather than a computational efficient, symbol-level description of task reasoning. We claim that knowledge is much more a semantic than a syntactic notion and therefore a semantic account is worthwhile.

However, there are ways of turning the framework presented here into a computational effective way of solving problems. One approach is to represent information states not as sets of all models (worlds) but to limit this set to a number of characteristic models. This is described in detail by Kautz et. al [51] and Khardon et. al [53]. Kautz et al. describe their approach for propositional Horn theories. Briefly, the idea is as follows: given the set of all models of some theory \( \Sigma \), this set can be represented by a subset of models, called the characteristic models of \( \Sigma \). These characteristic models can be seen as a minimal "basis" for all models of \( \Sigma \), in the sense that one can generate all models by taking the conjunctive closure of the set of characteristic models. (For details we refer to Kautz et. al. [51].) As a result a model-based approach to knowledge intensive reasoning becomes computationally very efficient. For example, in the same paper, Kautz et. al show that abduction can be performed in polynomial time.

Characteristic models can be used in the representation of information states of update semantics. Instead of defining a state as a set of worlds one uses only the characteristic worlds of that state. How to work out all details, particularly for the case of ordered worlds and default rules, remains a interesting topic for further research.

Another approach of making update semantics a computationally efficient way of describing problem solving is to relate it to theorem proving techniques. An interesting option is to use analytic (or "semantic") tableaux. Tableaux have been used in chapter 3. They are particulary interesting because of their 'semantic properties'. Except being used as a classical proof method, tableaux can be used for other purposes, like model finding, minimal entailment [68] and abductive reasoning [62, 3]. Tableaux have the nice property that they combine syntactical and semantic ideas. They can be considered as purely syntactical structures with rules for their manipulation. On the other hand they can be interpreted semantically in the sense that each open branch corresponds to a model of the represented theory.

The propositional update system with tests for consistency and acceptance can very well be translated into a computational more attractive format. We will show how to use analytical
There is a straightforward relation between tableaux and the (non-preferential) propositional information states introduced so far. Let \( \tau(\phi) \) be the tableau for the formula \( \phi \). Two tableaux \( \tau_1 \) and \( \tau_2 \) can be joined into a tree \( \tau_3 \) by adding the root of one of the two trees to all the branches of the other. We use the notation \( \tau_1 + \tau_2 = \tau_3 \) for the operation where the root of \( \tau_2 \) is added to the branches of \( \tau_1 \). We denote the tableau of a knowledge state \( \sigma \) by \( \Upsilon(\sigma) \). Its meaning is as follows:

- The minimal state: \( \Upsilon(W) = \tau(\top) \) (The empty tree).
- The absurd state: \( \Upsilon(\emptyset) = \tau(\bot) \) (Any closed tree).
- Update: \( \Upsilon(\sigma[\phi]) = \Upsilon(\sigma) + \tau(\phi) \).
- Consistency test: \( \Upsilon(\sigma[\Diamond \phi]) \):
  - If \( \Upsilon(\sigma) + \tau(\phi) \) closes then \( \Upsilon(\sigma[\Diamond \phi]) = \tau(\bot) \), else \( \Upsilon(\sigma[\Diamond \phi]) = \Upsilon(\sigma) \).
- Acceptance: \( \sigma \models \phi : \) The tableaux \( \Upsilon(\sigma) + \tau(\neg \phi) \) closes.

The minimal state corresponds to the tableau for \( T \): all possibilities are open and no branches have been closed. The absurd state corresponds to the absence of possibilities, and hence all branches have been closed.

The tableau of an update \( \sigma[\phi] \) is defined recursively. First construct the tableau for \( \sigma \) which can be seen as series of updates of the minimal state. Then add the root of the tableau for \( \phi \).

The consistency and acceptance tests are pretty straightforward. To test a formula for consistency with a theory one checks whether adding the tableau of the formula to the theory does not lead to closure of the resulting tableau. The same is done in terms of states. Acceptance tests whether a formula follows from the theory/state. One adds the negation of the formula to the tableau of the the theory/state and checks for closure.

In order to get from a tableau \( \Upsilon \) of a theory \( \Sigma \) to an information state \( \sigma \) we make use if the fact that each open branch of \( \Upsilon \) represents a model. More specifically an open branch corresponds to one or more worlds. Note that a tableau of a theory only contains those atoms which occur in the theory, whereas worlds refer to all atoms in the language. Therefore there can be more than one world corresponding to an open branch.

To find the worlds corresponding to a branch we look at the positive and negative literals. We then select all worlds which contain all positive literals and do not contain the duals of the negative literals. If we repeat this for every branch we end up with the set of worlds which forms the corresponding state.

**Example 6.3:** Consider a language with three propositional atoms \( p, q, r \). The minimal knowledge state is then formed by taking the powerset \( \wp(\{p, q, r\}) \). The tableau of this state \( \Upsilon(\wp(\{p, q, r\})) \) is the tableau \( T \). Updating the minimal state with \( \neg r \) removes from the powerset all sets in which \( r \) occurs. The associated tableau of this state is the tableau of \( \neg r \) (which is just the leaf \( \neg r \)) added to \( T \) which results in the one leaf tableau \( \neg r \).

Checking consistency of \( r \) in terms of tableaux is done by adding \( r \) to the tableau just built and checking for open branches. In this case consistency obviously fails, since the new tableau

---

6A literal is an atom or its negation. Positive literals are atoms, negative literals their negations. The *dual* of a negative literal is the atom without negation.
closes. Acceptance of \( \neg r \) succeeds since adding \( r \) to the tableau of \( \neg r \) results in closure.

Notice that where reasoning with knowledge states involves the reduction of possibilities after updates, tableaux do just the opposite. One starts with an empty tableau and expands it as updates are being made. This has the obvious advantage that not all models are being represented, whereas knowledge states contain all possible models explicitly.

We will not present tableaux for update systems with preferential operators like \( p \) and \( n \). We will leave this subject for further study. We note that tableaux are frequently used in non-monotonic reasoning and minimal entailment [68].

6.5.2 Knowledge and solution states

We will use, extend and adapt the above formalism to describe knowledge level problem solving. In update semantics an information state was defined as a subset of the powerset of the atoms of the language. We will use the information states of update semantics as a representation of our knowledge states. However, we add some structure to knowledge states which reflects different types of knowledge.

As described above, the knowledge an agent has can be divided into two categories: a static part and a dynamic part. The static knowledge remains invariant during problem solving, while the dynamic part is subject to change. Hence, when we talk about knowledge states we are mostly concerned with the dynamic part, taking the static part for granted.

In the representation of tasks, domain knowledge, common-sense knowledge and some assumptions do usually make up the static part of a knowledge state. When new information becomes available the agent will be able to make new deductions. Of course, these new deductions are made possible by applying the static knowledge to the new information.

Dynamic knowledge is subject to change. However, not every dynamic sentence of a knowledge state is relevant to the problem solving process. As new information comes in, many useless sentences can be derived. All these qualify in principle as dynamic knowledge.

This problem of which information is relevant can be solved pragmatically. Only knowledge that is related to the possible solutions of the problem is important. We therefore often equate dynamic knowledge with the solution space.

To see how this affects the representation of knowledge states consider the domain knowledge of some problem, formulated in a language. The minimal information state of this language is formed by taking the powerset of the atoms of the language. Next we update this state with every sentence representing domain knowledge. The resulting state then contains the domain knowledge.

From a semantic point of view the worlds of this state represents all the models of the domain knowledge. Since the domain knowledge will remain invariant a model of the solution to the problem will be among the worlds in this state.

Since we are interested mainly in knowledge about possible solutions we want to be able to talk about the candidate solutions which are present in the state which contains the domain knowledge. Therefore we abstract a knowledge state by ignoring all information that is not about possible solutions.

Knowledge states can be abstracted to solution spaces (or solution states as we will call them) in the following way:
Definition 6.2: Let $S$ be the set of possible solutions. A solution state $\sigma_s$ is a subset of the powerset of solutions $\wp(S)$.

A solution state $\sigma_s$ is a representative of state $\sigma$ iff the following holds:

$$\sigma_s = \{ w_s \mid w_s \subseteq w, \forall w \in \sigma, w_s \in \wp(S) \}.$$  

Hence, a solution state is a representative of a knowledge state $\sigma$ if it can be produced by removing all elements not in $S$ from its worlds. Note that several knowledge states can be represented by a single solution state.

The role of the notion of solution state is to see which possible solutions are considered by the agent with respect to a given state. The abstraction allows us to hide details about knowledge and updates which do not affect the solution space.

One can describe two kinds of updates in terms of solution states. The first are updates with sentences of which the atoms are all candidate solutions. For these it is immaterial whether one updates the underlying knowledge state or the solution state itself, since such sentences will only affect the candidate solutions.

Other updates that can be described in terms of solution states instead of knowledge states are those in which one wishes to hide the details of all changes of knowledge, except those related to the possible solutions. This use is pragmatically motivated. For example, if a knowledge state contains the knowledge that a blackbird is black and we update this state with the observation of a white bird, then a description of the update in terms of solution state will just show the elimination of blackbird as a candidate solution. The logical justification of this elimination is of course given by the description of the update to the underlying knowledge state. The solution state description presents a less detailed and more abstract account in which trivial steps do not have to be spelled out. Hence, updates to solution states can be used where one assumes that their effect needs no further elaborate description.

It is not always necessary to think of solution states as subsets of the powerset of all solutions. In many problems one looks for a single solution to a problem. In diagnosis this is called the single fault assumption and similarly in classification one usually assumes that a single class should account for the observations. In such cases this information reduces solution sets immediately to singleton solution states (those containing one solution only). We regard such single solution assumption as part of the invariant part of the agents knowledge.

One should be careful how to interpret solution states. Every world of such a state contains zero or more solutions, and can be said to express that it is possible that its elements are solutions. Every world expresses a possible state of affairs. Whether an element really is a solution depends on the criterion of the problem at hand. For example: the criterion of weak explanation expresses that a class is a solution if and only if it occurs in some world. The criterion of strong classification demands that solutions should be necessary, or occur in all worlds of the solution state.

6.5.3 Updates

Updates give rise to changes in knowledge states. As we described above, from a knowledge level perspective there are two kind of updates: reactive and proactive updates.

Formally speaking there is no difference between reactive and proactive updates. The difference lies in the interpretation of what an update means. From a knowledge level perspective reactive updates reflect state changes resulting from processing new information coming from the environment. For example, in diagnosis and classification tasks observations play a key role. The
agent reacts to new information contained in these observations by updating it into his current
knowledge state.

In more detail an observation can be represented as a formula \( \phi \) which is updated to the agent’s
current knowledge state. Depending on the content of the observation the knowledge of the agent
then changes state.

On the other hand, an agent may also make decisions, guesses or explore possibilities within
his own knowledge state. These decisions are represented as proactive updates which reduce the
knowledge state to fewer worlds. As an example consider the Master Mind game where the player
makes a guess. We can describe this as a state change of the players solution state in such a way
that the new state only contains this guess.

In detail this can be described as follows. Let \( \sigma_s \) be the player’s solution state, containing
those Master Mind configurations which he judges to be possible solutions. If he makes a guess \( g \)
then the result is the new state \( \sigma'_s \) which only contains \( g \) as possible solution. We can describe the
proactive selection of one of the possible solutions from a solution state as the result of the update
with that choice. Continuing the example, \( \sigma'_s \) is the result of updating \( \sigma_s \) with \( g \).

Hence, "selecting some worlds from a state" and "proactively updating" are two sides of the
same coin. To select from a state \( \sigma \) (in a propositional update system) all worlds which contain
\( p \), one simply updates \( \sigma \) with \( p \). In this way every selection of worlds from \( \sigma \) corresponds to a
sentence which after updating to \( \sigma \) results in a state containing the desired selection.

A guess is an example of a proactive update. In the Master Mind game it is a selection of
one configuration. This selection is proactive because the agent himself takes the initiative for it.
However, it should be clear what choices the agent can make and which not. Proactive updates
should reflect rational capabilities of an agent which are strongly related to the structure of the
knowledge state. In the case of the Master Mind player the choice of \( g \) was a random selection of
the solutions contained in \( \sigma_s \). The player then could have chosen another configuration but only
among those which were among the candidate solutions in \( \sigma_s \).

A proactive update then reflects a capability of the agent to use the structure of its knowledge
state to initiate state change. These capabilities can be described in terms of pre- and postcondi-
tions which involve quantifications over epistemic tests.

The precondition describes in terms of epistemic tests whether a proactive update is applicable
or not. The postcondition is a description of the state resulting after the proactive update.

Consider a state where some solutions are consistent and others have been ruled out, as
in the case of the Master Mind player. In addition assume that solutions exclude each other.
Suppose the agent has the capability to proactively update its solution state \( \sigma_s \) to any sub-state \( \sigma' \)
consisting only of one of these solutions. This capability can be expressed in terms of pre- and
postconditions as follows.

\[
\begin{align*}
\text{Precondition} & : \quad \text{There is a solution } s \text{ such that } \Diamond s \text{ holds in the solution state } \sigma_s. \\
\text{Postcondition} & : \quad \sigma'_s = \sigma_s[s].
\end{align*}
\]

Here \( \sigma_s \) is the solution state before the update and \( \sigma'_s \) the solution state immediately after.
Another example involves an agent who can produce a solution which is consistent with all
optimal worlds of its state \( \sigma_s \).

\[\begin{align*}
\text{Precondition} & : \quad \text{There is a solution } s \text{ such that } \Diamond s \text{ holds in the solution state } \sigma_s. \\
\text{Postcondition} & : \quad \sigma'_s = \sigma_s[s].
\end{align*}\]

\[\text{By a substate } \sigma' \text{ of a state } \sigma \text{ we mean a state which consists of a subset of worlds of } \sigma \text{ and preserve the order of } \sigma \text{ if there is one.}\]
Precondition: There is a solution \( s \) such that \( \Diamond p(s) \) holds in the state \( \sigma_s \).

Postcondition: \( \sigma'_s = \sigma_s[s] \).

There is a choice of how to represent the capabilities of an agent's knowledge. One could opt for a structured knowledge state with few (or no) proactive updates. Alternatively, one could keep the knowledge state simple and put more emphasis on the definition of the proactive updates.

For example, suppose a Master Mind player when making a guess has a strong preference for making guesses in which red pegs occur. We consider two ways of representing this. First, we could order the worlds in the solution state, giving preference to those containing red-pegged configurations. The proactive update can then be described as the (random) selection of one of the preferred worlds.

Another solution is to keep the knowledge state unstructured and impose no orderings on the different solution. Instead we define the precondition for a proactive update as a test for a solution which contains red pegs. In addition we must also provide a proactive update for the eventuality that there are no more red pegged solutions in the current state. This can be described in the form of a simple random selection of a solution, since the player had no preferences when there are no red-pegged candidate solutions to be chosen.

Another capability that we want to assign to a knowledge-level agent is related to reactive rather than proactive updates. Reactive updates deal with processing incoming information. However, we want to enable the agent to order, to ignore and to select the incoming information. For example, in classification problems many observations are being made. The agent must have the capability to process these observations in a certain order. In addition it must also be able to ignore some observations and select others as more important.

Again, a preferential representation, by means of ordered knowledge states, can be used in such cases. However, there is one principle, which is implicit in the use of information states, which can be used to describe the selection of new information. In general an agent will have a preference for a high information value. A sentence \( \phi \) has a higher information value than the sentence \( \psi \), relative to an information state \( \sigma \), if the update of \( \phi \) to \( \sigma \) leads to a smaller state than the update of \( \psi \) to \( \sigma \).

In diagnosis one observation might lead to a bigger reduction of possible solutions than the other and hence has a higher information value. This principle of information value can be seen as an introspective capability of the agent. In chapter 2 it was used to order the observation attributes in rock classification. The system would ask for the value of the attribute with the highest average information value.

Finally, as proactive updates are concerned with making choices, there is always the possibility that a wrong choice will be made. Therefore the agent needs some means of backtracking to previous choice points.

6.5.4 Memory

Proactive updates are strongly linked to a memory. An agent is always allowed to retract its proactive updates. Again the master mind game may serve as an example. When a guess fails to be a solution, the player has to return to its former state. Therefore, to describe proactive updates a knowledge state must be extended with a memory.

This can be done by giving the agent complete recall [24]. We will do this as follows: The agent remembers the initial state and keeps a stack of all updated formulas. Any previous state can
then be retrieved by removing the unwanted updates from the stack and updating the initial state with the formulas still on the stack.

**Definition 6.3**: A knowledge state with complete recall is a tuple $(\sigma, M)$, where $\sigma$ is a knowledge state as defined before, and $M$ a memory.

A memory $M$ is a tuple $(\sigma_i, L)$ where $\sigma_i$ is the initial knowledge state and $L$ a list of sentences.

Two operations, adding and deleting are defined on the memory. Adding involves inserting a sentence to the end of the list $L$. As we will use memory to trace the updates made, we have to record both proactively and reactively updated sentences.

To return to a previous state $\sigma_p$, we look for the sentence $\phi$ in the memory that was proactively updated to $\sigma_p$. We then remove $\phi$ from the memory list and all sentences that come afterwards. The initial state, kept in memory is then updated with all sentences in the new memory list. In addition, it may be necessary to update the state with the negation of $\phi$ in order to prevent it from being chosen again.

We will not elaborate further on the formal aspects of the use of memory in the representation of knowledge states. Here, we would like to draw attention to the fact that a memory is needed for proactive behaviour and that its use can be described in a straightforward manner.

### 6.5.5 Criteria and requirements

Knowledge states change when updates are made. State change ends when a solution has been found. We represent this as follows: A state can be considered to be a final state when it accepts a given epistemic test. A knowledge state is final when all the candidate solutions in its associated solution state meet the *criterion* of the problem. A criterion describes when a candidate solution is regarded a solution to the problem.

For example, in the Master Mind game the criterion is given by three closed dots, indicating that all pegs have the right colours and are on the right position. Acceptance of this criterion in a solution state means that all solutions of the state are solutions of the problem. When more than one solution is required, quantification over the criterion becomes necessary.

Proactive updates are choices which are made in an attempt to reach a state which fulfills the criterion. Hence, the agent’s proactive behaviour is *goal-oriented*. The criterion can be seen as the specification of a goal state, very much like a postcondition in the specification of programs.

We also use the notion of *requirements* to specify certain aspects of a solution. A requirement can thus be seen as a partial specification of the criterion. They can be useful in the representation of problems where the solution has to be constructed.

For example, in assignment problems one is looking for an assignment which satisfies certain constraints and is maximal with respect to a number of given preferences. Each constraint and preference can be seen as a different requirement, and the criterion is the expression which describes that an an assignment is a solution only if it satisfies all constraints, and is most preferred. Like the criterion, requirements can be represented as epistemic tests and the agent’s pro-active behaviour is aimed at reaching a state such that all these tests succeed.

In general, the framework we describe, stresses a distinction between knowledge about the solution and the specification of a solution. When solving the problem the two gradually move towards each other, and finally meet.
6.6 Describing problem solving

Problem solving can be described as a series of updates (both reactive and proactive) and epistemic tests, together with some flow of control. An agent starts problem solving with an initial solution state, containing all possible solutions and by successive state changes ends in a state in which the criterion of the problem is accepted. In the description of a problem solving method for the problem one can use the criterion to come up with a method. We will give some examples of how this works in the next section. A criterion is an epistemic test and may give hints how to reach a state which satisfies it.

6.6.1 Example: three classification criteria

In order to illustrate the notions explained above for a knowledge intensive task, we will describe three types of classification problems. It will be shown how an update system can be tailored to meet the requirements of each classification criterion. This will enable one to make systematic comparisons between them.

For each problem a simple method will be given. These methods will emerge quite naturally from the description of the criterion in an update system. Since updates and tests already provide some basic problem solving behaviour one often only needs to identify some iterations. For these proactive updates can be used. It will be shown that the nature of these proactive updates will differ among criteria.

Preliminaries

The goal of classification is to identify an object which is described by observations, as belonging to a certain class. The task ontology describes the vocabulary in which classification problems can be formulated. A detailed description of this vocabulary was given in chapter 4.

Next, we choose a simple propositional representation. Observations are atomic propositions of the form attribute=value. The knowledge which expresses that an attribute can have only one value is left implicit. The same representation for classification was used in chapter 3.

Class definitions are sentences in which a class implies conjunctions of attributes and value pairs. As an example consider the following class definition:

granite → grainsize=large ∧ origin=plutonic

Observations will be represented as attribute-value pairs, for example grainsize=small. The agent's domain knowledge consists of all class definitions. This knowledge will remain invariant during problem solving and is therefore a static part of the agents knowledge state. The dynamic part of the agent's knowledge state, its solution space, is built from the set of classes.

If \( C \) is the set of all classes then its powerset \( \wp(C) \) generates all possible sets of classes. As in update semantics these sets will serve as models. The dynamic knowledge forms the variant part of the knowledge state. The solution space will be used to keep track of changes over states.

The minimal knowledge state is given by the powerset of all atomic sentences of the language. This has to be updated with the domain theory, to arrive at the initial state. Let \( DT \) be the domain theory and \( W \) the minimal knowledge state. Then the state which captures the domain theory is given by \( W[DT] \).
The solution state of the minimal state \( W \) is given by \( \wp(C) \). If the domain theory \( DT \) contains knowledge which excludes certain combinations of solutions then these are not part of the solution state associated with \( W[DT] \).

As an example, consider a domain theory \( DT \) with three very simple class definitions in the domain of igneous rocks.

- granite \( \rightarrow \) grainsize=large \& origin=plutonic
- basalt \( \rightarrow \) grainsize=small \& origin=volcanic
- diorite \( \rightarrow \) grainsize=medium \& origin=plutonic

These are part of the static part of the agent’s knowledge state. The state also contains additional (static) knowledge expressing that identical attributes with different values are contradictory.

The dynamic knowledge is formed by possible solutions, and is represented as a solution state. The type of solutions is a class. In this example there are only three: granite, basalt and diorite which we abbreviate as \( g, b, \) and \( d \). The minimal solution state, associated with the minimal state \( W \), is simply the powerset of all classes:

\[
\wp\{g, b, d\} = \{\{g, b, d\}, \{g, b\}, \{g, d\}, \{b, d\}, \{g\}, \{b\}, \{d\}, \emptyset\}.
\]

The world \( \{g, b, d\} \) expresses the possibility that granite, basalt and diorite are solutions simultaneously. In this example the domain theory, together with the constraint that identical attributes may have no different values, excludes some of these worlds.

The solution state with respect to the the initial knowledge state \( \sigma = W[DT] \) then becomes:

\[
\sigma_s = \{\{g\}, \{b\}, \{d\}, \emptyset\}
\]

In words this solution state tells us that either granite, basalt or diorite may form a solution, or that there is none. Observations are represented like the av-pairs which occur in the class definitions. Every observation enters the knowledge state as an update. Consider the observation origin=plutonic. As we update the solution state \( \sigma_s \) with this observation, we get \( \sigma_s[\text{origin=plutonic}] = \{\{g\}, \{d\}, \emptyset\} \).

The update has eliminated one possible world, being \( \{\text{basalt}\} \). It is eliminated because origin=volcanic, occurring in the class definition, is contradicting the observation.

**Weak classification**

The criterion of weak classification can be stated as: a class is a solution if and only if it is consistent with the observations.

A class is *consistent* with an attribute when either one of the following two options hold:

1. The attribute as given by the observation does not have a value in the class definition.
2. The attribute value pair matches that of the class definition.

In order to deal with the first option an ‘undefined’ value has to be given to attributes which do not have a value in classes. The reason for introducing such a value has to do with the maximal consistency of the worlds that form a knowledge state: absence of an atomic proposition in a world is interpreted as its negation.

There are several ways to deal with undefined attributes. One is to incorporate it into the domain knowledge and state that its occurrence implies the disjunction of all other values of the attribute.
Another way is to check for undefined attributes in the method itself. When we update an observation to a knowledge state we lose the classes which are undefined for the attribute in the observation. In order to prevent this we update with the disjunction of the observation and its undefined variant. Notice that this mirrors the two options listed above.

Let \( a=v \) be an observation, \( c \) a class and \( \sigma \) the knowledge state containing the class definitions. Below we state three equivalent formulations of the criterion for weak classification:

\[
\begin{align*}
\sigma[a=v \lor a=\text{undefined}][c] \neq \emptyset &\iff (1) \\
\sigma \models (c \land (a=v \lor a=\text{undefined})) &\iff (2) \\
\sigma[a=v \lor a=\text{undefined}] \models \lozenge c &\iff (3)
\end{align*}
\]

The first formulation (1) states that a class \( c \) is a solution if we update \( \sigma \) with the observation (in disjunction with the undefined value) followed by an update of \( c \) then the result should not lead to the absurd state. The second formulation (2) puts this in the form of an epistemic test \( \lozenge \). The test is accepted (succeeds) when the conjunction of \( c \) and the observation passes the test with respect to \( \sigma \). The final equivalent way of putting this (3) chooses to update the observation and test the class \( c \) for consistency.

Remember that solution states contain possible worlds of classes. A class (an atomic proposition) is consistent with respect to a state iff it is in at least one world. Hence, after having updated the observations the union of all worlds of the resulting state contains all classes which are consistent.

Formulation (1) suggest to update the observation (with the disjunction of an undefined alternative) followed by a proactive update of a class. When this last update succeeds the class selected in the proactive update is a solution. The second formulation (2) does not mention updates. It defines the criterion entirely as a test for consistency. The third (3) suggests to update the observation and test a particular class.

Observations are seen here as providing information to the agent about the outside world and therefore they lead to reactive updates. Hence, when choosing one of the three formulations we prefer formulations (1) and (3) rather than (2), because in (2) the observations are not updated. In addition, we let the agent select a class which it will test for being a solution proactively. Formulation (3) expresses this most clearly.

The three formulations of the criterion are logically equivalent but can also be interpreted as having a distinct procedural meaning. One can look at such formulations as a high-level, initial description of a method. Viewing formulation (3) as a high-level specification of a method, it turns out to be a variant of the select-and-test method, presented earlier in the Master Mind example. Here the test part takes the shape of a test for consistency of the selected class.

However, note that no matter which class is selected proactively, all will pass the test. Since every proactive update is consistent the test becomes redundant. Observations will reduce the solution state, removing inconsistent classes with every update. Hence, weak classification can be seen as a purely informational or reactive method, where no decisions have to be made by the agent. When only one solution must be produced the agent may choose one proactively. Still, the criterion of weak classification allows any choice from the solutions found so far.

Note, that in the depiction of the method in figure 6.5 a few assumptions are made. First, it is assumed that updating the observations never leads to the absurd state - i.e. a contradiction. This
is a fair assumption and should be part of a classification task ontology. A second assumption is that after updating the observations a state is reached with at least one candidate solution. If this assumption is dropped the method should halt whenever this occurs.

### 6.6.2 Weak classification with preferred solutions

The criterion of weak classification defines a solution to a classification problem as a class which is consistent with the observations. The problem can be strengthened somewhat by adding knowledge of preferred solutions in combination with observations. This knowledge will be of the form: *if* obs *is observed then class will be preferred.*

We will show how this can be modelled in the update system just described. We consider again the example knowledge base of class definitions:

\[
\text{granite} \rightarrow \text{grainsize}=\text{large} \wedge \text{origin}=\text{plutonic} \\
\text{basalt} \rightarrow \text{grainsize}=\text{small} \wedge \text{origin}=\text{volcanic} \\
\text{diorite} \rightarrow \text{grainsize}=\text{medium} \wedge \text{origin}=\text{plutonic}
\]

In addition we add knowledge about preferred solutions. We like to express that when some observations are made some classes are preferred to others. For example: if the origin is plutonic we prefer granite to diorite (and other plutonic rocks). This can be represented by using the \(n\) operator from section 6.4.3. The resulting expression then becomes: \(n(\text{origin}=\text{plutonic} \rightarrow \text{granite})\).

To see what the knowledge states look like for this type of classification, consider the knowledge state containing the class definitions and this one sentence.

\(\text{origin}=\text{plutonic} \rightarrow \text{granite}\) is equivalent to \(\neg \text{origin}=\text{plutonic} \lor \text{granite}\). Hence all worlds containing granite or lacking origin=plutonic are optimal in the new state. When no observations are made we want all classes to be equally preferred. When origin=plutonic is observed and updated to the state we want granite to be the most preferred class.

Logically, as a result of the update with origin=plutonic, all optimal worlds of the new state will contain granite. Notice also that when origin=volcanic is observed (and updated) instead, then by using the class definitions all worlds containing granite are removed from the knowledge state, just as in weak classification. This is because the same attributes with different values are defined as being inconsistent.

The most preferred class in a given state \(\sigma\) can be described as the one which occurs in all optimal worlds of \(\sigma\). This can be tested by making use of the operator \(p\). Remember that the test \(p(\phi)\) succeeds with respect to a state \(\sigma\) when \(\phi\) occurs in all optimal worlds in \(\sigma\). If after updating
the knowledge state with the observations $p(c)$ is accepted for some class $c$ then this class is both consistent and most preferred.

The criterion for classification with preferred solutions can then be put as follows:

\[
\sigma[a=v \lor a=\text{undefined}] \models p(c)
\]

or equivalently:

\[
\sigma[a=v \lor a=\text{undefined}][p(c)] = \sigma
\]

From this criterion we can again formulate a method. A method description is given in figure 6.6. It starts off similar to the one for weak classification, updating observations as they are made. However, notice that here the worlds in each knowledge state are ordered by the preferential knowledge and that we are working in a different update system. Updating observations removes some solutions, exactly like in weak classification. Next, a proactive update is made to a state containing all optimal worlds. This reflects the part in the criterion which mentions the $p$ operator.

This update is labelled proactive in figure 6.6. Selecting all optimal worlds is not an indeterministic operation in the logical sense but what counts as an optimal world is a non-monotonic in the following sense: if $w_1 \ldots w_n$ are optimal worlds of $\sigma$ then they are not necessarily optimal in a state $\sigma' = \sigma[\phi]$ where $\sigma[\phi]$ is an informative update.

For example, consider an agent making all observations first and then proactively selecting all optimal worlds of the current state. For example, in the case of medical diagnosis a physician may select a number of most preferred diagnoses on the basis of a number of observations. When new information comes in after this selection has been made, the agent has to backtrack to the previous state, update the new information and select the optimal worlds again. This possibility of backtracking is not shown in the figure. It is assumed here that all observations have been made.

A class is a solution if it occurs in all worlds of this new state. The next move is then to choose a single class from a world and use it as a proactive update. The new state contains a single
solution class and satisfies the criterion.

6.6.3 Classification with inconsistent observations

A third classification criterion which will be described here, deals with handling inconsistent observations with respect to class definitions. In classification problems it frequently occurs that no class is consistent with all observations. In such situations it could be desirable to return the class which has the least number of inconsistent attributes relative to the observations made.

Several variations on such criteria exist. These are described in chapter 4. For example, one could demand that some av-pairs should always be consistent whereas others are allowed to be inconsistent with respect to the observations. We will not go into the details of such alternative criteria here. However, we would like to make clear that they can be described in an update system very similar to the one explained above.

The first point to note is that the criterion which describes the class with the least inconsistent attributes as the solution, defines another preference structure. One can partially order all classes in such a way that classes with more inconsistent attributes are more minimal than those which have less. Finding a solution means to return the minimal element in this preferential ordering.

The second point is that this preference structure is dynamic and changes as more observations are updated to the state. This is different from the preferential structure in the previous classification criterion we discussed. There the preference structure was given by preferences in the domain knowledge and remained static during problem solving.

A crude method for finding a class which has the least inconsistent attributes can be described as follows:
If $Obs = obs_1 \ldots obs_n$ are the observations than first a class $c$ is looked for such that $\{c\} \cup \{obs_1 \ldots obs_n\}$ is consistent. If there is one, $c$ is a solution. If not it is checked whether there is a $c$ for which $\{c\} \cup Obs \sim \{obs_i\}$ (for some observation $obs \ 0 < i \leq n$) is consistent. If still a solution has not been found $\{c\} \cup Obs \sim \{obs_i, obs_j\} (0 < i, j \leq n \ and \ i \neq j)$ etc.

As we have seen above, in update semantics one can describe a system in which the worlds are ordered. The update system with the operators $\Diamond p, n$ and $n$ can be used to describe this method.

To do this we represent the class definitions simply as implications. When an observation is made it is updated to the knowledge state. However when $obs$ is an observation and $\sigma$ a state then we update $obs$ to $\sigma$ as $\sigma [n(obs)]$. The $n$-operator results in an ordering of worlds within the state such that those worlds in which $obs$ occurs are preferred to those which do not contain $obs$. A class is consistent with all observations made in this manner if it is consistent in the state formed by all optimal worlds (which are also normal worlds at this stage).

If there is no such class, the state should be updated in such a way that all optimal worlds are removed from the state. Again it is checked whether a class is consistent in the optimal worlds. If not the process repeats itself.

As an example consider again the definitions of igneous rocks:
- granite $\rightarrow$ grainsize=large $\land$ origin=plutonic
- basalt $\rightarrow$ grainsize=small $\land$ origin=volcanic
- diorite $\rightarrow$ grainsize=medium $\land$ origin=plutonic

If $n(origin=plutonic)$ is updated then a preference for worlds containing origin=plutonic is realized. This does not mean that for example, granite is true in all optimal worlds. However it
does mean that are some optimal worlds in which granite occurs. The same is true for diorite but not for basalt.

To test whether a class occurs in an optimal worlds one can make use of the operator $\diamond p(c)$, defined in section 6.4.3. Now, suppose the observation $\text{grainsize=tiny}$ is made. This is updated as $n(\text{grainsize=tiny})$. At this stage none of the three rocks occur in any of the optimal worlds.

For example, granite implies $\text{grainsize=large}$ which is contradictory to $\text{grainsize=tiny}$ which is true in all optimal worlds. However, both granite and diorite are still preferable to basalt since they are consistent in worlds higher up in the ordering.

The criterion for this problem variation can be expressed by using the $p$ operator which is used to test whether classes occur in some optimal worlds. $\sigma[a=v \lor a=\text{undefined}] \models \diamond p(c)$ or equivalently:

$$\sigma[a=v \lor a=\text{undefined}] [p(c)] \neq \emptyset$$

Note, that this is criterion is similar to the previous classification criterion: classification with preferred solutions. Where we used $p$ previously we now use the weaker $\diamond p$. In words: instead of checking whether a class occurs in all optimal worlds we now look for a class which occurs in some optimal worlds.

The method is depicted in figure 6.7. The knowledge state gets ordered by observations which are all prefixed with the $n$ operator. The result is an ordered state. Then it is checked whether there are classes such that $\diamond p(c)$ holds. This is shown as a test with as input the criterion. If this test succeeds then state is proactively updated such that the solution is produced.

If no class is present in an optimal world then the test fails and backtracking takes place. Optimal worlds are now removed and the same procedure is repeated for the next optimal worlds etc. until a solution has been found.

---

**Figure 6.7**
Classification for classes with least consistent attributes.
6.7 Classification criteria compared

The last three simple examples are meant to show that problem solving behaviour follows quite naturally from the specification of the task in terms of the vocabulary of the framework. It also allows for a systematic comparison between these tasks, as we will now show.

Weak classification can be described in a intuitive manner by a propositional update system with an epistemic test for consistency. The static domain knowledge can be represented as is common in classification systems. The representation of a knowledge state consists of worlds without preferential structure. The criterion of weak classification is formed by the epistemic test for consistency, and updates take the role of observations.

Solving a problem of weak classification can be described without any reference to proactive behaviour of the agent. All that is needed is an update of observations to the agent’s knowledge state. This alone will prune inconsistent classes from the solution state. Stating that weak classification can be described purely reactive also means that from a knowledge level perspective the task is purely deterministic.

Extending weak classification with knowledge about preferential solutions changes the representation of the static domain knowledge. The logically machinery changes as well and we move to another update system. We have shown how the $\mathbf{p}$ and $\mathbf{n}$ operators were used in the representation of preferential domain knowledge.

This preferential structure remains static during problem solving. Observations are represented as propositional updates, exactly like in weak classification. Their role is again the pruning of classes from the solution states. The criterion is now formed by the epistemic test for acceptance in optimal worlds, $\mathbf{p}$.

This system is clearly different from the one used for the normal, weak variant. Its main characterizing feature is the use of a preference ordering. This leads to changes in domain knowledge and criterion.

The third criterion for classification: preference for classes with least inconsistent attributes, also makes use of a preference structure. Hence, the knowledge state is ordered here as well. But no adaptations to the static domain knowledge are needed here, as compared to weak classification. The updates of observations take a different role and are no longer propositional. The criterion is described by the epistemic test $\diamond \mathbf{p}$.

The two last problem variations differ in the representation of the knowledge state. In "classification with preferred solutions" the preference structure is static and part of the domain knowledge. In "classification with inconsistent observations", the preference structure is given by means of observations and changes dynamically.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Static knowledge</th>
<th>Knowledge state</th>
<th>Updates</th>
<th>Criterion</th>
</tr>
</thead>
<tbody>
<tr>
<td>WC</td>
<td>Class defs.</td>
<td>Propositional</td>
<td>Obs - atoms</td>
<td>$\diamond (c)$</td>
</tr>
<tr>
<td>WCP</td>
<td>Class defs. + $n$-prefs.</td>
<td>Statically ordered</td>
<td>Obs - atoms</td>
<td>$\mathbf{p}(c)$</td>
</tr>
<tr>
<td>WC1</td>
<td>Class defs</td>
<td>Dynamically ordered</td>
<td>Obs - $n(\alpha)$</td>
<td>$\diamond \mathbf{p}(c)$</td>
</tr>
</tbody>
</table>

Table 6.2
A comparison of the example classification problem descriptions.

Table 6.2 summarizes a comparison between the three classification variations. The description of these examples allows one to compare different task configurations in a systematic way.
Each may require a different representation of domain knowledge, observations or criterion. And each may require a particular logic to express it.

In table 6.2 weak classification is abbreviated as WC. The static part of the knowledge states is formed by the definition of the classes. The formal representation of the worlds of a knowledge state is by means of sets. There is no relation ordering these worlds and the underlying update system is propositional. The updates are observations and are atomic propositions. Finally the criterion whether a class c is a solution is given by the epistemic test \( \Diamond(c) \) for consistency.

In this way one can compare problem variations in a systematic way. Problem descriptions vary in the structure of the domain knowledge, the nature of the updates and knowledge states and of course the criterion. By modifying one of these "parameters" one can describe other problem variations quite easily. For example, one can provide "strong" versions of the above criteria as follows. For weak classification one has to replace the test \( \Diamond c \) with \( \Box c \), which means that c should be an explanation for the observations. To see this remember that acceptance of \( \Diamond c \) means that c is present in all worlds, whereas acceptance of \( \Box c \) means that c occurs in some worlds. This difference can be used to express weak (consistent with the observation) and strong (implying the observations) classification.

In the third criteria a similar change is needed, replace \( \Diamond p(c) \) with \( p(c) \). The second criterion involves a change in the static domain knowledge. In fact, one has to make sure that the preferential knowledge of a class c is expressed as:

\[
n(a_1=v_1 \land \ldots \land a_i=v_i \rightarrow c)
\]

provided that the class definition of c is \( c \rightarrow a_1=v_1 \land \ldots \land a_i=v_i \). This "completion" is necessary to make sure that c will imply all observations when selected as a solution. This is required for "strong classification" criteria, as explained in chapter 4.

### 6.8 Discussion

The framework presented here enables one to describe tasks and problem solving in terms of what an agent knows about problem solving. We view knowledge as a semantic notion and have chosen a semantic framework for its representation. For a description of the logical machinery we have made heavy use of Veltman's update semantics and adapted it in order to be able to account for problem solving behaviour of a knowledge level agent. We introduced such knowledge level constructs as proactive updates and memory.

The semantic point of view allows us to abstract to knowledge level descriptions. To summarize the approach we have taken, consider the following template of actions. First, when given a problem we identify the type of a solution. For classification this is a class. (Its definition is part of the task ontology.) Next we identify the static domain knowledge, the updates and the criterion of the problem.

We have shown how the specification of classification problems can lead to a description of problem solving behaviour in terms of changing knowledge states. The nature of the static domain knowledge, the formal structure of the knowledge states, the nature of updates and the criterion can be seen as key properties in the description of a problem. With a slight abuse of vocabulary we refer to such a description as "the semantics of the problem".

Different requirements on a solution will lead to different tests, updates and different structured knowledge states. We have illustrated this by looking at three classification criteria. Other classification criteria (see chapter 4) can be described in a similar way.
The framework described takes a model-based approach. In practice this means that we begin by making a knowledge-level analysis of the problem, identifying the static and dynamic knowledge an agent has access to. This results in an idea of what the structure of the knowledge states should look like. One can then choose or define update operations and describe problem solving in an update system. Formally, an update system can be seen as the semantics of some logical language. We have seen three illustrations of this in section 6.4.3.

Hence, where knowledge representation usually begins by selecting a proper representation language, we start with the identification of the structure and end up with a semantic description of a representation language.

The framework presented here has focussed on the knowledge-level description and functionality of methods. Means of making a representation of knowledge states computationally attractive are topics for further study. As we have indicated, characteristic models and the use of tableaux are two promising ways of pursuing this.
Towards a task ontology for knowledge-intensive assignment problems

7.1 Introduction

In recent years much of the research in knowledge engineering has focussed on the construction of various ontologies. These should provide a specification of the vocabulary used in the formulation of several related problems, or domains. In particular, a task-ontology comprises the specification of vocabulary used to describe different problems which all belong to the same task, or problem type.

Building a task-ontology involves an understanding of the similarities as well as the differences among problem instances within a task. Comparing and ordering various problem formulations should be facilitated by a task-ontology. Methods for solving problem instances are themselves not part of a task-ontology. However, having a good classification of problems within the task, one could associate with each class a Problem Solving Method (PSM). In this way ontologies can be seen as the specification of a library of PSMs.

In the CommonKADS methodology [82] a distinction is made between synthetic and analytic tasks. In analytic tasks the "system" the tasks operates on, exists, whereas in synthetic tasks the system is constructed [82, 71]. Diagnosis is an example of an analytic task, where the system is the artefact to be diagnosed. Planning, scheduling and assignment are instances of synthetic tasks. The goal is to construct a plan, a schedule and an assignment, respectively. In this chapter we will present an ontology for assignment problems.

In knowledge engineering, problems where individuals have to be assigned to locations or resources are known as assignment problems [71]. Typically, the goal of assignment problems is to find a mapping (or matching) between the elements of two sets, one called subjects or components, the other resources. However, there are assignment problems in which the goal is to find a matching of individuals all belonging to a single set. In such a case there is no distinction between subjects and resources.
The individuals to be matched are usually described in terms of predicates and relations. Their description forms the bulk of the domain knowledge of the problem. Both constraints and preferences are often formulated in terms of abstract properties and types of the individuals, mentioned in the domain knowledge.

Because of the diverse nature of knowledge-intensive assignment problems, having some understanding in what ways they can vary can be helpful in constructing general representational models. Since in knowledge-intensive assignment problems several preferences of different complexity may appear, it is worthwhile to be able to recognize the different types, and having problem solving methods (PSMs) associated with them. A good understanding and survey of different types of (sub)problems can facilitate building and maintenance of large systems. The task ontology we are about to present should be helpful in recognizing different subproblems in a knowledge-intensive assignment problem.

7.2 A basic task ontology for assignment problems

In order to present a basic task ontology for assignment problems we make a distinction between the input, the output and the invariant domain knowledge of the problem type. The input part of the ontology provides a specification of the vocabulary for denoting the input structures of an assignment problem. The output consists of the type of output plus a specification of the goal. The goal is a general post-condition for the problem type. The domain knowledge consists of knowledge which is specific for the domain of the problem and remains invariant during problem solving. Domain knowledge plays a role in the general description of constraints, preferences, subjects and resources.

7.2.1 INPUT

We call the elements which are involved in an assignment problem individuals. In many knowledge-intensive assignment problems this set is partitioned in two blocks (disjoint subsets) and the goal of the problem is to match elements of one subset to the other. In some problems the set of individuals $I$ is partitioned in two sets $S$ and $R$ such that $I = S \cup R$ and $S \cap R = \emptyset$. These problems are called bipartite matching problems. (The definition of a matching follows below.) A bipartition of a set $S$ involves splitting this set into two disjoint subsets whose union results in $S$.

The two subsets $S$ and $R$ of a bipartition of individuals $I$ are usually called subjects or sometimes components, and resources respectively. In general, one can view subjects as the elements which have to be matched, whereas resources are those elements to which subjects are matched. For instance, when employees are matched to offices [57], the employees are subjects and the offices are resources.

Not all assignment problems are bipartite. Those which are not, are called non-bipartite assignment problems. As an example, consider a problem where one has to find pairs of persons who would like to share rooms with each other. In such a problem one cannot partition the set of persons in two. Every person can be assigned to another person within the set of persons (except him/herself). In general, when each individual can in principal be assigned to any other individual (except itself) the problem is called a single set assignment problem.
We will also use some terminology reminiscent of constraint satisfaction (CS). If at some stage we consider some individuals as candidates for assignment to individual $x$, we will refer to these as the values of $x$. When describing a constraint, values are possible resources for a subject. The possible values of $x$ will sometimes be referred to as the domain of $x$. Clearly, in a CS context one could read 'variable' where we use 'subject'.

The difference between the use of 'subjects' and 'resources' is often pragmatic. One usually talks of assigning subjects to resources, and not the other way round. This would suggest that the underlying graph of the problem is a directed graph but this is not necessarily so.

The use of resource can be misleading as it used in a different way in the description of a different task, namely scheduling. In assignment problems resources are the entities to which subjects are assigned to. They can be organized into types, which may have a capacity for holding more than one subject. However, this capacity is fixed and each subject will make the same demand on a resource type. In scheduling problems activities are assigned to resources. But different activities may set different demands on resources, and the capacity of resources may change during the problem solving process.

Another difference between assignment and scheduling problems is the notion of time. In scheduling problems "resources provide a time range in which units (activities) can be occupied to satisfy their demands" [82]. In assignment problems time usually plays no role. However, this distinction between the two tasks can become rather vague. After all, the linear ordering of activities can be seen as just another constraint in an assignment problem. When the capacity of resources is fixed, scheduling problems can be seen as a specific type of assignment problems.

In addition to subjects and resources, constraints and preferences are (invariant) parts of the input for an assignment problem. We provide some elementary, informal definitions. Later we will present a more elaborate definition.

**Definition 7.1 :** A constraint is a relation between one or more individuals which needs to be satisfied. In particular a constraint limits the number of matchings between individuals. A set of constraints is called consistent when there is at least one matching which satisfies at least one constraint in the set. Otherwise it is inconsistent.

**Definition 7.2 :** A preference provides a partial ordering of candidate solutions.

Constraints are usually represented as expressions in some formal or informal language. Their meaning can be formulated as a partition of solutions in two groups: those which are acceptable and those which are not.

As mentioned above, the domain of an individual is a list of legal values for that individual. The same information can be represented as a constraint. However, often domains are restricted without the use of constraints. The knowledge is then part of the domain knowledge.

Processing constraints may affect the domain of an individual. In the CS framework, problem solving prunes domains when processing constraints. When all constraints are processed one is left with only legal values in the domains of the variables. Every assignment of the variables to one of it’s legal values then constitutes a solution. Our use of 'domain' differs from the CS one in that by 'domain' we only refer to the set of legal values prior to any processing of constraints.
7.2.2 OUTPUT

It can be helpful to view a matching problem in terms of a graph $G = (V, E)$. For bipartite assignment problems the set of points $V$ of the graph is the union of subjects $S$ and resources $R$: $V = S \cup R$. The lines in $E$ connect elements from $V$. A matching can then be defined as follows:

**Definition 7.3:** A (maximal) matching in a graph $G$ is a (maximal) set of lines of $G$ in such a way that no two lines have a point in common. 

Informally a matching is an assignment where no two individuals can be assigned the same value, and no individual can have more than one value. Of course one wants to assign as many individuals as possible, hence one looks for a maximal matching. The assignment of one individual to another is referred to as a match. A match is an element of a matching and can be treated as an ordered pair.

In many problem formulations resources are described as having a capacity for being matched to more than one subject. For example, one can assign more than one person to a room, or several planes to a hangar. This would conflict with the definition of a matching. However, one can always reformulate such a case by replacing each resource with a capacity of $n$, with $n$ new resources each with a capacity of one. In our basic ontology rooms are types of resources. Resources themselves can not have a capacity of more than one.

The goal of an assignment problem is to find a matching which satisfies the constraints and is optimal with respect to preferences. This is a very general statement. When dealing with several preferences, one needs to have a criterion which describes which solution is ultimately preferred. The same is true for inconsistent sets of constraints. In such cases, one may want to find the best non-solution to the problem. A criterion is an expression which describes when a candidate solution is a solution for instances of the problem type.

7.2.3 DOMAIN KNOWLEDGE

The entities involved in matching usually have certain properties and relationships between them. For example, when matching passengers to airplane seats, passengers smoke or not, have a certain age etc. Seats are in business or economy class, near windows or not etc.

This domain knowledge can be quite extensive and is used in the formulation of constraints and preferences. Subjects and resources can be classified on the basis of their properties and be organized in a hierarchy of types.

A type is an abstraction of individuals on the basis of one or more properties. The set of individuals which are of the same type is called the type's extension. By an instance of a type we mean a specific element of the extension of that type.

We make a distinction between two sort of types: aggregations and generalizations. Aggregations are abstractions based on a 'part-of' relationship, e.g. 'room' is an aggregation of places in a room. Generalizations are abstractions based on 'is-a' relationships, e.g. 'smoker' is a generalization of employees who smoke.

Aggregations of resources have a capacity. Typically, a capacity indicates how many resources are covered by the aggregation. For example, 'room' is an aggregation of a number of places. This number is its capacity. However, a capacity can also be a more complex expression about the number of subjects which a resource type can hold. As an example, consider a large room to

---

¹In the following, when we use 'matching' we mean 'maximal matching'. The definition comes from Lovasz [58].
which *either two researchers or one head of staff can be assigned*. Such expressions refer to the capacity of the resource type. We call them **capacity expressions**.

If one wants to talk about assigning persons to rooms, one has to deal with the capacity of the rooms. Hence, a capacity of an aggregation type refers to the number of individuals in its extension. We assume that the administration for computing the remaining capacity of an aggregation type during problem solving is part of the domain knowledge.

There can be more than one way to organize individuals in a hierarchy of types. For example, employees can be classified according to the role they have in an organization, or according to the skills and experiences they have. A classification is therefore dependent on a viewpoint.

In many cases one can think of a type as an *abstract data type*. Individuals are often instances of such types. The exception is formed by the aggregation types.

All constructs for modelling domain knowledge used by Schreiber *et al.* [82] can in principle be used to describe the domain knowledge of assignment problems. Individuals can have properties and relations among them. Constraints and preferences often make extensive use of abstracted types, rather than specific individuals.

Types are often used to define domains for individuals. For example, a type like *colour* may contain all values (colours). The definition of a domain for an individual is also part of the domain knowledge. (Notice that 'domain' in 'domain of an individual' and 'domain knowledge' has two different meanings.)

### 7.2.4 A note on methods

In the ontology presented here, we will not discuss problem solving methods or algorithms. The description of methods is not a part of a task-ontology, which is limited to a specification of the vocabulary for the task description.

In general, when describing methods for solving knowledge-intensive problems one usually makes a distinction between *weak* and *strong* methods. Strong methods are tightly linked to the initial problem formulation and are almost tailor-made for the problem instance at hand. On the other hand, weak methods are more general and one often needs to cast the problem into a particular representational mould first. Weak and strong methods can be viewed as two extremes of a continuum. Hence, methods can be compared to each other according to their 'strength', or place in this continuum.

The specification of a Problem Solving Method (PSM) is usually described (roughly) in terms of input, output, domain knowledge and an inference structure. The inference structure should specify some basic flow control and can be seen as a template for a problem solving method of the task.

The ontology presented in this chapter will sometimes be similar to the representations used in Constrain Satisfaction (CS) [63, 12, 33, 96]. We regard CS as a powerful, (weak) method for solving many constraint-based assignment problems. We will focus on the representational aspect of such problems, instead of how to specify the different methods for solving them.

In the literature, the CS framework has been extended to partial constraint satisfaction [33] and Generalized Constraints [16]. We will treat these, particularly the last one, extensively below.

Another class of methods (or algorithms) can be found in the literature for Stable Marriage Problems. (The best reference is Gusfield & Irving [39] which describes the history of the problem,
and provides algorithms and a formal analysis of many variations of the problem.) We will treat Stable Marriage Problems here, but algorithms for solving them are out of the scope of this chapter.

Methods for Stable Marriage Problems can be considered strong, rather than weak. The representation of the problem is tightly coupled to the algorithms used to solve them.

7.3 A basic classification of assignment problems

On the basis of the ontology represented above, one can begin to make some distinctions regarding the nature of some assignment problems. First, we present a formal definition of an assignment problem.

**Definition 7.4**: An assignment problem is a tuple \((I, C, P, D)\), where \(I\) is a set of individuals, \(C\) as set of constraints, \(P\) a set of preferences and \(D\) the domain knowledge. The goal of the problem is to find a matching \(M\) such that it is a maximal matching on \(I\), satisfies all constraints \(C\) and is most preferred according to the preferences in \(P\). Note that the constraints in \(C\) and preferences in \(P\) may refer to knowledge in the domain knowledge \(D\).

A first classification of assignment problems is based on the nature of the matching, and is the one between bipartite and non-bipartite problems. A second distinction, to be described here, is based on the nature on the set of preferences and is the one between one- and two-sided problems. We now describe these two problem classes in detail.

- **Bipartite and non-bipartite problems.**

As remarked above for some problems the set of individuals can be partitioned into two subsets \(S\) (subjects) and \(R\) (resources) and matchings consists of pairs \((s, r), s \in S, r \in R\). Such problems are called bipartite assignment problems. When we consider an assignment problem, and ignore the constraints and preferences, then a problem is completely bipartite if every element from \(S\) can be assigned to any element from \(R\) and to no others. However, often the candidate matches for an element \(s\) are constrained to a subset dom of \(R\). In that case dom is the domain of the element \(s\).

**Non-bipartite** assignment problems are those problems for which the matching is not necessarily across a bipartition of the individuals. In the ideal case every individual can be matched to any other individual, except itself. Such problems are called single-set assignment problems. Here the domain of any individual is the set of all individuals minus itself. Hence, every individual is part of all domains except its own. With some abuse of vocabulary we can say that in a non-bipartite assignment problem, all individuals play both the roles of subjects and resources.

Like the ideal bipartite case, one can strengthen single set problems by introducing more restrictive domains. In that case the domain of individuals can be restricted to a subset of the set of individuals.

In knowledge-intensive settings single-set assignment problems often occur as subproblems. For example, consider a problem where students have to be assigned to rooms. Each rooms can contain two students. Assume there are several constraints and preferences that have to be met. Instead of assigning students to rooms one could also form pairs of students. When pairs are formed in accordance with the constraints and preferences, these can then
be assigned in random order to rooms. This subproblem of finding groups (in this case pairs) is known as grouping. Grouping involves finding a matching within one set of individuals.

- **One and two-sided problems.**

Another general distinction is based on the nature of the set of preferences. Assignment problems can be divided into two categories: one-sided and two-sided problems.

In two-sided problems individuals express preferences for each other. For example, when assigning males to females for a dating service, both males can have preferences for females, and females for males. In a sense the preferences go in two directions: from males to females and vice versa.

In one-sided problems preferences go in one direction. Some individuals have preferences for others, whilst the ones which are subject of a preference are completely indifferent. An individual \( x \) is indifferent towards other individuals \( y_1 \ldots y_n \) when there is no preference which expresses any \( y_i \) as a preferred match above the others. By 'completely indifferent' we mean that no value of the domain of \( x \) is preferred above the other.

The distinction does not hold for constraints. They declare some matches to be unacceptable. It is easy to see that a two-sided constraint problem can be reformulated into a one-sided one. Consider a two-sided constraint problem with an individual \( x \) with acceptable matches \( a, b, c \). Because of the two-sided nature of the problem \( a, b \) and \( c \) may also declare whether \( x \) is acceptable to them or not. Suppose that \( a \) does not find \( x \) acceptable. Then we can reformulate this as a one-sided problem with a constraint which declares that \( a \) is not acceptable to \( x \). Hence, it does not make much sense to speak about two-sided constraint problems.

The distinction between one- and two-sided preferential problems can best be seen when the preferences are personal. A personal preference of an individual \( x \) is one which can be expressed as an ordered list of the domain of \( x \). The order of the list then is interpreted as the preference of a match for \( x \). For example, males can express their preference for a match with a female, by supplying an ordered list of females. And females can do the same for men.

On the basis of these two general distinctions one can categorize assignment problems into four categories: (1) bipartite and one-sided, (2) bipartite and two-sided, (3) non-bipartite and two-sided, and (4) non-bipartite and one-sided. We will give examples of problems for each of these categories.

1. As a typical example of bipartite, one-sided problems we will take the office assignment problem of the Sisyphus I experiment [57]. (We will treat this example in detail in section 7.9.) Here employees had to be assigned to offices. Hence, the set of individuals is bi-partitioned into a set of subjects (employees) and resources (places in offices). The preferences mentioned in this example are all one-sided.

In bipartite, one-sided problems preferences are often introduced as a way of dealing with inconsistency or overconstrained problems. Preferences are often attempts to maximize an unsatisfiable constraint. When they appear in problems, preferences are often not wishes of individuals, but orderings of candidate solutions.
2. A textbook example of a bi-partite, two-sided problem, is the Stable Marriage Problem [39]. In the standard case the set of individuals is bi-partitioned in \( n \) males and \( n \) females. Each male and each female express their preferences for a partner in a preference list. These lists are complete, in the sense that each male ranks all females in his list, and each female ranks all males in her list. Matches (male/female pairs) have to be formed in such a manner that all matches are stable. The standard notion of stability says that a matching is stable when no two pairs which are not partners prefer each other to their current match. In other words, a couple \( a, b \) is not stable when the potential partners which \( a \) prefers to \( b \), do prefer \( a \) to their current partner. (The same reasoning applies to \( b \).) So a marriage is stable when neither male or female can persuade another partner to break up and form a better marriage.

As an example consider the following preference lists for males \( M = \{a, b, c\} \) and females \( F = \{A, B, C\} \).

\[
\begin{align*}
\text{a:} & \quad \text{ABC} & \text{A:} & \quad \text{bac} \\
\text{b:} & \quad \text{ABC} & \text{B:} & \quad \text{acb} \\
\text{c:} & \quad \text{BCA} & \text{C:} & \quad \text{abc}
\end{align*}
\]

Male \( a \) prefers female \( A \) to \( B \) and \( C \). Female \( A \) prefers \( b \). The matching \( \{aC, bB, cA\} \) is not stable because \( a \) and \( B \) prefer each other to their current matches \( C \) and \( b \). The matching \( \{aB, bA, cC\} \) is stable.

3. An example of a two-sided, non-bipartite assignment problem is the Stable Roommate Problem (SRP) [39]. In fact, this is a generalization of the Stable Marriage Problem (SMP) and the representation of both problems is very similar.

In the Stable Roommate Problem the goal is to form pairs of persons (roommates) in such a way that the pairs are stable. Pairs are stable in the same sense as marriages are in the SMP. A matching is stable when no two persons prefer each other to their current roommate.

The difference with SMPs is that in SRPs each person has a preference list which ranks all other persons in order of preference. In other words: there is no bipartition of individuals, like males and females. As an example consider the following SRP: There are four persons \( a, b, c, d \) with the following preference lists.

\[
\begin{align*}
\text{a:} & \quad c \quad b \quad d \\
\text{b:} & \quad a \quad c \quad d \\
\text{c:} & \quad b \quad a \quad d \\
\text{d:} & \quad a \quad b \quad c
\end{align*}
\]

The goal is to find stable pairs of roommates such that every person is part of a pair. This problem is an example of a single-set problem since every individual (person) can in principal be assigned to any other individual except itself.

SRPs are clearly similar to SMPs. In both types of problems preferences are personal and two-sided. SMPs are bi-partite problems, whereas SRPs are non-bipartite, and single set. SMPs can be formulated as Stable Roommate Problems (SRPs), but they form a clearly
distinguished class. For SMPs algorithms exist which always produce a stable matching. This is not guaranteed for SRPs. In fact, the problem instance presented above has no solution. Hence, all SMPs have a solution, whereas some SRPs have not [39].

4. An example of a one-sided SRP can be given by allowing only a single person to have a preference list. This makes the problem trivial. However, since the problem is a single set problem there can be no other person with a preference list without given up the one-sidedness of the problem. This is so, because in a single set problem a persons preference list contains all persons except him-/herself. And when two persons have a preference list they both are owners of a preference list and are part of one. This makes the problem a two-sided problem.

One-sided SRPs (and SMPs) make not much sense anyway, since the stability criterion is based on two-sidedness. In general, problems which are both one-sided and non-bipartite are either trivial or collapse into bipartite problems. Problems with only a single person allowed to have a preference list are trivial.

If one restricts preference lists to range over a subset of all individuals, more than one person can have a list, but then the problem becomes bipartite. The reason for this is that one-sidedness imposes a bipartition itself. One-sidedness of preferences involves a partition of the set of individuals into those which have a preference and those which are subject of preference. It is clear that matches will always be made between elements of both these sets, and hence the problem is bipartite.

It may also happen that in a one-sided problem there are individuals which are neither subject of a preference or object of a preference, and therefore fall into a third subset. These individuals can, trivially, be assigned to each other.

7.4 Constraints and preferences

The above classification was based on general characteristics of assignment problems. Next, we will proceed by focussing on the nature of constraints and preferences themselves. We will show that preferences can be seen as generalizations of constraints. In this section we first present the ideas put forward by Brewka et al [16]. In the next section we will show how this can be linked to the ontology described above.

7.4.1 Overconstrained problems

Consider assignment problems in which one is given a set of constraints and the criterion that a matching is a solution when it satisfies all the constraints. In this case it can very well happen that there are no solutions to the problem. In other words: a solution which satisfies all constraints for a given problem might not exist. In that case the problem is said to be overconstrained, and the set of constraints inconsistent.

In general, there are two approaches for dealing with this. One is to look for the best non-solution, the other is to redefine, or extend the definition of a constraint. Basically, these approaches amount to the same, but lead to different terminology. We will first show how one can
deal with overconstrained problems by picking the best non-solution. Then we change the definition of constraint accordingly, namely in such a way that it includes preferences.

One approach for solving overconstrained problems is to make a distinction between those constraints which have to be satisfied and those which should preferably but not necessarily be satisfied. The first category usually contains so-called hard constraints, the second soft constraints. The use of this distinction is strongly connected with the criterion that one wants a solution which satisfies all hard constraints and as many soft constraints as possible.

In fact, distinguishing between hard and soft constraints is an instance of the general approach of ordering all maximal consistent subsets of the set of all constraints. A set of constraints is maximal consistent, iff it is consistent and can not be extended by adding another constraint without given up consistency.

**Definition 7.5:** (Approximate solution I) Let $C$ be a set of constraints. Let $<$ be a strict partial ordering on the maximal consistent subsets $C_1 \ldots C_n$ of $C$. The most preferred solution $s$ is the one which satisfies the set of constraints which for the minimal element of this ordering.

The distinction between hard and soft constraints imposes an ordering $<$ by demanding that every subset contains the hard constraints, and $C_i < C_j$ when $C_i$ contains more soft constrains than $C_j$.

In case a problem is overconstrained the hard-soft distinction allows one to deal with troublesome constraints. They can be declared as 'soft' and will be pruned when they lead to unsatisfiability. However, this approach has important limitations. All soft constraints are treated with equal status. There is no room to declare that some constraints are more important than others.

This can be remedied by distinguishing more than just the hard and soft categories. The set of constraints $C$ can be partitioned in a number of blocks $C_1 \ldots C_n$ each with an own level of importance. A solution to the problem is then defined as a solution which satisfies a maximal consistent subset of $C_1$, augmented with a maximal consistent subset of $C_2$ etc.

This approach is similar to the preferred subtheories approach of Brewka [15] in the field of default reasoning. The main difference is that with preferred subtheories one deals with a logical theory, instead of a set of constraints. In the case of preferred subtheories, the logical theory is partitioned into subtheories which all represent a different degree of reliability.

The approaches for dealing with overconstrained problems are described on the level of all available constraints. One picks the best non-solution after having made a distinction between more and less important constraints.

However, this approach does not suffice for all overconstrained problems. Often one is looking for the most optimal solution to a single constraint. One popular way of doing this is by counting the number of constraint violations of candidate solutions and prefer the one with the lowest number. This technique is employed in partial constraint satisfaction, see Freuder [33] for details.

As an example consider a constraint like "all employees should be assigned a room in department A". Suppose this constraint is unsatisfiable and not all employees can be assigned to department A. In that case, one might settle for as many employees as possible, instead of all. Hence, instead of disregarding a constraint like this one would like to replace it by a weaker constraint of the form "As many employees as possible should assigned a room in department A" and prefer the solution with the largest number of employees on the department.

Hence, in such cases one does not order the constraints, but the solutions (matchings) themselves.
**Definition 7.6 :** *(Approximate solution)* Let $C$ be a set of constraints and $<$ an ordering of the solutions of all maximal consistent subsets of $C$. $s$ is an approximate solution if it is a $\ll$-minimal element of this ordering.

However, there is still something lacking in this solution. A solution is still defined in terms of maximal consistent subsets of constraints. Consider a problem where there is a choice between "heavily" violating a few constraints, or "weakly" violating a superset of these. One may very well opt for violating more weaker, than fewer heavier constraints. Remedying this problem lead Brewka et al. to a new notion of constraint networks.

### 7.4.2 Generalized Constraint Networks

Brewka et al [16] present a generalized notion of constraint problems which defines classical constraints as a limited case of a general preferential structure. By doing this, all of the ways of dealing with relaxing constraints, as discussed above, can be captured in an uniform way.

One has to note that the framework proposed by Brewka et al. is a generalization of the Constraint Satisfaction (CS) framework. The terminology reflects this. In CS one looks for a value assignment of the elements of a set of variables. Associated with each variable is a domain of values, its domain. A constraint in the CS framework can be seen as a function which maps each value assignment to true or false. The CS framework is generalized by the definition of a Generalized Constraint Network.

**Definition 7.7 :** A Generalized Constraint Network (GCN) [16] consists of the following items:

- $V = \{v_1 \ldots v_n\}$ is a set of variables.
- $D = \bigcup_{1 \leq i \leq n} D_i$ is a set such that $D_i$ is the domain of $v_i$.
- $C = \{c_1 \ldots c_k\}$ a set of generalized constraints where each generalized constraint is a strict partial order on value assignments.
- $\text{Comb}$ is a combination function which, given $C$ produces a single partial order on value assignments.

**Definition 7.8 :** *(Solution)* Let $CN = (C, V, D, \text{Comb})$ be a constraint network. A value assignment $\text{val}$ is a solution of $CN$ iff $\text{val}$ is a minimal element of the partial order $\text{Comb}(C)$.

Intuitively, this can be explained as follows: each constraint (or preference) is a judge with his own preferences for value assignments. A committee of judges will decide which of the value assignments is ultimately to be preferred above the other. This is done in accordance with a combination rule.

In the classical, CS case the judges only describe their preferences as being either acceptable or unacceptable. The combination rule in this case is the veto-rule: All judges have to agree a value assignment is acceptable if it is to be a solution.

Of course more complicated examples are possible. The combination function may take many forms. One can think of assigning numerical weights to some value assignments or count the number of times a constraint is violated by a particular value assignment. Such techniques are used in partial constraint satisfaction [33].
7.5 Using GCNs in the assignment ontology

As remarked above, the GCN framework is a generalization of the standard Constraint Satisfaction approach. Here, we want to incorporate it into the ontology for assignment problems. For this purpose we translate the terminology of GCNs into the vocabulary of the ontology used so far.

In the GCN framework, solutions are value assignments. In assignment problems the solutions are matchings. Value assignments are assignments of variables to values, and are not necessarily matchings, since two variables may well have the same value. Matchings therefore correspond to value assignments with a constraint of difference, stating that all variables must have different values. (The relation of constraints of difference, matchings and value assignments is used by Regin, in the description of an improved CSP algorithm [75].)

In CSPs the set of values and variables are distinct and hence value assignments seem only to refer to bi-partite matchings. We will come to this problem in the following subsection.

The main motivation for incorporating the GCN approach in our framework is the notion of generalized constraints as orderings of value assignments. Generalized Constraints correspond to preferences in our ontology. We will use the GCN framework to provide a formal definition of the notion of preference.

In the following we will use preference as a generalization of constraint. A constraint distinguishes two sorts of matchings: those which are acceptable, and those which are not. This can be expressed as an ordering on matchings but can also be expressed simply as a set (as is done in CS), where the complement contains the unacceptable matchings.

The combination function Comb in GCNs resembles the notion of criterion from the basic ontology. The combination function specifies how to combine all the preferences into one preferential structure. In this ordering the minimal element is the preferred solution. The criterion of an assignment problem can be seen as the specification of such a function.

7.5.1 Adjustments to the GCN framework

As a generalization of the CS framework, GCNs can be used to describe bipartite, one-sided assignment problems. A bipartite, one-sided assignment problem can be represented as a GCN as follows. The set of individuals is bipartitioned into subjects and resources. The subjects are represented as variables, the subjects as values. A matching is a value assignment with a constraint of difference. If constraints and preferences are one-sided they do not pose any problems to this representation. They express which value assignments are acceptable, or preferable.

However, it is not immediately clear how two-sided, or non-bipartite problems can be described as GCNs. As an example of the problems concerning two-sided preferences, consider a SMP. A set of males $M$ has to be matched to a set of females $F$ and each male and female have a list of individuals of the other sex in order of preference.

In a GCN the preferences are orderings of value assignments (matchings, in this case). Hence, when dealing with SMPs, the preference list of each male and female must be translated in an ordering of matchings. This is not difficult. Suppose a person $x$ (male or female) has the preference list $a, b, c$. Then $x$ prefers all matchings where $x$ is matched to $a$, to all matchings where $x$ is matched to $b$. And all matchings where $x$ is matched to $b$ are preferred to all matchings where $x$ is preferred to $c$. In general, an individual $x$ prefers matching $M$ to $M'$ if $x$ prefers his match in $M$...
to his match in $M'$. Preference lists in a SMP can therefore safely be interpreted as orderings of matchings and therefore are in accordance with the GCN approach.

However, the GCN framework requires that the problem is represented in terms of variables and values. When one tries to describe an SMP this way, one faces the choice which individuals to represent as variables and which as values. If males are represented as variables, then the females are the values. But alternatively, one may choose to represent females as the variables and males as the values.

There is no clean solution to this, although one can come up with a patch: We represent each male and each female by a value and an individual. Then the following condition (or meta-constraint) should hold for all individuals $x$. If $x$ is assigned the value $d$ then the individual denoting the same male/female as $d$, let's say $y$ should be assigned the value $d'$ which denotes the same female/male as $x$.

We can adapt the definition of GCN and make it suitable for two-sided problems like SMPs as follows:

**Definition 7.9:** An Adapted GCN for SMPs consists of the following items:

- A set of individuals $I = \{e_1 \ldots e_n\}$ bipartitioned into two equal subsets $M, F$.
- A set of domains $D = \bigcup_{1 \leq i \leq n} D_i$ such that the domain $D_i \subseteq M$ if $e_i \in F$, otherwise $D_i \subseteq F$. $D_i$ is the domain of the individual $e_i$.
- An ordering on each domain $D_i$.
- A combination function $\text{Comb}$ which given the individuals and their ordered domains produces an strict partial ordering on matchings.

The set of individuals is partitioned into two equal sized subsets, $M$ and $F$. With every individual we associate an ordered domain, such that if the individual is in $M$ ($F$) the domain is a subset of $F$ ($M$). The combination function then translates these ordered domains to an ordering of matchings.

This adapted GCN is stronger than the original. We have shown how each ordered domain can be seen as an ordering of matchings. But not all orderings of matchings can be represented as ordered domains of individuals. (We will discuss this in the next section.) Also, every individual is both represented as a domain and as a possible value.

The situation for one-sided problems, like SRPs, is not very different. As we have seen these problems can be safely assumed to be two-sided, and each individual will therefore have to be represented both as a value and a variable. The GCN for SMPs can be made suitable for SRPs by giving up the demand that the set of individuals must be bipartitioned and allowing the domains to be a subset of the set of individuals, excluding the owner of the domain.

### 7.5.2 Preferences as ordered partitions

In the ontology presented here, preferences can thus be seen as a strict partial order on matchings. This ordering can be seen as the formalization of the preference relation. A strict partial order $<$ is an irreflexive, and transitive relation. (It is also antisymmetric which means that whenever $x < y$ and $y < x$ then $x = y$.) Given a partial order, we introduce another relation $\sim$. If $< is
a strict partial order then the relation ~ is defined as follows: $m_i \sim m_j$ if neither $m_i < m_j$ nor $m_j < m_i$ [29].

We use strict partial orders to express preference relations, like a is preferred to b. In order to express that a is as much preferred as b we make use of the relation ~. However, we must be careful here. Obviously, x is as preferred as y is an equivalence relation (which is reflexive, transitive and symmetric) but ~ does not need to be transitive as it was defined above.

To illustrate this point we first distinguish three types of strict partial orders. These are illustrated in figure 7.1.

![Figure 7.1](image)

The partial order on the left shows four elements linearly ordered. The middle one, shows a weak order, the right one a tree. The order should be read from left to right: if (and only if) $x < y$ then there is left-to-right path from x to y. $x \sim y$ holds when there is no left-to-right or right-to-left path between nodes x and y.

These three types are based on the nature of the relation ~. The order on the left shows four elements linearly or strictly ordered. The relation ~ here, takes the meaning of the identity relation. That is: if neither $x < y$ nor $y < x$ then $x = y$. This is not the case in the order shown in the middle. Here, all six elements are connected by a line and ~ is transitive. Important for our purposes is that in this case we can safely interpret ~ as as preferred as y. As an example, one can look at the elements a and b in figure 7.1. Clearly $a \sim b$ holds, for there is no left-to-right or right-to-left path connecting them. Both are less preferred than all elements on the left and more than those on the right.

The order on the right shows a tree and here ~ can have a different meaning. In this case ~ is not transitive, and therefore no equivalence relation. For example, for nodes c and d we have $c \sim d$ but also $d \sim e$. If ~ would be transitive we should have $c \sim e$ but this is not the case: c and e are joined by a line. Hence, in this last order we cannot interpret ~ as as preferred as but instead have to resort to is incomparable with.

Orders for which ~ is not transitive can represent preferential relations of which knowledge is only partial. For example, one may lack the knowledge whether a should be preferred to b, b to a or if they are equally preferred. In such a case one may decide to declare them incomparable [60]. Another option is to try to fill in the absent knowledge and so force ~ to become transitive. For example, in figure 7.1 c, d and e are incomparable. If one assumes that d is preferred to e then the graph becomes equivalent to the middle one. One may also make different assumptions: for example that d is preferred to f and equally preferred to e.

In this way every preferential ordering based on partial knowledge has an 'extension' in which preferential knowledge is complete and all elements can be compared to each other.
In the remainder of this section we will focus on preferential orders based on complete knowledge only. In other words, we will assume that all two different matchings are either equally preferred or one is preferred to the other. The types of strict partial orders for which this holds are known in the literature as weak orders, and strict (or linear) orders. (Notice that there is a difference between strict orders, and strict partial orders. All orders discussed here are strict partial orders.)

Definition 7.10: If \( A \) is a set and \(<\) a strict partial order, and \( \sim \) is transitive, then \( \langle A, <, \sim \rangle \) is called a weak order.

A strict or linear order is a weak order for which \( \sim \) is the identity relation.

Notice that both in the case of a weak and strict order, we can safely interpret \( \sim \) as as preferred as. In the linear case, \( \sim \) is actually the identity relation and hence there are no different elements equally preferred.

When \( \langle A, < \rangle \) is a weak order, then we can partition \( A \) in subsets of elements which are equally preferred. For example, nodes \( a \) and \( b \) in figure 7.1 are equally preferred and form a subset of the set of all nodes of the graph. Hence, \( \sim \) defines a partition on the set of matchings in such a way that when \( x \sim y \) holds, \( x \) and \( y \) are in the same block (or equivalence class) of the partition. These blocks themselves are linearly ordered by preference [29]. When ordering matchings we will call these equivalent classes, i.e. sets of matchings which are equally preferred, categories.

For strict orders the situation is simple, there are just as many categories as there are elements. This is so because no two different elements are equally preferred in a strict order.

One may characterize preferences based on the number of categories they have. Suppose that for an assignment problem there are \( n \) possible matchings. As we have seen, for each classical constraint there are only two categories: acceptable and unacceptable. The set of matchings for a given problem is bi-partitioned in this case, and the number of categories equals two. All \( n \) matchings are distributed over these and there can be at most \( n \) matchings in one of the categories. If we consider a preference which orders all matchings in a linear fashion then the number of categories is \( n \), each of which contains exactly one matching.

In general, if we consider a preference \( c \) which imposes a weak ordering \(<\) on value assignments, then the associated relation \( \sim \) gives us a partition on value assignments. The blocks (categories) contain matchings which are equally preferred and the blocks themselves are linearly ordered by preference. This is illustrated in figure 7.2.

![Figure 7.2](image)

The picture shows matchings \( M_1 \) to \( M_6 \) weakly ordered. A matching is preferred to another if there is a left-to-right path from one to the other. For example \( M_1 \) is preferred to \( M_5 \) and \( M_6 \) etc. and no matching is more preferred than \( M_1 \) and \( M_2 \). Equally preferred matchings, which form a category, are shown in boxes. The categories themselves are linearly ordered.

The trick being performed here is that we transform a weak ordering of matchings to a linear ordering of categories. For linear orderings of matchings no such trick is needed. In this case the categories each contain a single matching. We can thus view weak and linear ordered preferences
as linear orderings of categories.

In order to compare these orderings of categories we make use of the fact that categories are blocks in a partition. In general, partitions of a set can be ordered by a refinement relation. Intuitively, a partition $P$ is a refinement of the partition $Q$ ($P \subseteq Q$) if $P$ divides the set $S$ along the same lines as $Q$ and maybe some more.

Similarly, one could define one preference to be a refinement of another. In that case, one has to take care that the order imposed on the blocks is not disturbed.

**Definition 7.11**: Consider two preferences $p_1$ and $p_2$, which weakly order the set of matchings $M$ by $<_1$ and $<_2$ respectively. The relations $\sim_1$ and $\sim_2$ express equally preferred matchings for $p_1$ and $p_2$ respectively. The partition $\sim_1$ imposes on $M$ is $P_1$, the partition $\sim_2$ imposes on $M$ is $P_2$. Preference $p_1$ is a refinement of $p_2$ iff the following conditions hold:

1. Partition $P_1$ is a refinement of $P_2$: $P_1 \subseteq P_2$

2. $p_1$ preserves the order of $p_2$: For each pair of matchings $m, n \in M$ if $m <_2 n$ then $m <_1 n$.

Hence the refinement of preferences is equal to the refinement of their associated partitions plus the preservation of order. An example will illustrate this:

**Example 7.1**: Suppose that persons have to be assigned to departments. There is a constraint $C_1$ which weakly orders all matchings of persons to departments in such a way that all matchings which assign 50 people or more to department $A$ are preferred to all other matchings. $C_1$ distinguishes two categories of matchings:

(1) Those matchings in which at least 50 people are assigned to department $A$.

(2) All other matchings.

A preference $C_2$ distinguishes three categories of matchings, presented in preferential order:

(1') Those matchings in which at least 50 people are assigned to department $A$.

(2') Those matchings in which between 40 to 50 people are assigned to department $A$.

(3') All other matchings.

$C_2$ is a refinement of $C_1$. It partitions category (2) of $C_1$ into two blocks (2') and (3'). Therefore the two partitions are refinements. Also, if a matching $M$ is preferred to another matching $M'$ according to $C_1$ then this is also the case according to $C_2$.

Notice that if we define a preference $C_3$ as:

(2'') Those matchings in which between 40 to 50 people are assigned to department $A$.

(1'') Those matchings in which at least 50 people are assigned to department $A$.

(3'') All other matchings.

then $C_3$ is a refinement of $C_1$, (but not of $C_2$). Hence there are several ways of refining a preference.
In this manner preferences can be seen as refinements of constraints. Constraints bipartition the set of matchings and distinguish acceptable the unacceptable matchings. Preferences can be seen as refinements of constraints, when they respect this order but introduce new preferential orderings in the acceptable and/or unacceptable categories.

In practical problems the refining of preferences often occurs, not only in assignment problems. One may start problem solving with an initial set of constraints or preferences. Afterwards it may happen that too many solutions have been found. In that case one can refine the preferences in such a way that one orders the solutions found thus far. Refining preferences comes down to 'zooming in' on one or more categories. Matchings which are equally preferred may change status after refining, and one can become more preferred than another. The opposite, or 'zooming out', means putting categories together in such a manner that the order is not disturbed. This can be useful when too few solutions to a problem have been found.

In fact, this exercise shows that one can make use of the formal aspects of the ontology to come up with new classifications. Formalizing an ontology not only leads to a more precise understanding of the notions involved, it also allows for new classifications on the basis of formal properties.

7.6 Representing preferences

In the previous section we have discussed preferences (and constraints) as orderings of matchings. However, in problem formulations preferences usually occur as expressions and in Stable Marriage or Roommate Problems they occur as preference lists of individuals.

The way preferences are represented differs among problems. It is therefore helpful to understand how a formulation of constraints is related to its 'semantics' in terms of an ordering of matchings.

We will first discuss the preference lists of SMPs and SRPs. It will be shown which preferences, i.e. partial orderings of matchings, can and which cannot be represented as preference lists. Next, we will use this characterization to come to a notion of dependency between individuals, expressed by a preference.

7.6.1 Preference lists

As was shown above, SMPs and SRPs present preferences as preference lists of individuals. In the standard Stable Marriage Problem a preference list is a strict order and is complete in the sense that it mentions every individual of the opposite sex. In the standard Stable Roommate problem preference lists contain all individuals except the owner of the list.

Variations of these problems allow the relaxation of these requirements. By allowing partial lists individuals can express the unacceptance of partners by excluding them from their list. Using weak instead of strict ordered lists makes it possible to express ties: some potential partners are equally preferred. When two or more partners are equally preferred to an individual he or she is said to be indifferent towards them. Indifference allows one to weaken a bi-partitioned, two-sided problem to a bi-partitioned, one-sided problem by declaring one side completely indifferent towards the other. For example, if in the standard SMP all females are indifferent towards the men (but the men not towards the women), then the resulting problem is one-sided.
When preference lists are complete and strictly ordered then the number of categories the preference distinguishes is equal to the number of elements in the list. If the list is partial the number of categories is equal to the length of the list plus one, (because the unacceptable partners are not in the list). When ties are possible one has to count the categories in the preference themselves: the number of sets of partners which are equally preferred.

As mentioned before, a preference list can be interpreted as an ordering of matchings: an individual \( x \) prefers matching \( M \) to \( M' \) if \( x \) prefers his/her partner in \( M \) to the one \( x \) has in \( M' \).

But not all orderings of matchings can be expressed as the preference list of an individual. Those which can be expressed this way form a special class, as we shall now show.

Suppose we have an ordering of matchings \( \langle M. < \rangle \) and all matchings from \( M' \subset M \) in which \( x \) is matched to \( a \) are preferred to all matchings from \( M'' \subset M \) in which \( x \) is matched to \( b \). In that case \( x \) prefers \( a \) to \( b \).

Hence, if (and only if) we have an ordering of matchings \( \langle M. < \rangle \) and the categories of the preference can be described in terms of representative elements of assignments of \( x \), then the preference can be represented as preference list of \( x \). More precisely, there must be a one-to-one, or isomorphic mapping between elements in the preference list and categories in the matchings ordering.

**Example 7.2:** Consider a problem where there are two subjects \( x, y \). \( x \) can be assigned to resources \( a, b, c, d \) and \( y \) to \( a, b \). Suppose there is a preferential ordering of six matchings \( M_1 \ldots M_6 \). These matchings are given in the table below. Suppose the preferential ordering is as follows: \( M_1 < M_2 < (M_3 \sim M_4) < (M_5 \sim M_6) \). Here, \( M_3 \) and \( M_4 \) are equally preferred and form a category, so do \( M_5 \) and \( M_6 \).

\[
\begin{align*}
M_1 : & (x, a), (y, b) \\
M_2 : & (x, b), (y, a) \\
M_3 : & (x, c), (y, a) \\
M_4 : & (x, c), (y, b) \\
M_5 : & (x, d), (y, b) \\
M_6 : & (x, d), (y, a)
\end{align*}
\]

(Here \((x, a)\) expresses that \( x \) is matched to \( a \).)

There are four categories in this preference order; two containing one matching and two containing two matchings. They can be characterized by assignments of \( x \), because \( x \) is assigned a different value in each different category, and the same value in matchings of the same category. Hence, each category contains a distinct assignment of \( x \). The strict ordered preference list of \( x \) denoting this ordering of matchings is \( a, b, c, d \).

It is clear that not all orderings of matchings can be described by an individual’s preference list. An ordering of matchings \( \langle M, < \rangle \) can only be represented as a strict preference list \( a_1 \ldots a_n \) of \( x \) if each match \((x, a_i)\) \((1 < i < n)\) occurs in exactly one category of \( \langle M, < \rangle \) and preserves the order of the categories.

There is another way of looking at these preference lists. They can also be be characterized as ordered assignments of single individuals. The preference of \( x \) only expresses what \( x \) prefers, without consideration to the preferences of any other individual: \( x \) is indifferent about any match which does not involve \( x \). The preferences of \( x \) are thus unrelated, or independent with respect to any other individuals preferences.
Towards a task ontology for knowledge-intensive assignment problems

Those constraints which can be expressed as "preference" lists for individuals are exactly those which limit the domain of a single individual. Constraints order matchings in two categories by definition. If in all acceptable matchings \( x \) is assigned one of the values \( a, b, c \) and in all unacceptable matchings other values, then this constraint just limits the assignments of \( x \) to \( a, b, c \).

In CS terms these constraints are \textit{unary} since they range over a single individual. Likewise, we will call preferences, which can be expressed as a preference list of one individual, unary, since the entries in the preference list are independent of other individuals.

### 7.6.2 Dependency in preferences

A \textit{binary constraint} with individuals \( x \) and \( y \) expresses that there is a dependency between \( x \) and \( y \). This means that if we assign \( x \) some resource then this affects the choice of an assignment for \( y \). Notice, that a dependency is not symmetric: it can happen that \( x \) is dependent on \( y \), but \( y \) not on \( x \).

In general a \textit{n-ary constraint} involves the dependency of \( n \) individuals. Clearly, \textit{unary} constraints only involve one individual. Our aim is to generalize this notion of dependency, which is common in the CS literature, to preference lists of individuals. We propose the following definition:

\textbf{Definition 7.12}: A preference list of \( x \) is dependent on \( y \) if the occurrence of some entry in the preference list of \( x \) depends on some assignment of \( y \).

This definition states that the preferences of an individual can depend on the assignment of another individual. Strictly speaking, the fact that one searches for a matching does already express a dependency. Since, a matching can be seen as a value assignment with a constraint of difference, assigning one individual affects the possible assignments of others.

Apart from this constraint of difference, which we regard as implicit in the definition of matching, dependency does not occur in preference lists in the standard SMPs or SRPs. However, there is one variation of SMPs in which mutual preference lists for two persons are used [39]. This problem is called the hospital-residents problem with couples, and typically involves the assignment of a number of residents to a (not necessarily equal) number of hospitals. The problem comes with an adapted stability criterion, which we will not present here (see Gusfield and Irving [39] for details).

The problem is bipartite and two-sided. Both residents and hospitals express preferences for each other. Hospitals have a capacity for housing more than one resident. The interesting element here is that residents may form couples and present a \textit{joint} preference list. If residents \( x \) and \( y \) form a couple then there is obviously a dependency between them.

\textbf{Example 7.3}: Consider two persons \( x \) and \( y \) and hospitals \( h, k \), each capable of taking two persons. In general, the joint preference list of a couple is an ordered list of ordered pairs. In this case the list of \( x \) and \( y \) is : \((h, h), (k, k), (h, k), (k, h)\). A pair \((h, k)\) represents the assignments of \( x \) to \( h \) and \( y \) to \( k \). So \( x \) and \( y \) prefer the joint assignment to \( h \) to the joint assignment to \( k \).

Clearly, there is a dependency between \( x \) and \( y \). If we assign \( x \) to \( k \) then \( y \) prefers \( k \) to \( h \). But if we assign \( x \) to \( h \), then \( y \) prefers \( h \) to \( k \). Hence \( y \)'s preference for hospitals is dependent of the

\footnote{The list is partial and only contains tied entries. It is the new domain of the individual \( x \).}
assignment of $x$. Similarly, $x$ is dependent on $y$.

We regard joint preference lists as an example of binary preferences, because they express the dependency between two individuals. Also note that all the entries of an individual in a joint preference list are dependent on the assignment of another individual. In that sense the dependency can be said to be total. Furthermore, if $x$ and $y$ have joint preference lists then $x$ is dependent on $y$ and $y$ on $x$. In that case we call the dependency between $x$ and $y$ mutual, or interdependent.

Not every preference list with a binary preference is a joint preference list. One could think of lists, which are not total, in which only some entries are dependent on the assignment of another individual. Such dependent entries do not have to be mutually dependent. One can think of entries as: if $x$ is assigned to $h$, $y$ prefers $k$ to $h$. Such 'conditional' preference lists are not described in the SMP literature. However, they would still qualify as binary preferences in our framework.

In the previous section it was shown that standard preference lists are equivalent to an ordering of matchings where there is a one-to-one mapping between categories and entries in the preference list, with the preservation of the order. For joint preference list a similar equivalence holds: there must be a one-to-one mapping between elements in the preference list and assignments of the owners of that list in the categories of the corresponding ordering of matchings.

For the joint preference list in the above example, it means that in the corresponding ordering of matchings there are four categories: one in which $x$ and $y$ are both matched to $h$, one in which they are both matched to $k$ etc. The order of the categories should be equal to the order of the entries in the preference list. It is clear that preferences are now described in terms of pairs because of the dependency between individuals.

Finally, we use the notion of scheme to express the dependency of a preference. (A similar notion of 'scheme' for constraints is used by Dechter [22].) The scheme of a preference is a set of tuples, each of the form $(x_1 \ldots x_n)$ such that $x_i$ is dependent on $x_{i+1} \ldots x_n$. Hence, in the above resident-hospital problem the scheme of the joint preference list is $\{(x, y), (y, x)\}$.

### 7.7 Combining preferences: criteria and combination functions

We have remarked that constraints order the set of matchings for a problem into two categories: those which are acceptable and those which are unacceptable. However, not all preferences which order all matchings into two categories are constraints.

As an example, consider my preference for a window seat in an aeroplane. A window seat may be available and I will be most pleased, but if no such seat is available I will accept another one. Hence my preference bi-partitions the set of matchings: those in which I am given a window seat, and those in which I am denied one, but neither of the two contain unacceptable matchings.

The interpretation that a category of matchings of a preference is unacceptable is part of the criterion. The notion of criterion is used to describe when a matching is a solution. However, we use it also to interpret the status of categories of preferences.

In the GCN framework the combination function is defined as a function which produces a final strict partial ordering on matchings based on the input of all preferences. A solution then was defined as a minimal element in this ordering. Strictly speaking, to arrive at a conclusion it is not necessary that the combination function produces the entire ordering, just the minimal elements, (or one of them) would suffice.
Towards a task ontology for knowledge-intensive assignment problems

One may wonder what use the other, non-minimal categories of this final ordering of matchings could have. But they can be of use when preferences are added to the problem at a later time. New information may prune all solutions found thus far, and one may use the ordering produced by the combination function to incorporate new knowledge.

Another reason why non-minimal categories (which contain sub-optimal solutions), can be useful, is that some preferential knowledge is not included into the representation of the problem. Some preferential knowledge may be too difficult or too expensive to capture in a formal or semi-formal way, and one may choose to browse the optimal as well as the sub-optimal solutions in the order presented by the combination function.

As mentioned above, Brewka [16] illustrated the functionality of the combination function by the simile of a system of votes. We would like to elaborate this simile and illustrate how preferences can be combined.

First of all, all preferences will be represented by a judge which is given the right to object to a matching. The strongest objection a judge can raise is to veto the matching, and declare it unacceptable. Every judge will object to matchings according to the categories of the preference he represents: he will raise stronger objections when the preference of matchings decreases.

For example, a constraint has two categories and it's judge will veto all matchings in the second category. My preference for a window seat also has two categories, but none of the matchings is vetoed. In this case the objection against a matching in which I am given a window seat is less than all others.

How strong the objections of a judge are for each category depends on the criterion. One can assign weights to each category for each preference, representing the strength of the objection the judge will raise. The combination function will then compute an ordering of matchings based on these weights.

One could think of many different ways of combining preferences this way. However, there are two invariants. First, judges will raise stronger objections to matchings when they appear in decreasing order in the preference they represent. Second, judges will object equally strong to matchings within the same category of the preference they represent.

What remains variant is the distance between categories. I may prefer Bach to Stravinsky and Stravinsky to Mozart, but the difference, or preferential distance, between my preference for Bach and Beethoven is much smaller than for Beethoven and Mozart. In other words: I like Bach and Stravinsky, but dislike Mozart.

Based on this simile of voting judges it is not difficult to come up with a general form of the combination function which should suffice for many one-sided assignment problems. All categories of each preferences are assigned numerical weights expressing the strength of objection, in such a way that the two invariants hold. The combination function then simply puts every matching to the vote and adds the weighed objections. It then produces an ordering in such a way that a matching is more preferred when its objection number is smaller.

Another example of a combination function captures the criterion of stability in standard SMPs. For these problems the judges representing the preference are the owners of the preference list. A matching $M$ is stable when no two judges which are not partners in $M$ both object less to a match in which they are partners. In this instance no numerical weights are necessary. In fact, the majority of algorithms which solve SMPs and related problems, do not make use of numerical weights.

Weighed combination functions have been proposed many times in the literature [12]. Usually
one assigns weights to preferences instead of the categories of preferences. The reason behind this is that one looks for solutions which satisfy a maximum number of constraints. Weighing systems like the ones used in Partial Constraint Satisfaction [33] are exceptions to this. Here one counts the number of times a constraint is violated, and opts for the solution with the less violations. These violations can be weighed and the result is a system which is similar to the one described here.

Note, that we have assumed here that all preferences are either weak or strict orders. In other words all matchings can be compared to each other in all preferences. When one has incomplete knowledge about the preference of two matchings (if they are incomparable) then no exact weight can be assigned to some categories. There are several options in this case. One could assign weights with a measure of uncertainty. Alternatively, one could assign intervals of weights when at least the lowest and highest weight of a category is known.

### 7.8 Abstractions in preferences

In initial problem formulations preferences (and constraints) are usually given as expressions ranging over individuals. In knowledge-intensive assignment problems individuals are often referred to in terms of their properties, relations or types.

In general, two sort of types can be distinguished: **aggregations** and **generalizations**. Aggregation types involve the use of a has-part relationship, whereas generalization types make use of the is-a relation.

For a given domain, every type has a set of instances associated with it. This set is called the **extension** of the type. For example the extension of the generalization type *smoker* contains all individuals that smoke.

Individuals and types can also have relations between them. The extension of an n-ary relation is the set of n-tuples of individuals for which the relation holds. Relations can sometimes have properties as well. As an example, consider the distance between two rooms. The floor-plan can be represented as a graph where the room types form the nodes and there is an edge from one node to the other if the rooms are next to each other. The length of a path from one room to the other, is the distance between the rooms. In modelling terms distance is a property of a relation between room types. With the help of the distance between rooms one can decide whether a room is centrally located, or not.

Properties (or features) are usually represented as attribute-value pairs. One can convert properties to types by defining its extension as containing all individuals which have the property. For example, one can use the attribute smoker with the value true to indicate that an individual is a smoker. Similarly, one can introduce a type smoker and put all smokers in its extension.

By using types this way one abstracts over individuals. Abstractions are usually seen as partitions on the set of individuals. Binary properties, like smoke/non-smoke, bipartition this set.

Preferences can be entirely formulated in types. As an example consider the constraint "smokers and non-smokers are not allowed to share a room". The types use here are smokers, non-smokers (generalizations) and rooms (aggregation). (The reason that room is an aggregation is that it can host more than one individual.)

In such preferences the use of types declares an indifference towards the assignment of individuals in the extensions. In the above constraint one does not distinguish between individual smokers, non-smokers and places in the rooms.
This property can be used to assign chunks of types, or groups instead of individuals. In the above example, when each room has two seats, then one can first form pairs of smokers and non-smokers and assign these. The assignment of individual persons to specific seats is irrelevant. Such grouping problems will be the subject of chapter 8.

In preferences with types and dependencies the scheme of the preference can sometimes be expressed in terms of types as well. Consider again the smoker/non-smoker constraint: "Smokers and non-smokers are not allowed to share rooms."

If a non-smoker is assigned a place in a room then this affects the possible assignments of all other smokers. Hence every non-smoker is dependent on every smoker. And by reasoning similarly, every smoker is also dependent on every non-smoker. The scheme of the preference then contains all possible pairs of smokers and non-smokers. In terms of types the scheme can be given much more elegantly as the set \{\langle{\text{smoker, non-smoker}}\rangle, \langle{\text{non-smoker, smoker}}\rangle\}.

This illustrates the fact that the use of abstractions in the formulation of preferences facilitates an easy and intuitive representation.

7.9 Case study: Sisyphus I

In the Sisyphus I project [57] researchers from the Knowledge Engineering community were invited to build a system which could solve an office-assignment problem according to some given criteria. Goal of the project was to compare different approaches, employed by different researchers, to a knowledge-intensive problem. The results, together with the original problem statement, were published in 1994 [57]. We will analyze the problem here as an example of the application of our ontology.

The problem presented was to assign a number of researchers to offices, in such a way that certain constraints and preferences were respected. The official Sisyphus problem statement can be seen as consisting of two parts. The first part which provides the explicit domain knowledge concerning researchers and offices. Second, constraints and preferences are given implicitly in the form of a think-aloud protocol by an expert. In order to formulate the constraints and preferences one has to interpret the protocol. The original problem statement, including the protocol can be found in appendix A.

7.9.1 Explicit knowledge in the Sisyphus domain

In the office assignment problem of Sisyphus I, researchers have to be assigned to offices. There are 15 researchers and 10 offices. Offices are represented in a floor-plan, see figure 7.3.

The offices to which researchers can be assigned to, are all on one floor. The grey coloured rooms C5-118, C5142, C5-144 and C5-143 (see floor-plan) are not available for the assignments of employees.

The offices C5-117, C5-119, C5-120, C5-121, C5-122, C5-123 are large offices and can host two researchers. The others are small offices which can host only a single researcher. A large office can also be used to host one head of group.

Researchers have a number of features and there are relations between them. As an instance consider the following representation of an employee, which was part of the original problem description:
Figure 7.3
The floor-plan of the office assignment problem. The grey coloured offices (C5-118, C5142, C5-144 and C5-143) are on a different floor and cannot be used for assigning researchers.

Similar representations were given for all subjects in the problem. Note that the attribute-value representation is a standard way of representing domain knowledge. In fact, the subjects can be modelled as instances of an abstract data type.

In the Sisyphus problem statement the resources (offices) have less attributes than the subjects. Apart from their capacity (capable of hosting one or two persons), information about offices is given by the floor-plan.

The employees are part of a hierarchical, organizational structure. In the original problem statement all the persons fulfilling the roles in this structure are mentioned by name. One person is head of the group, there is one manager and there are two secretaries. There are three large projects on which multiple persons work. In addition, there are two individual projects. There are three heads of projects, but only one of these projects is mentioned in the problem statement.

Apart from the capacity of the rooms, there are no explicit constraints or preferences in the problem statement. Instead, a think aloud protocol of a wizard solving the problem is given. The goal of the Sisyphus project was to build a system which could mimic this line of reasoning. Therefore the constraints and preferences should be extracted from the protocol. In fact this knowledge acquisition process is an important step in modelling the problem. We will give an analysis of the protocol here.
7.9.2 Implicit knowledge: constraints and preferences

The wizard starts with the assignment of the head of the group to a large central office. The protocol states: "The head of group needs a central office, so that he/she is as close as possible to all the members of the group. This should be a large office."

At face value this can be interpreted as the description of a preference: "The head of group should be placed as close as possible to all the other members of the group". However, it can also be interpreted as a much simpler constraint: "The head of group should have a large central office."

According to the wizard the assignment of the head of group is performed first because it restricts the possibilities of other assignments. However, it seems that the importance of the head of group is not totally irrelevant, and one could argue that the wizard performs assignments in an order which mirrors the importance of the people in the organization.

We interpret this statement in the protocol first as a constraint: "The head of group should have a large central office." When this appears too strong an assumption, we can refine the constraint to a preference, containing more than two categories. As for now, we describe the constraint as having two categories. The first one contains the matchings in which the head of the group is given a central office, the other one contains all other matchings. This is illustrated in the upper half (a) of figure 7.4. The bottom half (b) shows a possible refinement into more than one category. The "all-other-matchings" category should be interpreted as unacceptable.

![Figure 7.4](image)

The figure shows two interpretations of the preference that the head of group (HOG) should have a large office. (a) has two categories: the preferred, left one contains matchings assigning the HOG to C5-117 or C5-119. (b) is a possible refinement of (a). After C5-117 and C5-119 the preferences for assigning the HOG are C5-120 and C5-121.

We have interpreted the floor-plan in figure 1 in such a way that C5-117 and C5-119 are seen as large central offices. Supporting this interpretation is the fact that the wizard assigns the head of the group to C5-117. Looking at the floor-plan, C5-119 seems to be a reasonable alternative. We note that this constraint is unary, and hence does not involve a dependency on other individuals.

Next, the secretaries are assigned a room close to the head of group and the wizard remarks: "Both secretaries should work together in one large office". We interpret this as the combination of a constraint (together in a large office) and a preference (close to the head of group).

The constraint which states that the secretaries should work together in a large office is binary. There is an interdependency between the two secretaries and one can express this constraint as
a joint preference list. However, there only two categories: large offices and small offices. The small offices are unacceptable. It would be more interesting if the secretaries would have joint, personal preferences for an office. But this knowledge is not part of the problem statement.

The preference that the secretaries should be close to the head of the group (HOG), expresses a dependency between the secretaries and the head of the group. This dependency could be a reason for assigning the secretaries an office, immediately after the head of group, but this is not mentioned in the protocol.

The minimal number of categories of this preference is two. The set of matchings is bi-partitioned into those which assign the secretaries next to the head of group, and those which do not. In fact, this minimal interpretation will result in a correct assignment. Another, possibility is to refine this constraint and distinguish categories along a measure of distance relative to the two central offices.

The first two categories of such a refined preference are shown in figure 7.5.

![Figure 7.5](image)

The figure shows two categories regarding the assignment of the secretaries. Their joint assignment is dependent on the HOG, as is shown by a conditional statement. So, when the HOG is assigned C5-117 the secretaries will be assigned C5-119 etc.

The first, most preferred category contains the matching in which the HOG is assigned to C5-117 and the secretaries C5-119. Also part of this category are the two matchings, both in which the HOG is assigned to C5-119 and the secretaries to C5-117, and C5-120. The second category can be described as follows: if the HOG is in C5-119 then the secretaries go to C5-121 and if the HOG is in C5-117 then they go to C5-120.

For the sake of simplicity, figure 7.5 only shows the dependency of the secretaries to assignments of the HOG with regard to the previous preference. If we would treat this preference in isolation we would have to take into account every possible assignment of the HOG. Now we have limited its assignments to just C-117 and C-119.

The next entry in the protocol assigns the manager an office. "She must have maximal access to the head of group and to the secretariat. (...) she should have a centrally located office. A small office will do." Strictly speaking, this preference is dependent on both the assignments of the head of group and the secretaries. The remark in the protocol "this is the earliest point where this decision can be taken" can be interpreted as referring to the dependency of the preference. Since the assignment of the manager is dependent on the assignments of the secretaries and head of group, but not vice versa, it is highly recommended to assign the manager after them.

However, the same expression can be interpreted in a more simple way as "The manager should have a small central office". There are only four small offices and a minimal interpretation would be to accept only C5-116 and reject all others.
Next, the three heads of large projects are assigned a small office. It is remarked that: "The heads of large projects should be close to the head of group and secretariat." Again, this preference is dependent on the assignments of the head of group and the secretaries. The fact that the heads of large projects are assigned after the manager could be an indication that, in case of conflict, the manager should be closer to the head of group and secretaries than the heads of projects. This would mean that the manager preference has a higher weight than the heads of projects preference in the final ordering of solutions, as defined by the criterion.

There is no preference for assigning the heads of projects individually. No personal preferences of any of them are given. They are assigned to C5-113, C5-114 and C5-115.

Finally, the last eight researchers are assigned in pairs to large rooms. It is remarked that "there are really no criteria for the sequence of these twin-assignments". This is typical of a grouping strategy, see section 7.3 and chapter 8. Individuals are no longer assigned to a room, but pairs are assigned to large rooms. The order of individual assignments is therefore immaterial. The contribution of Schreiber [80] is the only one which makes use of this strategy.

The assignment of these researchers to the rooms still available, is a good example of a sub-problem. All other individuals already have been assigned an office and the assignment of the researchers is not allowed to change them. Hence, there is no longer any dependency between these researchers and any other individual, and vice versa.

It is remarked that smokers and non-smokers can not share a room ("the smoker/non-smoker conflict is a severe one"). We interpret this as a constraint: all matchings assigning a smoker and non-smoker to a room are unacceptable, all others are acceptable.

Also it is mentioned that researchers are not eligible for a single room. This is a simple unary constraint which limits the domain of all researchers to large rooms. In addition, "members of the same project should not share offices. Sharing with members of other projects enhances synergy (...)". The enhancement of synergy is also given as a reason for putting together researchers with different types of work. But the information about the kind of work of researchers in the problem statement is limited to a works-with relation and the name of a project.

This preference for enhancement of synergy can be represented as preferring minimal violations of a state of "perfect synergy", see figure 7.6. Such a preference is a nice example of the use of Partial Constraint Satisfaction [33]. One counts the number of violations of the ideal solution and prefers the matching with the smallest number.

In all large rooms

	two projects

In all large rooms

except one
two projects

In all large rooms

except two
two projects

... etc.

Figure 7.6
The left, most preferred, category contains those matchings in which all large rooms host researchers who work on different projects. The next preferred category contains matchings to which there is one exception to this. Matchings with two exceptions are in the third category etc.

In order to combine the preferences we can make use of the general combination function described in section 7.7. Numerical weights can be assigned to categories for preferences, each
weight expressing the strength of objection. The combination function adds all weights for each matching and orders them. The least objected matchings are solutions.

In this case some categories of preferences have precedence above categories of other ones. For example, the most preferred offices for the head of group are regarded as more important than the most preferred offices for researchers. This should be reflected in the distribution of weights. There is no need to assign weights to categories of constraints. This is so, because one category is (by definition) labelled unacceptable, therefore the interpretation of the matchings in the other category is always the same.

For the Sisyphus problem, there are several ways to assign weights. The problem with this example is that one should build a problem solving model based on one instance of solving a single problem configuration. This comes down to performing induction on the basis of a single example. In the Sisyphus case this can lead to a very simple problem solving model. As we have seen, one can safely interpret some preferences in a minimal way and represent them as constraints. In fact, the solution produced by the wizard is an element of every most preferred category of all the preferences. In this case there is no need to assign weights and an intersection of all the first categories of the preferences will produce the solution.

In general, one can state that the more categories the preferences have, the harder it is to come up with a nice distribution of weights. For the problem presented here, one may choose to represent the preferences as having just a few categories and still come up with a working model.

### 7.9.3 General characteristics

Some general characteristics of the Sisyphus office assignment problem have been summarized in table 7.1. The problem is a one-sided, bi-partite assignment problem. The subjects and resources are the employees and places in offices, respectively. There are several types of researchers, mainly based on their role in the organizational structure. This structure can be pictured as a hierarchy.

The order of the assignments by the wizard in the protocol, follows the organizational structure. The top of the hierarchy (head of group) is assigned first, the bottom (common researchers) last. It seems likely that preferences regarding the top of the organizational structure have precedence above those regarding lower personnel. But this is not explicitly formulated.

Both subjects and resources have attributes. Those of the subjects are explicitly given, while those of the resources (offices) must be inferred from the floor-plan. The offices (small and large) form a simple hierarchy. They have either a single or double capacity for holding subjects. However, the capacity depends on the role of the subjects in the organizational structure. Higher placed personnel should be given a large room for their own use.

Table 7.1 only lists the smoker/non-smoker requirement as a constraint. As we have seen, other preferences can be represented as a constraint too. However, we interpret the protocol in such a way that the smoker/non-smoker one is the only one that cannot be refined. Other preferences can be relaxed when needed, this one cannot.

There are dependencies in the preferences, as we have already discussed. The presence of a subproblem (the assignment of 8 researchers to 4 large rooms) has also been treated above.

Finally, the criterion can be described in terms of the simile of objecting judges. In that case one would prefer the least objected matching as a solution. This criterion may use numerical weights, but as we have seen, this is not strictly necessary. Of course, other criteria and combination functions are possible for this problem.
Table 7.1
Some general characteristics of the Sisyphus office assignment problem.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bipartite:</td>
<td>yes</td>
</tr>
<tr>
<td>One-sided:</td>
<td>yes</td>
</tr>
<tr>
<td>Subjects:</td>
<td>researchers</td>
</tr>
<tr>
<td>Resources:</td>
<td>places in rooms.</td>
</tr>
<tr>
<td>Types of subjects:</td>
<td>researcher, head of group, head of staff, secretary.</td>
</tr>
<tr>
<td>Hierarchy in subject types:</td>
<td>yes.</td>
</tr>
<tr>
<td>Attributes of subjects:</td>
<td>name, role, project, smoker, hacker, works-with.</td>
</tr>
<tr>
<td>Attributes of resources:</td>
<td>location on floor.</td>
</tr>
<tr>
<td>Types of resources:</td>
<td>offices, small offices, large offices.</td>
</tr>
<tr>
<td>Hierarchy in resource types:</td>
<td>yes.</td>
</tr>
<tr>
<td>Capacity of resource types:</td>
<td>dependent on type of subject and resource.</td>
</tr>
<tr>
<td>Constraints:</td>
<td>smoker/non-smoker.</td>
</tr>
<tr>
<td>Preferences:</td>
<td>central offices, synergy, putting different projects together.</td>
</tr>
<tr>
<td>Personal preferences of subjects:</td>
<td>none.</td>
</tr>
<tr>
<td>Dependencies in preferences:</td>
<td>one binary, two trinary (see text).</td>
</tr>
<tr>
<td>Subproblem:</td>
<td>grouping (see text).</td>
</tr>
<tr>
<td>Criterion:</td>
<td>Least objected solution.</td>
</tr>
<tr>
<td>Combination function:</td>
<td>Assigned numerical weights to categories</td>
</tr>
</tbody>
</table>

Inspecting the protocol, the wizard solving the problem, does not backtrack. One could see this as an indication that the problem is not as difficult as is claimed in the problem description [66], which states that the wizard is the only one who ever managed to solve the problem.

Another dubious claim in the problem description is the statement that all employees "have their personal preferences (...) that had better be observed" [57]. However, as we have seen, preferences are entirely based on the organizational role of the employees, not on any personal wish, as is used in SMPs. Inspecting the preferences it seems unlikely that any of them would change with another set of employees, provided the organizational structure and type hierarchy is preserved.

In this way elements from the ontology presented above can be used to analyze an assignment problem. It provides a way of structuring and representing a problem description and although we have not presented problem solving methods, solving the Sisyphus problem has become quite easy.

7.10 Discussion

In this chapter we have presented an ontology for assignment problems that can be used to structure problem descriptions of knowledge intensive assignment problems. The ontology provides insight regarding the nature of assignment problems and in what respects they can differ.

We have tried to capture the notions used in the description of CSPs and related problems, as well as in those used in SMPs and their variants. Central to the ontology is the idea of preference as an ordering of matchings. We have generalized this idea of Brewka et al. to characterize preference lists, and the use of categories.

The notion of preferential orderings does not apply to assignment problems only. In fact, many knowledge intensive problems make use of preferential reasoning. Preferences occur explicitly in
the knowledge representation of tasks like scheduling and design. From a formal point of view many types of non-monotonic reasoning can be seen as a form of preferential entailment. [86]. Therefore, some of the things claimed here, especially those regarding weak and strict orders, could be used in other problem types as well.

We did not discuss Problem Solving Methods (PSMs) for any of the problems treated here. A natural next step would be to classify known PSMs in accordance with our ontology. In this way, a task ontology can be used to organize and compare methods which can be used within the range of the given problem type.

As an illustration of how the ontology can be used to get some grip on a problem which should qualify as a knowledge intensive assignment problem, we have analyzed the Sisyphus I experiment. The use of the ontology certainly differs from the contributions to the Sisyphus experiment [57]. One of the main differences is that we started to give a formal interpretation of the notions involved in the ontology.

We do not claim that the ontology presented here is complete, and covers all variations of assignment problems. The ontology provides a clear analysis of assignment problems as identified by Puppe [71]. A next step would be the inclusion of an ontology of methods for assignment problem solving.
Formalizing group assignments

8.1 Introduction

An assignment task can be characterized by the goal of assigning a fixed number of elements (called subjects or components) to an equal or larger number of so-called resources [82, 71]. Examples of assignment problems are assigning persons to seats in rooms, airplanes to gates, passengers to seats etc. In each assignment problem the mappings from subjects to resources are subject to one or more constraints. In addition, solutions should be optimal with respect to a given number of preferences.

On a general level there are two strategies for solving assignment problems. The first is finding the required assignments by allocating individual subjects to individual resources. The other involves first forming groups of subjects, such that each group can be simultaneously assigned to resources. This grouping strategy will be the focus of this chapter.

In knowledge intensive assignment problems constraints often have the form of conditionals like: If subjects \(a\) and \(b\) (do not) both have property \(C\) then they should (not) be assigned to a resource of type \(D\). Such constraints make use of abstractions (e.g. properties and types) of both subjects and resources.

These constraints can be characterized in terms of equivalence relations. In this chapter we will give a formalization of the method of grouping subjects when constraints of this kind are used.

This chapter is structured as follows. First, we introduce grouping as a combinatorial advantageous method. This is followed by a description of abstractions in Knowledge Engineering. We then provide a formalization of abstraction in terms of partitions and equivalence relations. Next we introduce four basic types of constraints that make use of abstractions, and show how they reduce finding a grouping to the search for a particular partition. The final section provides some points for discussion.
8.2 Grouping

Knowledge-intensive assignment problems are often formulated in terms of types or features of subjects and resources. Similarly, constraints may contain general predicates over both subjects and resources, which can be characterized as abstractions.

Grouping is the method where the assignments themselves are done in terms of these general constructs, instead of individual subjects and resources. An example should clarify this process.

If persons are to be assigned to seats in rooms, then, instead of assigning each person to a seat individually, grouping proceeds as follows. Rooms contain one or more seats where persons can be assigned to. One can form groups of persons, similar to the distribution of seats over rooms. When this has been done, each group can be assigned to a room of the same size.

From a combinatorial point of view grouping is more advantageous than trying to assign each subject individually. Suppose that $X$ is a collection of $n$ distinct objects which have to be allocated to $k$ locations, such that location $i$ gets $n_i$ objects ($1 \leq i \leq k$), where $n_1 + n_2 + \ldots + n_k = n$. The total number of ways of allocating these objects is given by:

$$\frac{n!}{(n_1!)(n_2!)(n_k!)}$$

However, the number of ways of partitioning a set of cardinality $n$ consisting of $p_i$ subsets, each of cardinality $n_i$ ($1 \leq i \leq k$) where no two of the numbers $n_i$ is equal is given by:

$$\frac{n!}{(p_1!)(n_1!)(p_2!)(n_2!)(p_k!)(n_k!)}$$

The following example, taken from Balakrishnan [5], illustrates these formulas.

**Example 8.1:** The number of ways of allocating 43 persons into 7 different rooms such that the first two get 5 persons each, the next three get 6 persons each, the sixth room gets 7 persons, and the seventh 8 is:

$$\frac{43!}{(5!)(5!)(6!)(6!)(7!)(8!)}$$

When 43 persons have to be divided into 7 groups such that there are 5 persons in each of 2 groups, 6 persons in each of 3 groups, 7 persons in one group and 8 in one group, the number of ways is:

$$\frac{43!}{(2!)(5!)(5!)(3!)(6!)(6!)(7!)(8!)}$$

It is clear that grouping is advantageous when there are many groups of the same size. The reason for this lies in the fact that the assignment of individual subjects within a group are immaterial. This combinatorial advantage might be one of the reasons why human experts in assignment sometimes turn to a grouping method. For example, the think-aloud protocol of a human expert for the persons-to-rooms assignment task in the Sisyphus I experiment suggests the use of grouping [57]. In the contribution of Schreiber [81] to this experiment grouping is described and implemented.
8.3 Abstraction

Abstraction is an inference which is often mentioned in the description of problem solving methods. Clancey [17] distinguishes three types of data abstraction:

**Definitional abstraction**  Definitional abstraction involves abstraction based on "necessary features of a concept".

**Qualitative abstraction**  Qualitative abstraction involves the abstraction of quantitative data to some qualitative category.

**Generalization**  Generalization associates a general concept out of a concept-hierarchy with the abstraction data.

In Clancey’s description of the heuristic classification method abstraction plays a key role. In heuristic classification data is first abstracted. The abstracted concepts are heuristically linked by domain knowledge to other abstract concepts. As these concepts are part of a different conceptual hierarchy, refinement to more specific concepts can take place. This process is shown in figure 8.1.

![Figure 8.1](image)

Heuristic classification. Data is abstracted (indicated by the upward pointing arrow), heuristically linked (indicated by the horizontal arrow) and subsequently refined (downward pointing arrow).

One of the important features of heuristic classification is that reasoning takes place in terms of abstract constructs rather than individual entities. Protocols of human problem solving often show this feature.

Grouping can be seen in a similar way. Constraints are not formulated in terms of individuals but mention abstractions like types and properties. Groups of subjects are then allocated to groups of resources. A group mapping can be refined to one out of several possible mappings of individual subjects to individual resources.

8.4 Formalizing abstraction

Although abstraction plays an important role in Knowledge Engineering in general, and Problem Solving Methods in particular, its formulation has not been without controversy. As an example, consider the following characterization of abstraction [14]:
Formally, in abstraction the relation between the input concept and the output concept is that the latter contains less attributes than the former: "irrelevant" attributes are abstracted. What is relevant and what is irrelevant depends on a point of view.

Aben [1] has used this description to give a formalization of abstraction. He also notes that not all forms of abstraction can be seen as a removal of attributes of a concept. In heuristic classification, for example, abstraction is used to abstract observations. These are usually not represented as concepts.

Aben’s formalization is in line with the idea that abstraction involves the removal of irrelevant attributes. In fact, this approach is very reminiscent of pre-fregean, 19th century ideas of abstraction. However, since the time of Frege [6] relational accounts of abstraction have evolved, instead of those in terms of the ancient subject-predicate distinction.

One of the major problems with seeing abstraction as the procedure of removing irrelevant attributes is how to decide which attributes should be removed and which not. The notion of 'point of view' in the above quote summarizes this problem to a certain extent. However, it falls short of explaining the basic ideas behind abstraction.

A modern interpretation interprets abstractions in terms of equivalence relations and partitions. An equivalence relation \( \approx \) is a transitive, symmetric and reflexive, binary relation. When elements in a set are connected by an equivalence relation they are said to be part of the same equivalence class. The equivalence class of an element \( a \), under equivalence relation \( \approx \) is written as \([a]_\approx\) and is the set which contains all elements \( x \) such that \( a \approx x \). If \( b \approx a \), then \( a \) and \( b \) are in the same equivalence class and \([a]_\approx\) is the same set as \([b]_\approx\).

Equivalence relations can also be seen as partitions of a set \( A \). A set of subsets of \( A \) is called a partition of \( A \) if the subsets, called blocks do not overlap and their union results in the set \( A \).

The correspondence between equivalence relations and partitions is illustrated in figure 8.2. Blocks of a partition correspond to equivalence classes, since equivalence classes do not overlap and their union results in the complete set. If \( \approx \) is an equivalence relation over the elements of set \( S \), and \( P \) is a partition of \( S \), and each block of \( P \) corresponds to an equivalence class given by \( \approx \) then \( \approx \) is said to characterize the partition \( P \) of \( S \).

![Figure 8.2](image)

The correspondence between equivalence classes, defined by an equivalence relation on elements (on the left) and blocks of a partition of the same set (on the right). The depicted equivalence relation characterizes the shown partition. Picture taken from Landman [55].

The correspondence between partitions and equivalence relations is the key to understanding what abstraction is about. As an example consider the sentences in table 8.1. The sentences on the left-hand side make use of equivalence relations, like 'as tall as', 'as heavy as', etc. The sentences on the right-hand side are identity statements which compare abstractions, like 'length', 'weight'
etc. Both types of sentences are interchangeable, in the sense that they have the same meaning. The meaning of an abstraction is given by an equivalence relation over elements of a set.

<table>
<thead>
<tr>
<th>John is as tall as James</th>
<th>the length of John = the length of James</th>
</tr>
</thead>
<tbody>
<tr>
<td>John is as heavy as James</td>
<td>the weight of John = the weight of James</td>
</tr>
<tr>
<td>The set A has as many elements as the set B</td>
<td>the number of A's = the number of B's</td>
</tr>
</tbody>
</table>

Table 8.1
Sentences on the left make use of equivalence relations. On the right abstract properties are compared.

As an example consider a set $P$ of persons with an equivalence relation $\approx$ which we will interpret as $X$ is as tall as $Y$. The meaning of 'the length of $a$' is defined as the equivalence class of $a$ under $\approx$, which is $[a]_{\approx}$. In other words, the meaning of the length of $a$ consists of the set of all persons which are as tall as $a$.

Note that this modern notion of abstraction involves adding things together, rather than removing them. Abstracting means considering the equivalence classes of a set, rather than its elements.

Clancy's notion of qualitative abstraction can now be illustrated as follows:

**Example 8.2**: Consider the set of temperature values $\{36, 37, 38, 39, 40, 41\}$. One may choose to partition this set into three blocks: high fever: $\{40, 41\}$, fever: $\{38, 39\}$, no fever: $\{36, 37\}$. The relation $x$ is as high a fever as $y$ is the corresponding equivalence relation. The qualitative abstractions fever, high fever and no fever are the equivalence classes. Abstract reasoning would involve taking the equivalence classes, fever, high fever and no fever, as entities rather than the temperature values itself.

Other, more elaborated examples can be given, but the idea is always the same. Abstract classes or concepts are not formed by removing entities. Instead, the meaning of an abstract class is associated with its extension. As the temperature example makes clear, this description is much clearer than the idea of imposing a criterion for removing attributes.

Hence, every abstract property defines a partition of a set. Partitions will be quite important in the pages to come and therefore we will first present some formal preliminaries on partitions in general.

### 8.5 Partitions as formal structures

In this section we provide some basic, formal definitions. Readers familiar with the mathematics of partitions can skip this section.

We will first give a formal definition of a partition of a set:

**Definition 8.1**: A set $\pi = \{A_1 \ldots A_n\}$ of non-empty subsets $A_i \subseteq A$ is a partition of the set $A$ iff

- $\bigcup_{i=1}^{n} A_i = A$ (the $A_i$ are a covering of $A$)
- $A_i \cap A_j = \emptyset$ for all $A_i, A_j$ (they are mutually disjoint)

The subsets $A_i$ are called the blocks of the partition $\pi$ of $A$.

When we consider two different partitions of the same set then we can order them by a refinement relation.

**Definition 8.2**: Let $P$ and $Q$ be partitions of the same set $S$.

$P$ is a refinement of $Q$, $P \subseteq Q$, iff $\forall X \in P, \exists Y \in Q : X \subseteq Y$. Intuitively, if $P$ and $Q$ are
partitions of set $A$ and $P$ is a refinement of $Q$ then $P$ divides the set $A$ along the same lines as $Q$, and maybe some more. As an example consider the partition with only one block, namely $A$ itself. Every other partition of $A$ is a refinement of it. On the other extreme, consider the partition in which every element of $A$ is put in a different block. Hence, the blocks of this partition form singleton subsets. It is easy to see that this partition is a refinement of every partition of $A$.

When $P$ is a refinement of $Q$ ($P \subseteq Q$) then we will call $Q$ a **generalization** of $P$.

Two operations on partitions are **intersection** and **union**. Intersection is the easiest to define:

**Definition 8.3** : The **intersection** of partitions $P$ and $Q$ on set $A$ is given by:

$$P \cap Q = \{X \cap Y : X \in P \text{ and } Y \in Q \text{ and } X \cap Y \neq \emptyset\}$$

*Note, that the intersection $P \cap Q$ is a refinement of $P$ and $Q*.

Intuitively, the intersection of two partitions $P$ and $Q$ on the same set, is the partition where each block is a non-empty intersection of blocks of $P$ and $Q$. One could also think of the intersection in terms of equivalence relations. The intersection of two partitions corresponds to the conjunction of the corresponding equivalence relations, which is itself an equivalence relation. If $a$ is as tall as $b$ but both are not as tall as $c$, and $b$ smokes as much as $c$, but not as much as $a$, then the conjunction smokes as much and is as tall as results in three different equivalence classes for $a$, $b$, and $c$.

In a similar way as the intersection, one could define the union of two partitions as the disjunction of the two corresponding equivalence relations. Unfortunately, the disjunction of two equivalence relations is often not an equivalence relation. For example, suppose $a$ is as long as $b$, and $b$ smokes as much as $c$. Then this does not imply that $a$ is as long as $c$ or smokes as much as $c$. In other words the disjunction of two equivalence relations lacks transitivity, and therefore is not an equivalence relation. The solution is as simple as it is effective: one takes the **transitive closure** of the disjunction of the two equivalence relation [70]. The transitive closure $T(R)$ of a relation $R$ is the set which includes all elements of $R$ and whenever $(a, b)$ and $(b, c) \in T(R)$ then also $(a, c) \in T(R)$.

A different but equivalent definition is formulated entirely in terms of blocks:

**Definition 8.4** : The **union** of $P$ and $Q$ ($P \cup Q$) can be obtained as follows: $Z$ is the unification of non-empty set of blocks $X$ of $P$ and $Y$ of $Q$ iff $Z = \cup X = \cup Y$. All unifications of $P$ and $Q$ can be ordered by set inclusion. The union of $P$ and $Q$ is the set of minimal unifications of both. Both $P$ and $Q$ are refinements of this partition. See Landman [55] for details and an introductory example.

Intuitively, the union of two partitions $P$ and $Q$ is the minimal way of removing lines from $P$ and $Q$ such that they are both refinements of the resulting partition.

The intersection and union operations, $\cap, \cup$, can be used to combine abstract properties. For example, the property of smoking and working on the same project defines a partition of a set of persons which is defined by the intersection of the partition for smoking and the one for working on the same project. The disjunction of the properties may cause problems as we have shown above. The union operation will result in the transitive closure of the properties which may not correspond with the intended meaning. In that case the disjunction of the properties does not lead to an equivalence relation and can no longer be considered as an abstract property. In other words, the union of two abstract partitions is only applicable if the disjunction of the two abstractions is itself an abstraction.

**Definition 8.5** : Let $P$ be a partition of the set $S_1$, and $Q$ a partition of the set $S_2$. The function $h : S_1 \rightarrow S_2$ is an **isomorphism** between $P$ and $Q$ iff:
- $h$ is bijective (or one-on-one).
- For all $s_i, s_j \in S_1$: If $s_i$ and $s_j$ are together in a block of $P$ then $h(s_i)$ and $h(s_j)$ are together in a block of $Q$.
- For all $s_i, s_j \in S_1$: If $h(s_i)$ and $h(s_j)$ are together in a block in $Q$ then $s_i$ and $s_j$ are together in a block in $P$.

In general an isomorphism is a bijective function between two structures such that the relations and operations are preserved and anti-preserved. In the case of isomorphic partitions this means that the equivalence relation is preserved and anti-preserved.

As we have seen, the blocks of a partition correspond to equivalence classes under a certain equivalence relation. An isomorphism on partitions can then also be seen as a one-to-one mapping which preserves (second item in the definition) and anti-preserves (third item in the definition) this relation.

It is not difficult to see that when two partitions are isomorphic they have the same number of blocks and each block from one can be mapped to an equal sized block from the other. The reverse is also true: If every block from one partition can be mapped to a block of another partition with the same size, the two are isomorphic. In the pages to come we will refer to isomorphic partitions as sharing the same form. By the form of a partition we mean the number of blocks and the block sizes. Note, that this notion is not found in the literature, since it is our own.

8.5.1 A partition lattice

When discussing grouping we will make use of the fact that all partitions of a set, ordered by the refinement relation, form a lattice. We will define and explain this notion here. Readers familiar with lattices can safely skip this section.

Let $A$ be a set and $\leq$ be a partial order (a transitive, reflexive and antisymmetric relation). The structure $A = \langle A, \leq \rangle$ is called a partial ordered set, or poset for short. If $A$ has an element $O$ such that $O \leq x$, for all $x \in A$ then $O$ is called the null element or bottom element. Dually, if $A$ has an element $I$ such that $x \leq I$, for all $x \in A$ then $I$ is called the universal element or top element.

If $x$ and $y$ are elements of $A$ then an element $u$ is called a least upper bound (lub) if:

(a) $u \geq x$ and $u \geq y$ ($u$ is an upper bound of $x$ and $y$)
(b) $w \geq x$ and $w \geq y \Rightarrow w \geq u$

Dually, we can define the greatest lower bound (glb) as an element $v$ such that:

(a) $v \leq x$ and $v \leq y$ ($v$ is a lower bound of $x$ and $y$)
(b) $w \leq x$ and $w \leq y \Rightarrow w \leq v$

Both the lub and glb of a pair $x$ and $y$ are unique if they exist. Therefore one can define two operations $+$ and $\cdot$ as follows: $\text{lub}(x, y) = x + y$ and $\text{glb}(x, y) = x \cdot y$.

**Definition 8.6**: (First Definition) A lattice $L$ is a poset $L = \langle L, \leq \rangle$ in which every element $x$ and $y$ has a lub and a glb.

As an example of a lattice consider the powerset $\wp(A)$ of some set $A$. $\langle \wp(A), \subseteq \rangle$ (where $\subseteq$ is the set inclusion relation) is a lattice. The intersection $\cap$ corresponds to the $\cdot$ operation and union
∪ to the + operation. This lattice has a bottom and a top element, namely ∅ and A. It is clear that ∅ ⊆ x and x ⊆ A for all x ∈ P(A).

Another example of a lattice is the set of all partitions of a set, ordered by the refinement relation ⊆. If A is a finite set and II is the set of all the partitions on A then (II, ⊆) is a lattice. The lub and glb operations are ∪ and ⊓ respectively. The top and bottom element have been introduced above: Every partition is a refinement of A (the top) and the singleton block partition (the bottom) is a refinement of all partitions on A.

A lattice can be conveniently pictured as a graph. The elements of the lattice form the nodes and a downward path from x to y denotes x ≤ y. As an example consider Figure 8.3 in which the partition lattice of the set \{a, b, c, d\} is shown.

![Figure 8.3](image)

The lattice of partitions on the set \{a, b, c, d\} ordered by the refinement relation ⊆. P ⊆ Q is depicted by a downward line from P to Q.

The top element I is shown at the top of the picture, and contains as it's only block the set \{a, b, c, d\} itself. We will use the following notation for partitions: We write elements in the same block as a string (in which the order is immaterial) and use spaces to separate blocks. Hence the I partition is written as abcd. The bottom element O is shown at the bottom and consists of the four blocks a b c d.

To find the union of the two partitions one locates them in the figure and follows a line upward to the node where they first meet. For example, the union of ab cd and ac bd (depicted on the extreme left and right in figure 8.3, respectively) is the I partition. To find the intersection of two partitions one follows a downward line. Hence, ab cd ∩ ac bd = O. Notice that when two partitions, like a b cd and ab cd, are both on a downward (or upward) path, then one is a refinement of the other (a b cd ⊆ ab cd).
An alternative, but equivalent definition of a lattice is to view the structure as an algebra. We will not prove that the two definitions are equivalent, see for example Rutherford [78] for a proof.

**Definition 8.7:** *(Second definition)* A lattice is a triple \( \langle A, +, \cdot \rangle \), where \( A \) is a set and + and \( \cdot \) are two binary operations such that the \( L \)-laws in table 8.5.1 are valid for elements \( x, y, z \in A \).

\[
\begin{align*}
L_1 : x \cdot y &= y \cdot x & L_{1+} : x + y &= y + x \\
L_2 : x \cdot (y \cdot z) &= (x \cdot y) \cdot z & L_{2+} : x + (y + z) &= (x + y) + z \\
L_3 : x \cdot (x + y) &= x & L_{3+} : x + (x \cdot y) &= x \\
D : x \cdot (y + z) &= x \cdot y + x \cdot z
\end{align*}
\]

**Table 8.2**
The commutative (first row), associative (second row) and absorptive laws which are valid in any lattice. The law \( D \) (distributivity) is not valid in all lattices.

Hence, the powerset lattice mentioned above denotes an algebra \( \langle \mathcal{P}(A), \cup, \cap \rangle \) and the partition lattice the algebra \( \langle \Pi, \cup, \cap \rangle \). In all lattices the following three conditions are equivalent:

\[
x + y = y \quad x \cdot y = x \quad x \leq y
\]

And so in particular for partitions \( x, y \):

\[
x \cup y = y \quad x \cap y = x \quad x \subseteq y
\]

The reader can verify that the \( L \) laws hold for the \( \cup \) and \( \cap \) operations on partitions, where instead of + one reads \( \cup \) and instead of \( \cdot \) one reads \( \cap \).

However, not all lattices have the same properties. For example, all powerset lattices are distributive which means that the law \( D \) in table 8.5.1 is valid.

Partition lattices are not distributive (which we will not prove). This is important because it means that the corresponding algebra does not have a complement operation.

**Definition 8.8:** If a lattice \( \langle A, +, \cdot \rangle \) has a top \( I \) and bottom element \( O \) than \( x \) is a complement of \( y \) (and \( y \) of \( x \)) iff \( x \cdot y = O \) and \( x + y = I \) for \( x, y \in A \).

In a powerset lattice \( \langle \mathcal{P}(A), \cup, \cap \rangle \) with finite \( A \), the complement of a set \( x \in \mathcal{P}(A) \) is the set difference \( A \sim x \). So, if \( A = \{1, 2, 3, 4\} \) and \( x = \{1, 2\} \) the complement \( y \) of \( x \) is \( \{3, 4\} \). Here, as in any distributive lattice, the complement of \( x \) is an unique element \( y \), and therefore corresponds to an operation (set-difference). Because a partition lattice lacks distributivity the complement of a partition is not necessarily unique, and therefore does not correspond to an operation (a function). We will see examples of how a partition can have more than one complement later on.

### 8.6 Constraints and abstractions

The subject of this chapter is the application of the mathematics of partitions to the use of abstractions in constraints in assignment problems. We will show how a grouping problem can be seen as searching the partition lattice. In particular we will limit ourselves to the analysis of four general constraints which make use of abstractions.

In the following we assume that the number of resources and subjects are equal. We will consider four elementary formulations of such constraints. As an introductory example, consider the following constraints:
- People working on the same project must share rooms.
- Smokers and non-smokers are not allowed to share the same room.
- People working on the same project are not allowed to share a room.
- People working on different projects must share rooms.

These constraints make use of equivalence relations such as "sharing the same room", "working on the same project". They define partitions on subjects and resources. For example, the "room"-partition partitions all places into blocks, where each block corresponds to a room. In the same way calling every individual a smoker or a non-smoker partitions the set of subjects (persons) in two. The corresponding equivalence relation could be formulated as: "x is as much a smoker as y".

Also, these constraints demand that all subjects which share or differ in some abstract property should be assigned a resource with the same/different abstract property. The four example constraints above, can be seen as representing four basic types of constraints.

Let x, y be different subjects which should be assigned to resources. Let f(x) and f(y) be the resources assigned to x and y under the assignment f. The constraints we are considering all mention a partition of subjects and one on resources. Let \( \approx_S \) be the equivalence relation on subjects, which corresponds to the partition of subjects mentioned in the constraint, and \( \approx_R \) the equivalence relation corresponding to the partition of resources, also mentioned in the constraint.

The basic types of constraint can then be formulated as follows (for all subjects x, y):

\[
\begin{align*}
(C_1) \quad [x]_{\approx_S} & = [y]_{\approx_S} \Rightarrow [f(x)]_{\approx_R} = [f(y)]_{\approx_R} \\
(C_2) \quad [x]_{\approx_S} & \neq [y]_{\approx_S} \Rightarrow [f(x)]_{\approx_R} \neq [f(y)]_{\approx_R} \\
(C_3) \quad [x]_{\approx_S} & = [y]_{\approx_S} \Rightarrow [f(x)]_{\approx_R} \neq [f(y)]_{\approx_R} \\
(C_4) \quad [x]_{\approx_S} & \neq [y]_{\approx_S} \Rightarrow [f(x)]_{\approx_R} = [f(y)]_{\approx_R}
\end{align*}
\]

Note that \( [x]_{\approx} = [y]_{\approx} \) is equivalent to saying that x and y are in the same block of the partition under the equivalence relation \( \approx \), i.e. they belong to the same abstract class, or share the same abstract property.

Grouping is the method where the goal is to find a partition (or grouping) of subjects which has the same form as the partition of resources. In other words: Grouping involves finding a partition of subjects which is isomorphic to the partition of resources.

When we have a found such a grouping, which has the same number of blocks, each of the same size as a block of resources, then any block of subjects can be easily assigned to a block of resources. This assignment of blocks of subjects to blocks of resources is simple: For a block of subjects find a block of resources of the same size.

The four elementary constraint types \( C_1 \ldots C_4 \), can be reformulated for grouping. Instead of \( [f(x)]_{\approx_R} = [f(y)]_{\approx_R} \) in the above formulation of the above constraint types, we can read \( [x]_{\approx_G} = [y]_{\approx_G} \). Here \( \approx_G \) is the equivalence relation of the wanted grouping on subjects, which is isomorphic to the equivalence relation on resources \( \approx_R \).

The four constraints types can now entirely be formulated in terms of partitions on the set of subjects S.
We will call constraints of type \( C_1 \) \textit{together—together} constraints. It states that if \( x \) and \( y \) are together in a block of the partition \( P_{\approx_S} \) (characterized by \( \approx_S \)) on the set of subjects \( S \), then they should also be together in the grouping \( P_{\approx_G} \) (the partition of \( S \) characterized by \( \approx_G \)).

Constraints of type \( C_2 \) state the reverse of \( C_1 \) and will be called \textit{not-together—not-together} constraints. \textit{Together—not-together} constraints are of type \( C_3 \). \( C_3 \) expresses that subjects \( x \) and \( y \) which occur in one block should be separate in the grouping partition.

Finally, the \textit{not-together—together} constraints are the ones described by \( C_4 \). It states that all subjects which are separated must be put together in the grouping. Notice that this last constraint is not very meaningful. If there are subjects in different blocks, it can (only) be satisfied by putting all subjects together in one block. One might object that the natural language formulation, given at the start of this section, is wrongly translated into formal language. In that case, one will find that the wanted formalization is among the other three. We will provide a more thorough account in section 8.9.4.

The constraint types \( C_1 \ldots C_4 \) can be equivalently formulated as follows:

\begin{align*}
(C_1) & \quad P_{\approx_S} \subseteq P_{\approx_G}. \\
(C_2) & \quad P_{\approx_G} \subseteq P_{\approx_S}. \\
(C_3) & \quad P_{\approx_S} \cap P_{\approx_G} = O. \\
(C_4) & \quad P_{\approx_S} = I \text{ or } P_{\approx_G} = I.
\end{align*}

(In \( C_3 \) \( O \) is the partition with singleton sets as blocks, the bottom element of the partition lattice. In \( C_4 \) \( I \) is the top element and is the partition with one block.)

We will show that these formulations are indeed equivalent to the previous formulations of \( C_1 \) to \( C_4 \). To see the validity of \( C_1 \) and \( C_2 \) we will prove the following:

\textbf{Theorem 8.1} : Consider two partitions \( P \) and \( Q \) on the same set \( S \) and the following holds:

If \( x, y \) are together in a block of \( P \) then they are together in a block of \( Q \). We claim that in that case \( P \) is a refinement of \( Q \), \( P \subseteq Q \).

\textbf{Proof} : Suppose not, then there must be a block \( B \) in \( P \) which is not a subset of any block in \( Q \). This block \( B \) must contain more than one element. Then there must be elements \( b_1 \) and \( b_2 \) together in \( B \) which are not together in a block in \( Q \). Contradiction.

From this we can see that \( C_1 \) can be expressed as \( P_{\approx_S} \subseteq P_{\approx_G} \). Regarding \( C_2 \), notice that \( [x]_{\approx_S} \neq [y]_{\approx_S} \Rightarrow [x]_{\approx_G} \neq [y]_{\approx_G} \) is equivalent to: \( [x]_{\approx_S} = [y]_{\approx_G} \Rightarrow [x]_{\approx_S} = [y]_{\approx_S} \). This proves the validity of the above formulation of \( C_2 \).

\( C_3 \) is equivalent to the requirement that the result of the intersection of the two partitions on subjects is a partition where each block contains exactly one element, and the disjunction of the two partitions yields the partition with one block: the set of subjects itself. We will prove the
following theorem.

**Theorem 8.2**: Let $S$ be a finite set and $O$ the singleton block partition of $S$. Let the equivalence relations $\approx_S$ and $\approx_G$ both define partitions on $S$, denoted by $P_{\approx_S}$ and $P_{\approx_G}$ respectively.

$[x]_{\approx_S} = [y]_{\approx_S} \rightarrow [x]_{\approx_G} \neq [y]_{\approx_G}$ for different $x,y \in S \Leftrightarrow P_{\approx_S} \cap P_{\approx_G} = O$

**Proof**: ($\Rightarrow$) By contraposition, suppose $[x]_{\approx_S} = [y]_{\approx_S} \rightarrow [x]_{\approx_G} \neq [y]_{\approx_G}$ holds, and there is a block in the intersection of $P_{\approx_S}$ and $P_{\approx_G}$ with more than one element. We denote two (different) elements of this block by $a$ and $b$. Hence $a$ and $b$ have the same equivalence class in both $P_{\approx_S}$ and $P_{\approx_G}$, and in $P_{\approx_G}$, otherwise they would not appear in the same block in the intersection. And hence $[a]_{\approx_S} = [b]_{\approx_S}$ and by assumption $[a]_{\approx_G} \neq [b]_{\approx_G}$. Contradiction.

($\Leftarrow$): Suppose $P_{\approx_S} \cap P_{\approx_G}$ yields a partition with only blocks of exactly one element. Assume by contraposition, for some different $x, y$, $[x]_{\approx_S} = [y]_{\approx_S}$ and $[x]_{\approx_G} = [y]_{\approx_G}$.

$[x]_{\approx_S}$ contains $x$ by reflexivity. $[x]_{\approx_S}$ contains at least a second element $y$ because $[x]_{\approx_S} = [y]_{\approx_S}$ and $x \neq y$. The same reasoning for $[x]_{\approx_G}$ and $[y]_{\approx_G}$ yields that $x, y$ are also two different elements in these blocks. Hence $x, y \in [x]_{\approx_S} \cap [x]_{\approx_G}$. So there is a block in the intersection of $P_{\approx_S}$ and $P_{\approx_G}$ with more than one element. Contradiction.

### 8.7 The nature of grouping

In summary, the constraints we have described are all of the form:

If subjects $x$ and $y$ are (not) together in a block in partition $P$ then they should be assigned to a block such that they are (not) together in partition $Q$.

The original partition of subjects ($P$ in the above formulation) is given. In addition the form of the grouping partition is also known. It is equal to the partition of the assignments ($Q$ in the above formulation). Finally, the relation between the partition of subjects and the grouping is given by the constraints, as presented in the previous section. The grouping partition (on subjects) itself, is unknown.

For example, a *together*—*together* constraint presents us with a partition of subjects $S$, $P_{\approx_S}$, an unknown grouping (partition of $S$) $P_{\approx_G}$ of which we only know the number of blocks and the block sizes, and the relation $P_{\approx_S} \subseteq P_{\approx_G}$. The problem is to find a grouping (partition) of the proper form which satisfies the relation.

We can formulate this problem in a partition algebra $\langle \Pi(S), \sqcup, \cap \rangle$ (where $\Pi(S)$ denotes all the partitions on $S$). Remember that $P_{\approx_S} \subseteq P_{\approx_G}$ is equivalent to $P_{\approx_S} \cap P_{\approx_G} = P_{\approx_S}$. Hence the problem is to find $P_{\approx_G}$ of which we know the number and sizes of blocks such that $P_{\approx_S} \cap P_{\approx_G} = P_{\approx_S}$ holds for a given $P_{\approx_S}$.

As a simple example consider four people $a, b, c, d$ of which $a$ and $b$ work on the same project and $c$ and $d$ each on a different project. Working on the same project then imposes a partition, namely $ab \ c \ d$ on the set of four persons. Suppose that we have a *together*—*together* constraint which states that persons working on the same project should share the same room. The rooms,
and the places in them, provide us with the form of the wanted grouping. In this example there are two rooms, each with two places.

To satisfy the constraint we should look for a partition with two blocks of size two, of which \(ab\ c\ d\) is a refinement. Inspecting Figure 8.3 one quickly sees that there is only one such partition, namely \(ab\ cd\). This grouping satisfies the constraint.

Hence, we can make use of the partition lattice as a search space. We are given a node in the graph, a specification (form) of another and a relation (the constraint type) between the two which should be satisfied.

Another view is to look at the partition lattice as an algebraic structure which is used in a 'reverse' way. The standard, or 'forward' manner of using an algebra is to perform operations on certain, given elements. In the case of a partition algebra one performs intersection and union operations on given partitions.

However, in this case one is given only one partition (of subjects) and the form (a partition of resources) of another. The objective here is to fill in the unknown partition by applying operations in a restricted way. Hence, in a sense, one uses the algebraic structure in a 'reverse' manner.

This situation is reminiscent of performing abduction in logic. Formally, in inductive logic one has a formula \(\beta\), called the observation, and a theory \(\Theta\) and one looks for a formula \(\alpha\), called an explanation, such that \(\Theta, \alpha \vdash \beta\). So, instead of using 'forward' deduction, one tries to reason backwards from the conclusion to the hypothesis.

Deductive reasoning in logic involves the application of operations on the formulas in the theory in order to arrive at a conclusion. When performing abductive one starts with the conclusion and tries to find a missing formula. Similarly, instead of performing straight, 'forward' reasoning in the partition algebra, the reasoning is 'backward', and one searches for the identity of a partition which satisfies the statement.

### 8.8 Merging and splitting blocks

When one looks at a partition lattice like for example the one in Figure 8.3 one sees that the number of blocks are the same for all partitions at each level of the lattice. By the level of the lattice, we mean all nodes in the graph with an equal distance from the top/bottom.

The number of blocks of the top element of the lattice \(I\) is one. And the number of blocks of the bottom element \(O\) equals the number of elements in the set \(S\), written as \(|S|\).

In Figure 8.3 \(O\) has four blocks, \(I\) one and their difference in size is \(4 - 1 = 3\). This number also indicates the distance between the two nodes in the graph. As another example, consider the partition \(ab\ cd\) which has two blocks and is at a distance of one from \(I\) and two from \(O\).

In general, if we start at a certain node in a partition lattice then moving down one node in the graph increases the number of blocks by one. Going up decreases the number of blocks. There are some useful properties between the number of blocks of partitions and their refinement relation. First, if \(P \subseteq Q\) then \(P\) has more blocks than \(Q\). Second, if \(P\) has \(n\) blocks and \(Q\) \(m\) and \(P \subseteq Q\), such that \(n > m\), then \(P\) can be reached from \(Q\) by a \(n - m\) length downward path in the lattice.

When one partition \(P\) is a refinement of another partition \(Q\), then \(Q\) can be constructed from \(P\) as follows: Choose two blocks from \(P\) and merge them together into a single block (which is their union). Continue to merge blocks either chosen from \(P\) or obtained by successive merges until \(Q\) is obtained. Note, that in order to obtain \(Q\) from \(P\) one has to make the right choices of which blocks to merge.
Also, \( P \) can be constructed from \( Q \) by the successive splitting of blocks from \( Q \). By splitting a block we mean splitting it in two. Obviously, for blocks with more than two elements, there are several ways of doing this.

Moving down the lattice one level deeper involves the splitting of a block in two. We call this procedure refining the partition. Moving up one level involves merging two blocks and we speak of generalizing the partition. So, by generalizing and refining one can move through the lattice. The search for a particular grouping as specified by the constraints which we have defined thus far, can thus be viewed as a search through the lattice. This search is limited by the distance of the two partitions, which can be computed if the number of blocks is known.

These observations can be used in the search for a grouping partition. In the following section we will treat the constraint types separately and provide examples.

### 8.9 Finding groupings for the constraints

We will treat the three constraint types separately now. Providing examples, we will show how the partition lattice serves as a search space for the problem of finding the grouping. The formal structure of the problem can be used to control the search.

#### 8.9.1 Together—together constraints

As we have seen, together—together constraints express that the given partition of subjects should be a refinement of the wanted grouping, of which only the form is known \( (P \cong C) \). This property can help us to find the desired partition in instances of assignment problems. Here, we can make use of the fact that when \( P, Q \) are partitions and \( P \subseteq Q \), then \( Q \) can be obtained from \( P \) by merging blocks from \( P \).

**Example 8.3**: Six persons \( C = \{a, b, c, d, e, f\} \) have to be assigned to six places. All persons work on projects \( \{p_1, p_2, p_3\} \). Person \( a \) works on project \( p_1 \), \( b \) on \( p_2 \) and \( p_3 \), \( c \) on \( p_2 \), \( d \) on \( p_1 \), \( e \) and \( f \) on \( p_3 \). Notice that the equivalence relation on the set of subjects \( S \) "\( x \) works on the same projects as \( y \)" partitions the set of persons into four blocks: \( P \cong S = \{a, d, b, c, e, f\} \). This is depicted on the left side of figure 8.4.

The following constraint must hold: all persons who work on the same project(s) must share the same room. Consider a situation where there are 4 rooms with 1 seat and 1 room with 2 ((a) in figure 8.4). This problem has no solution. The form of the desired grouping partition of persons \( P \cong C \) is the same as the given room partition. We denote this form by \( X.X.X.X.X.X \) where each \( X \) is an (yet unknown) element of \( S \) and each sequence of \( X \)'s a block (see (a) in figure 8.4). The number of blocks of this form is 5. The projects partition \( P \cong S \) must be a refinement of \( P \cong C \), and therefore should have more blocks. But \( P \cong S \) has 4 blocks. Hence, the problem can not be solved.

Consider another situation where there are 3 rooms with 2 places each ((b) in figure 8.4). This partition has 3 blocks and hence the problem is solvable. By merging blocks of \( P \cong S \), we can obtain the desired form. This can be done by \( 4 - 3 = 1 \) merges. It is easy to see that this merge involves the single element blocks \( \{b\} \) and \( \{c\} \).

Finding a grouping which will solve a together—together constraint can thus be formulated as trying to construct a partition \( P \cong C \) of a certain form (sizes and number of blocks), by the successive merging of blocks, starting with \( P \cong S \). If the number of blocks of the grouping \( P \cong C \) is
Fig. 8.4
The partition $P_{\text{eq}}$ must be a refinement of the goal partition $P_{\text{eq}}$. The figure shows two problems ((a) and (b)). In (a) the elements together in a block in $P_{\text{eq}}$ must be assigned such that they end up together in a block of $P_{\text{eq}}(a)$. The problem (b) uses $P_{\text{eq}}(b)$ instead of $P_{\text{eq}}(a)$.

$m$ and that of $P_{\text{eq}}$, the partition on subjects, is $n$ then $P_{\text{eq}}$ can be constructed from $P_{\text{eq}}$ in $n - m$ merges.

Assuming $n > m$, a greedy algorithm can be used to find the wanted grouping. Let $P_{\text{eq}} = \{b_1 \ldots b_n\}$, with block sizes $|b_1| \ldots |b_n|$. Let $x_1 \ldots x_m$ be the sizes of the blocks in the wanted partition $P_{\text{eq}}$. Starting with the largest number $x_i$ we try to find blocks $b_j \ldots b_k$ such that $x_i = |b_j| + \ldots + |b_k|$. If this fails the algorithm ends with no solution. Else we merge the blocks, obtaining the first block of the wanted grouping $P_{\text{eq}}$. The total number of blocks merged so far should not exceed $n - m$. Next, the blocks $b_j \ldots b_k$ are marked as used so that they will not be used again. This procedure can be repeated for every $x_i$ in decreasing order, using only unmarked blocks from $P_{\text{eq}}$.

That this algorithm is correct follows from the analysis of partitions presented so far. The algorithm is greedy [13] because we can start with the largest block size of the grouping and try to fill it by one or more blocks of $P_{\text{eq}}$. That we can do this follows from the fact that we can only merge blocks. Hence in order to fill the largest block of the grouping it must consist of merged blocks of $P_{\text{eq}}$.

The marking of blocks prevents the blocks of being used more than once in a merge. This has to be prevented otherwise elements could end up in the grouping more than once.

### 8.9.2 Not-together—not-together

Not-together—not-together constraints express that if $x$ and $y$ have different abstract properties they should be assigned to resources of a different type. This states that the goal partition of subjects $P_{\text{eq}}$ should be a refinement of the given partition of subjects $P_{\text{eq}}$: $P_{\text{eq}} \subseteq P_{\text{eq}}$. This is similar to together—together constraints, except that the partitions on both sides of $\subseteq$ are switched. Similar to the together—together constraints this property can help us find the desired grouping. Instead of moving up the lattice, starting from $P_{\text{eq}}$, the search proceeds downwards.
The grouping can be obtained by refining the partition of subjects $P_{\geq G}$. Refining a partition means splitting one or more of the blocks. Here we make use of the fact that if $P$ and $Q$ are partitions and $P \subseteq Q$ then $P$ can be obtained from $Q$ by successive splitting of blocks, starting with the ones in $Q$.

If the partition on subjects $P_{\leq S}$ does have more blocks than the grouping $P_{\geq G}$, the problem is unsolvable, for blocks of $P_{\leq S}$ should be split to arrive at $P_{\geq G}$. The greedy algorithm to find the wanted grouping as presented for together-together constraints can be adapted for the not-together—not-together case, by just switching the roles for $P_{\leq G}$ and $P_{\leq S}$.

As above, let $P_{\leq S} = \{b_1 \ldots b_n\}$, with block sizes $|b_1| \ldots |b_n|$, and $x_1 \ldots x_m$ be the sizes of the blocks in the wanted partition $P_{\geq G}$.

Starting with the largest block $b_i$ we try to find numbers $x_j \ldots x_k$ such that $|b_i| = x_j + \ldots + x_k$. If this fails the algorithm ends with no solution. Else we split the block $b_i$ in such a way that we obtain blocks with sizes $x_j \ldots x_k$. Every extra block counts as a split, and the total number should not exceed $m - n$, where $m$ is the block size of $P_{\leq G}$, and $n$ of $P_{\leq S}$. The procedure proceeds exactly as in the together-together case.

### 8.9.3 Together—not-together

**Together—not-together** constraints express that the intersection of $P_{\leq S}$ and $P_{\geq G}$ yields the bottom partition $O$ of which each block contains exactly one element: $P_{\leq S} \cap P_{\geq G} = O$.

More directly, as the name indicates a together—not-together constraint states that when two elements are in a block of the given partition they should not be together in a group. We will describe a procedure which will take as input a partition $P$ and produce as output a partition $Q$ such that every two elements which are together in a block in $P$ are not together in a block in $Q$.

We will call this procedure **distribution**, and the result $Q$ a distribution of $P$. Figure 8.5 shows an example of a simple distribution from one partition to another.

![Figure 8.5](image)

*Figure 8.5*

A partition $P$ of 5 elements in two blocks, is distributed over a partition of three blocks. Notice that each two elements which are together in a block of $P$, end up in a different block in the distribution.

From the input partition $P$ we take the largest block. All elements of the largest block in $P$ must be in a different block in the resulting partition $Q$. Hence, the distribution $Q$ will have as much blocks as the largest block size of $P$. The elements of the other blocks in $P$ can likewise be distributed over the blocks in $Q$. After all elements have been distributed this way all elements
that were together in a block in $P$ are not together in any block of $Q$. Hence, the distribution $Q$ of $P$ satisfies the $\text{together} \rightarrow \text{not-together}$ constraint.

The distribution $Q$ of a partition $P$ is a complement of $P$. Remember that saying that $Q$ is the complement of $P$ is equivalent to saying that $P \cap Q = O$ and $P \cup Q = I$ (where $O$ is the bottom, $I$ the top of the partition lattice).

In the distribution $Q$ of a partition $P$, elements are apart which were originally together in a block in $P$. Hence $P \cap Q = O$ holds when $Q$ is a distribution of $P$. Proving $P \cup Q = I$ involves showing that all elements are either together in a block of $P$ or of $Q$, or are in the transitive closure of this disjunction. Note that the number of blocks of $Q$ is equal to the largest block size in $P$. Let the largest block of $P$ be $B$. Then in $Q$ every element is either together in a block with an element from $B$ or is itself an element of $B$. From this it follows that the union of $P$ and $Q$ results in the single block partition.

The distribution procedure is clearly indeterministic. There can be various ways to distribute the elements of a partition and equivalently there can be more than one complement of a partition. This fact hampers the search of a solution for a $\text{together} \rightarrow \text{not-together}$ constraint.

After distribution of a partition $P$, in the result $Q$ all the elements which were together in a block in $P$ are not together in a block in $Q$. Note however that $\text{together} \rightarrow \text{not-together}$ does not express that $P_{\geq G}$ should be a complement of $P_{\leq S}$. Instead it says that $P_{\geq G}$ should be a refinement of a complement of $P_{\leq S}$.

So when $Q$ is the result of a distribution of $Q$ we can safely split some blocks from $Q$ without having to fear that elements which were together in a block in $P$ will end up together in a block again. Hence, a distribution of $P_{\geq G}$ can safely be refined without disturbing $\text{together} \rightarrow \text{not-together}$, however we can not merge any blocks in it. Merging blocks would mean putting elements together which should be separated, whilst splitting blocks will leave the constraint intact. In summary we can say that $\text{together} \rightarrow \text{not-together}$ constraints express that $P_{\geq G}$ should be a refinement (not necessarily a proper refinement) of a complement of $P_{\leq S}$.

The form of $P_{\geq G}$ is given. We can again use properties of the form to check whether the problem is solvable or not. The minimal number of blocks in the distribution equals the maximum block size of the given partition. This leads to the following result:

**Theorem 8.3**: A together$\rightarrow$not-together constraint is unsolvable if (and only if) the number of blocks of $P_{\geq G}$ is less than the largest block size in $P_{\leq S}$.

**Proof**($\Rightarrow$) By contraposition. Suppose a together$\rightarrow$not-together constraint is solvable, then $P_{\geq G}$ is a refinement of the complement of $P_{\leq S}$. Any complement of $P_{\leq S}$ has $n$ blocks, where $n$ is the number of the largest block size in $P_{\leq S}$. $P_{\geq G}$ is a refinement of a complement of $P_{\leq S}$ and so as at least as many blocks.

($\Leftarrow$) By contraposition. Suppose a together$\rightarrow$not-together constraint is unsolvable but the number of blocks of $P_{\geq G}$ is equal or greater than the largest block size in $P_{\leq S}$. The together$\rightarrow$not-together constraint is unsolvable so $P_{\geq G}$ is not a refinement of a complement of $P_{\leq S}$. However, the number of blocks of $P_{\geq G}$ is equal or greater than the largest block size in $P_{\leq S}$, and so we can form a distribution of $P_{\leq S}$ and take 0 or more steps to refine it to arrive at the form of $P_{\leq S}$. Hence the together$\rightarrow$not-together constraint is solvable.

The following algorithm can be used for together$\rightarrow$not-together constraints: First, check
whether the problem is solvable, using the result established above. Then, take the largest block with size \( m \) from \( P_{\bowtie_S} \) and distribute its elements over \( m \) different groups. Distribute the elements of the other blocks over these groups in such a way that that no two elements of a block end up in the same group. Obviously there are several ways of doing this, and any of them will do. Next, refine the obtained partition using the greedy algorithm for the \( \text{together} \rightarrow \text{together} \) constraints, until the wanted form is established.

**Example 8.4** : We consider again the situation described in the example for the previous constraint: person \( a \) works on project \( p_1 \), \( b \) on \( p_2 \) and \( p_3 \), \( c \) on \( p_2 \), \( d \) on \( p_1 \), \( e \) and \( \text{f} \) on \( p_3 \). This project partition of subjects (persons) is shown on the left of figure 8.6. The resource partition is again formed by rooms: there are 4 rooms with 1 seat and 1 room with 2. This time we demand that all people working on the same projects must not share rooms. Hence, the form of the grouping we look for is \( X, X, X, X \), shown in the middle of figure 8.6.

![Figure 8.6](image)

The goal is to find a partition \( P_{\bowtie_G} \) (of which only the form is known) such that the intersection with \( P_{\bowtie_S} \) results in a partition \( O \) where each block contains a single component.

Because we are dealing with a together—\( \text{not-together} \) the intersection of the partition \( \text{ad, b, c, ef} \) and the wanted grouping \( P_{\bowtie_G} \) must be equal to the partition \( O \) where each block has a single element: \( a, b, c, d, e, f \) (on the right of figure 8.6). In order to find such a grouping we will first produce a distribution (complement) of the partition \( \text{ad, b, c, ef} \) and, when necessary, refine it till it has the proper form \( (X, X, X, X, X, X) \).

In order to distribute the partition \( \text{ad, b, c, ef} \), we note that the largest block size of \( \text{ad, b, c, ef} \) is two. Hence, the distribution has two blocks and we can distribute the elements over them as follows: \( \text{abcdef} \) (other distributions are possible). \( P_{\bowtie_G} \) must be of the form \( X, X, X, X, X, X \), and hence the obtained two-blocked partition needs refinement. A solution can now be obtained by splitting blocks of this distribution repeatedly: \( a, d, e, f, bc \) is one of the results which can be obtained in three splits.

An alternative algorithm is to directly "distribute" the elements of each block of \( P_{\bowtie_S} \) into the form of the grouping, when possible. By "distributing" here we mean that all elements of the same block are distributed over different blocks in the target partition. In the above example this means that the elements of the blocks of \( \text{ad, b, c, ef} \) can directly be distributed over the blocks of \( X, X, X, X, XX \) to get \( a, d, e, f, bc \).
8.9.4 Not-together—together

We have already remarked that constraints of this type are not meaningful. Either the partition $P_{=s}$ or the wanted grouping is equal (or both) to the top partition $I$ which has one block. We will prove this by means of the following theorem:

**Theorem 8.4:** Suppose $P_{=s}$ is a partition of a set $C$ with more than one block. $I$ is the partition with $C$ as its only block. Let $[x]_{=s}$ and $[y]_{=s}$ be blocks of $P_{=s}$ and $x, y$ elements of $C$.

\[ \forall x, y \in C: [x]_{=s} \neq [y]_{=s} \Rightarrow [x]_{=G} = [y]_{=G} \Leftrightarrow P_{=G} = I. \]

**Proof:** $(\Rightarrow)$ We assume $P_{=s}$ has more than one block and $[x]_{=s} \neq [y]_{=s} \Rightarrow [x]_{=G} = [y]_{=G}$. By contraposition suppose that $P_{=G}$ has more than one block. Then there are two elements $a, b$ such that $[a]_{=G} \neq [b]_{=G}$. Then by assumption $[a]_{=s} = [b]_{=s}$. $P_{=s}$ has more than one block, hence there is an element $c$ in a different block from $a$ and $b$ in $P_{=s}$. But then $c$ must be in the same block as $a$ and $b$ in $P_{=G}$, and hence $[a]_{=G} = [b]_{=G} = P_{=G}$.

$(\Leftarrow)$ Trivial. If $P_{=G} = I$ then $[x]_{=G} = [y]_{=G}$ for all $x, y \in C$.

Clearly, putting all subjects into the same group is often not a very useful constraint. Subjects can only be put into different groups if the partition of subjects is equal to the unpartitioned set of subjects. An example may clarify this. Suppose we demand that for all (different) people $x$ and $y$: if $x$ and $y$ work on different projects then they must be placed in the same room. We also assume that there is more than one project. If there are people working on different projects then everybody ends up in the same room. To see this, suppose you work in a room together with another person. Suppose there is another room next door, then the persons occupying this room should work on the same projects as you and your roommate. Otherwise they should be in your room. However, we have assumed more than one project so somewhere there must be sitting someone who works on a different project than you. He or she must be in your room! But then this roommate of yours will also work on different projects from the people next door. Hence he/she must also be in their room and everybody ends up in a single room. Only if all people work on the same projects can they be assigned to different rooms.

The form of this constraint can be quite confusing. One can argue that the natural language statement: "If people work on different projects they must share a room" can mean something like "rooms should not contain people working on the same projects". If one agrees with this, then the statement "If people work on different projects they must share a room" cannot be translated to the universally quantified statement "$x$ and $y$ work on different projects $\Rightarrow x$ and $y$ share a room", but rather to "$x$ and $y$ share a room $\Rightarrow x$ and $y$ work on different projects". This last formulation is a together—not-together constraint.

8.10 Combining constraints

A grouping problem often consists of more than one of the constraints discussed above. If these are of the same type they can be rewritten to one constraint of that type. For example two constraints of the same type can be rewritten to one constraint of that type by taking the intersection of the two partitions of subjects, and the intersection of the two partition of resources. The resulting constraint then defines a relation between these intersections.
Note that the intersection of partitions is equivalent to the conjunction of abstract properties. A small example might explain the idea intuitively: Suppose a number people, each working on certain projects are assigned to offices. Some persons are married to each other. Here, we have two partitions on persons: a projects partition and a marriage partition. Two constraints of the same type like, for example: "people working on the same projects must share rooms", and "people married to each other should share a room", can be rewritten into one constraint: "People working on the same projects and who are married to each other, must share a room". Clearly the conjunction corresponds to the intersection of the projects and marriage partitions.

8.10.1 Combining constraints of different types

When multiple constraints of different types have to be satisfied, finding groupings for each constraint will not suffice. Satisfying a single constraint can be done, as we have seen, by finding a grouping which is isomorphic to the partition of resources, i.e. has the same form. To satisfy more than one constraint requires the same isomorphism for all the groupings which satisfy the individual constraints.

Problems with more than one constraint of the types described above, can mention different partitions on subjects and resources. For example, consider a constraint stating that all smokers and non-smokers are not allowed to share a room, and another constraint stating that all people working on project X should be given a desk close to a window. The first constraint mentions "smoker/non-smoker" as the subjects partition and "rooms" as resource partition. The second distinguishes "workers on project X" (and those who do not work on project X) as a bipartite partition of subjects. The partition of resources is here given by "window seats/non-window seats". In such cases one must find a grouping which satisfies both constraints. It is clear that multiple constraints constrain each others solutions further.

We consider the problem of finding a grouping which satisfies two constraints, $C_1$ and $C_2$. $C_1$ mentions a partition of subjects $P_{\approx 1}$ and its solution is a partition (a grouping) $P_{\approx 1;1}$ with the same form as a partition on resources $P_{\approx 1;2}$. (The same holds for $C_2$ using similar partitions, suffixed with 2 instead of 1.)

Suppose we find two groupings $P_{\approx 1;1}$ and $P_{\approx 1;2}$ which each satisfy the two constraints $C_1$ and $C_2$ respectively. In order to find a single grouping which satisfies both constraints at the same time we must make sure that the intersection of these two groupings is of the same form as the intersection of the resource partitions of the two constraints.

We make use of the following theorem.

**Theorem 8.5** : Let $P_{\approx 1;1}$ be a grouping with the same form as $P_{\approx 1;2}$, satisfying a constraint $C_1$. Let $P_{\approx 2;2}$ be a grouping with the same form as $P_{\approx 2;2}$, satisfying a constraint $C_2$.

The grouping $P_{\approx 1;1} \cap P_{\approx 1;2}$ is a solution to both $C_1$ and $C_2$ iff it has the same form as $P_{\approx 1;1} \cap P_{\approx 1;2}$.

We prove this as follows:

**Proof**: ($\Rightarrow$) Suppose $P_{\approx 1;1} \cap P_{\approx 1;2}$ is a solution to both $C_1$ and $C_2$ then we can map each block of it to a similar block of $P_{\approx 1;1}$ and $P_{\approx 1;2}$, because $P_{\approx 1;1}$ has the form of $P_{\approx 1;1}$ and $P_{\approx 1;2}$ of $P_{\approx 1;2}$, and hence to their intersection.

($\Leftarrow$) Suppose $P_{\approx 1;1} \cap P_{\approx 1;2}$ has the form of $P_{\approx 1;1} \cap P_{\approx 1;2}$ then it satisfies $C_1$ and $C_2$. This is trivial since $P_{\approx 1;1}$ and $P_{\approx 1;2}$ both have the same form as $R_1$ and $R_2$, respectively, and hence their
intersection is a solution to both $C_1$ and $C_2$.

Intuitively, the conjunction of two abstract properties stands for the intersection of the associated partitions. Formally, if the intersection of the two groupings is isomorphic to the intersection of the resource partitions then the two groupings can be mapped to their respective resource partitions under the same isomorphism.

### 8.10.2 Using the partition lattice for problem solving

An initial method for solving multiple grouping constraints $C_1 \ldots C_n$ can now be formulated as follows: Solve $C_1$ and $C_2$ and check whether the form of their intersection matches that of the intersections of the respective forms of the resource partitions. If not, start backtracking, else check the given grouping with the following constraint $C_3$ etc.

However, this is a very naive approach. In some cases it is not difficult to come up with better methods. The basic idea that we would like to emphasize is that one can use the partition lattice to conduct the search. Constraints provide information about where the solution should be located. One can make use of this when developing methods. We will give an example of how multiple constraints can make use of the partition lattice.

**Example 8.5:** The set \{a, b, c, d, e, f\} of persons is to be assigned to three rooms, each of two places each. Only b and c smoke. Person a works on project p_1, b on p_2 and p_3, c on p_4, d on p_1, e and f on p_3. There are two constraints, on the assignment of persons to places, $C_1$ and $C_2$.

- $C_1$: All people working on the same projects must share rooms.
- $C_2$: Smokers are not allowed to share a room with non-smokers.

*Constraint* $C_1$ is a together—together constraint, $C_2$ a not-together—not-together constraint. *Both constraints mention the same partition of resources, namely rooms.* Hence, $P_{\equiv R_1} = P_{\equiv R_2}$. Let $P_{s_1}$ be the projects partition of subjects, $P_{s_2}$ the smoke partition. Let $P_{G_1}$ be the wanted grouping for $C_1$, and $P_{G_2}$ for $C_2$. Obviously $P_{G_1}$ has the form of $P_{\equiv R_1}$, $P_{G_2}$ of $P_{\equiv R_2}$. Hence the constraints $C_1$ and $C_2$ can be written as:

\[
\begin{align*}
C_1: & \quad P_{s_1} \subseteq P_{G_1} \\
C_2: & \quad P_{G_2} \subseteq P_{s_2}
\end{align*}
\]

Since both constraints mention the same resource partition (rooms), $P_{\equiv R_1} = P_{\equiv R_2}$ the groupings must have the same form. Since we look for a grouping satisfying both constraints we have $P_{G_1} = P_{G_2}$ and hence the two constraints are equivalent to: $P_{s_1} \subseteq P_{G_1} \subseteq P_{s_2}$.

Notice that this implies that $P_{s_1} \subseteq P_{s_2}$. If we check this in our example, we find that this is indeed the case: the projects partition is a refinement of the smokers partition. If this were not so, (for example 'a' and 'c' would be the smokers) then the problem would have no solution.

Since $G_1 = G_2$, they occupy the same place on the lattice which is on a downward path from $P_2$ to $P_1$, see figure 8.7.

The wanted grouping can be found by splitting blocks from $P_2$ in such a way that elements together in $P_1$ are not separated by the splits. This will result in a grouping which is a refinement
of $P_2$, and which can be refined towards $P_1$. Actually, one split is all that is needed, since $G_1$ has to have 3, and $P_2$ has 2 blocks. $P_2 = bc adef$, splitting $bc$ makes no sense because it will not lead to the preferred form. $a$ and $d$ can’t be split because they are together in a block of $P_1$. This is also true for $e$ and $f$, hence the result is $bc adef$.

Notice that we can have a similar method by going upward in the lattice, starting from $P_3$ and merging blocks. In that case only merges are allowed which do not bring together elements which are separated in $P_2$.

This example shows how combining constraints of types $together-together$ and $not-together-together$ which share the same partition of resources can be solved. It serves to show how one can make use of formal properties in the formulation of methods to solve problems with multiple constraints.

Other combinations can be more difficult. From the properties of the individual types one can deduce that constraint sets which include $C_3$ types are harder to solve than others. The reason lies in the absence of a unique complement. Whereas in $C_1$ and $C_2$ the problem is given by refinement, $C_3$ involves the refinement of a complement. Since complements are not unique backtracking seems hard to circumvent.

We will not list all possible combinations of the three constraint types here. We will leave their analysis to some other time. Note however, that every constraint starts at a partition in the lattice and knows the form, and hence the level of the partition where the solution can be found. Combining constraints involves bringing two starting points together. Dependent on these positions one can exclude certain parts of the lattice, or come up with efficient methods for searching a solution.

### 8.11 An update system for grouping

We will show how grouping can be represented as an update system along the lines of chapter 6. In a propositional update logic the minimal state is formed by the powerset of the propositional atoms. When ordered by set inclusion the elements of this powerset form a lattice. Updates are operations on this lattice.

In a similar way the partition lattice (containing all partitions on the set of subjects) can be used as a minimal state of an update system for grouping. The objective is to find the right grouping within this lattice. Knowledge provided in the form of constraints is used to progress from the minimal state to a solution.
We briefly describe update operations for the three main constraint types: `together→together`, `not-together→not-together` and `together→not-together`.

Let \( S \) be the set of subjects, and \( \Pi(S) \) the set of all partitions of subjects. This last set, ordered by refinement gives us the partition lattice. The minimal knowledge state \( \emptyset \) is defined as \( \Pi(S) \). The absurd knowledge state \( 1 \) is defined as \( \emptyset \).

We introduce four basic update operations:

\( \sigma[generalisation(P)] \): This operation results in a knowledge state which only contains partitions which are in \( \sigma \) and of which \( P \) is a refinement. Formally:
\[
\sigma[generalisation(P)] = \sigma \cap \{ Q \mid P \subseteq Q, Q \in \Pi(S) \}.
\]

\( \sigma[refinement(P)] \): Here, \( P \) is a partition on subjects. This operation results in a knowledge state which contains only refinements of \( P \) which are also in \( \sigma \). Formally:
\[
\sigma[refinement(P)] = \sigma \cap \{ Q \mid Q \subseteq P, Q \in \Pi(S) \}.
\]

\( \sigma[refined-distribution(P)] \): This operation results in the knowledge state which contains all refinements of all complements of the partition \( P \), and are in \( \sigma \).
\[
\sigma[refined-distribution(P)] = \sigma \cap \{ Q \mid Q \cap P = I, Q \in \Pi(S) \}.
\]

\( \sigma[form(X,L)] \): Here \( X \) is an integer representing the number of blocks and \( L \) is a list of integers representing the size of each block. The update operation then results in a state which contains all partitions of the form specified by \( X \) and \( L \) provided they are in \( \sigma \).
\[
\sigma[form(X,L)] = \sigma \cap \{ Q \mid Q \text{ has the form given by } X \text{ and } L, Q \in \Pi(S) \}.
\]

With these basic update operations we can describe how the three types of constraints can be used. An agent knowledge state is formed by all those partitions which he regards as possible solutions. Initially these will include all partitions. Satisfying the available constraints will be the criterion of the problem.

Each constraint can be formulated as an epistemic test. The test is passed by a state which satisfies the constraint. In more detail we can specify the steps to arrive at such a state by making use of the formal properties of each constraint. This process is similar to the alternative formulations of the classification criteria in chapter 6. In order to come to a state which satisfies the constraint, a number of updates is needed. For example, an update of an instance of a `together→together` will proceed as follows. The constraint mentions a partition on subjects \( P \) and the form of the grouping. We know by the type of the constraint that \( P \) should be a refinement of the grouping. We therefore update the initial state \( \Pi(S) \) with \( generalisation(P) \). This results in the state \( \sigma \) containing all those partitions of which \( P \) is a refinement. Next, we update the form of the grouping \( form(X,L) \) to \( \sigma \). This leaves us with a new state containing all the partitions which satisfy the constraint.

For `not-together→not-together` constraints a similar procedure is followed, except that one uses \( refinement(P) \) instead of \( generalisation(P) \). `Together→not-together` constraints are dealt with using the \( refined-distribution \) update. This update operation together with an update of the proper form results in a state containing all partitions which satisfy the constraints.

These update operations are not intended as computationally viable methods. For example, an update with \( refinement(P) \) returns a state with all refinements of \( P \). This is often too much and one may prefer finding one refinement and backtrack to another one when necessary. These update operations should be seen as abstract steps in the description of the competence of the method.
8.12 Discussion

In the pages above, we have tried to show how a grouping problem can be formulated and solved in terms of partitions. Grouping can then be seen as the restricted search for a partition in a given partition lattice. This search is restricted by the number of steps (splits/merges) needed to arrive at a partition of suitable form, and criteria for the (un)solvability of the individual constraint types.

Given the proper partition algebra, and its lattice the solution space is given by those elements in the lattice which satisfy the constraints. Each constraint was defined as a condition in the partition lattice between the grouping and a known partition. Similar to the semantics of a sentence in logic, a problem is now translated to an algebraic representation.

There is a parallel between the structure of an update system, as presented in chapter 6 and the ordering of all lattices of a set by the refinement relation. The powerset of a propositional language $L$, as used in update semantics, also forms a lattice when ordered by set inclusion: $(\bigvee(L), \subseteq)$. In contrast to a partition lattice this one is distributive and the associated algebra is a boolean algebra.

Hence the basic structures of update semantics and grouping problems are similar: they are both lattices. The last section showed how this similarity can be exploited to define update actions for a partition lattice. Trying to satisfy a constraint of a certain type means to find a particular partition in the lattice. One starts with a known partition and the form of the grouping. This information alone leads to a substantial reduction of the number of solutions.

The update operations provided at the end of this chapter are too general to be computationally viable. However, when describing the types of grouping constraints we did provide algorithmic procedures for each of them. These procedures make use of the formal properties of each constraint formulation and consist of relatively simple combinatorial operations. In addition, we also provided effective procedures for recognizing unsolvable problems of each constraint type. Furthermore, we have shown how by combining constraints of different types one can use the structure of the lattice to produce an efficient search algorithm.

The 'semantics' of the problem is given here, not in terms of a boolean algebra but in terms of a partition algebra. Hence, by analyzing the mathematical structure of the problem first, we have shown that the problem can be described and solved within a language with limited expressive power.
This concluding chapter will provide answers to the research question formulated in chapter 1. The results of the previous chapters will be reviewed and put into perspective.

9.1 A review of the previous chapters

In the first chapter we have formulated the main research question as follows:

How can we give a knowledge-level characterization of the properties of knowledge intensive tasks?

This question was then refined into three research questions:

1. What does a conceptualization of a task look like?
2. How do we represent these knowledge-level conceptualizations in a formal way?
3. How can we use these representations to acquire a better understanding of the task?

In this section we will answer these questions on the basis of the results of the previous chapters. As a whole the chapters of the thesis can be seen as a series of explorations, aimed at answering the main research question, each of these can be matched to one of the three research questions. Three types of explorations can be distinguished:

1. Explorations regarding the nature of the conceptualizations of the task. (Chapters 2, 3 and 7.)
2. Explorations regarding the nature of the representation for the conceptualizations. (Chapters 6, 7 and 8.)
3. Explorations regarding the use of task ontologies and representations to specific problems and problem criteria. (Chapters 2, 4, 5 and 7.)
9.1.1 What does a conceptualization of a task look like?

A first analysis of a conceptualization of a task has been presented in chapter 2. The participation in the Sisyphus experiment can be seen as a 'pre-formal' study of the classification task. The approach taken was to focus on ontology construction rather than solving the problem instance itself.

Several domain ontologies were presented, as well as a task ontology for classification. The construction of such a task ontology proved to be an important step in the process of capturing the nature of the task. In addition a criterion and a task method (pruning) were identified. Hence, all elements of a knowledge-level conceptualization of a task were presented.

The task ontology for classification in chapter 2 differs from the one presented in chapter 3 (Adapting tableaux for classification). The first ontology gives a more detailed account of the conceptualizations of the classification task. The ontology in chapter 3 focuses on essential conceptualizations of the task and is therefore less detailed.

The main reason for this difference is that in chapter 3 the aim was to describe the problem solving process (or rather the development of knowledge in that process) of weak and strong classification. Hence, the task ontology here only plays the role of providing initial conceptualizations of the task.

A third set of conceptualizations for classification was presented in chapter 4, where a spectrum of classification criteria was identified. Again there are differences between the conceptualizations described here and the previous ontologies for classification. In this chapter the aim is not to describe a problem solving process but to identify the different solution criteria for the task. This change of viewpoint leads to a refinement of certain ontological elements, particularly those concerning the matching relation.

The ontology in chapter 2 is rich and detailed but informal and serves as input for the other chapters. The two following chapters focus in more detail on different aspects of the task and for that reason focus on essential elements in the ontology, which are then more accurately described.

A similar process takes place for the assignment ontologies in chapters 7 and 8. The ontology presented in the latter is very short, since it only focusses on grouping problems, which are identified as subproblems of assignment.

In chapter 7 the conceptualizations of the task are presented first in an informal way and are then refined into a formal representation of the task. There is much emphasis on the different problem variations within the task itself. Here we try to identify those ontological elements which are judged essential in describing the various problems which are covered by the task. The ontology presented in chapter 8 is very short, since it only focusses on grouping problems, which are identified as subproblems of assignment.

Hence, when describing task ontologies we have first selected those elements which are used extensively in the description of some aspect of the task. Then those elements were more refined and given a more detailed representation. As a consequence our ontologies for the same task differ from chapter to chapter as they are used to describe different aspects of the task.
9.2 How do we represent these knowledge-level conceptualizations in a formal way?

In this thesis we have used a variety of representations to formalize conceptualizations. The rationale behind this is that the choice of representation depends to a large extent on the intended use of the analysis of the task. For example, the choice of formal representation of classification in chapter 3 (Adapting tableaux for classification) was motivated by its intended use in the description of weak and strong classification problem solving. Propositional logic was chosen to show how analytic tableaux can be used to mimic problem solving behaviour of several task methods. The use of tableaux offered some insights into the nature of the problem solving for the task. One of the results was that the task method for strong classification can be described as an abductive method. In addition weak classification can be seen as a check for consistency which can be elegantly represented by analytical tableaux.

These results prompted some reflections about the adequacy of propositional representations in abductive background theories in chapter 5. The result was an investigation whether strict implication instead of material implication could be used. We presented a form of "strict abduction" as a possible alternative.

The representations in chapters 4 and 7 are also influenced by problem-solving aspects, namely the description of classification and assignment problems respectively. The aim of both these chapters was to give an overview of the different problem variations within the task. The choice of a representational formalism was here inspired by the analysis of those ontological elements which feature prominently in different problem descriptions. Hence the notions of "matching" (chapter 4) and "preferences" (chapter 7) are given formal interpretations in such a way that they can be used to describe a wide variety of classification problems and assignment problems respectively.

The choice of formalism in chapter 8 is an illustration of the same principle, as it was inspired by the use of abstractions in grouping, a subproblem in many knowledge-intensive assignment problems. Since abstractions can be associated with equivalence relations and partitions, one can represent the problem terms of a partition algebra. Hence, in this case the choice of the formal representation was motivated by a general characteristic of an ontological notion: i.e. abstraction.

At this point one may ask whether one can somehow generalize the task-specific representation developed in chapter 3, 4, 7 and 8. The answer given in 6 is affirmative. The key point underlying this chapter is that all problem/solution formalizations involve a notion of knowledge state. The use of this notion was inspired by the approach taken in chapter 3, where with the help of tableaux it is shown how knowledge about the solution changes during the execution of weak and strong classification. This insight prompted the development of a more general description of knowledge-level problem solving in terms of knowledge states and state transitions. As explained in chapter 6 the representational framework was motivated by ideas on information and knowledge as presented by Dretske [23], and the update semantics of Veltman [95]. This choice of formalism was meant to unify the several chapters into one general framework. Both the use of tableaux for classification and the use of partitions in grouping can be described in it. It should be noted that the framework does not provide a syntactic unification of the representations. Its purpose is solely to provide a semantic account of knowledge-level conceptualizations.
9.3 How can we use these representations to acquire a better understanding of the task?

Generally speaking the various formal representations were used in two ways: statically and dynamically. The static way of using the various representation involved a description of the problem variations within the task. This is most clearly illustrated in chapter 4 in which a spectrum of classification criteria is systematically generated on the basis of some elementary conceptualizations. The approach here is similar to spectra for diagnosis criteria in the literature [21, 89].

Similarly, chapter 7 offers an overview of different assignment problems. No such spectrum as in the case of classification criteria was presented due to the greater complexity of the task. Still, one gets a good idea of what different problems fall within the range of the task of assignment.

In summary, the static use of the ontological representation consists of the description of problem variations within a task. The dynamic use consists of the description of changing knowledge during problem solving.

The dynamic use of the representations was illustrated by various descriptions of problem solving. In chapter 3 propositional analytical tableaux (a decision method for propositional logic) were used for classification. The manipulation of these tableaux can be interpreted as modelling the dynamics of the problem solving process. The 'semantic' nature of tableaux illustrates that classification problem solving can be described as the search for a right model. This approach was made more explicit and taken a step further in chapter 6. The correspondence between tableaux and knowledge states was explained here in more detail.

In chapter 8 the partition lattice was used to formulate methods for grouping problems. It was shown that formal properties of this structure can be used in the description of methods to solve grouping problems. Here again the dynamics of problem solving can be seen as the manipulation of a formal structure, in this case the partition lattice.

As a unifying framework (not as a representational language) we presented a description of problem solving in terms of states and updates. This framework presented in chapter 6 is also aimed at a dynamic description of problem solving. In this case the underlying representational language (update semantics) is itself dynamically motivated. The framework is an attempt to capture the dynamics of knowledge-level reasoning and offers a particular view on knowledge. Problem solving for some tasks can be described as increasing knowledge about possible solutions.

9.4 Synthesis and discussion

The results of the thesis can be summed up in the form of an answer to the main research question: How can we give a knowledge-level characterization of the properties of knowledge intensive tasks?

1. By a good and extensive specification of the ontology of the task.
2. By a characterization of the type and criterion of a solution to problem instances of the task.
3. By the construction of a spectrum of solution criteria for the problem.
4. By a characterization of the solution space.
By identifying a solution with a model in some logical system and the search for a solution as the identification of that model.

6. By giving a knowledge level account of the reasoning of the task in terms of acquiring knowledge about possible solutions, and moving through the solution space.

Identification of the vocabulary of the task, some criteria for solutions and an initial high-level description of a task-template make up an initial description of a task. The vocabulary of the task is usually captured in a task ontology. Its construction can be seen as a prerequisite to task analysis.

The main contribution of the thesis in this respect is the fact that we presented a more rigorous specification of task ontologies than is usual in KE. Often ontologies give a specification on a higher level of abstraction and leave notions implicit which we have explicitly formulated. For example, comparing the classification ontology in chapter 4, with ones presented by Motta in [65] or Wielinga et al. [97], our spectrum of criteria describes much more criteria, which are all generated on the basis of a few ontological notions. The main difference in approach between those ontologies and the one compared here is that we focussed much more on the systematic generation of alternative classification criteria.

A point which features heavily in this thesis is the description of problem variation. Systematic descriptions of variations within a task have not been presented very often. As mentioned above, the construction of a spectrum of diagnostic criteria has been presented in the literature by Console et al. [21] and TenTeije et al. [89]. Since classification is similar to diagnosis, an interesting topic for further study would be to investigate in which respects these spectra can be compared to each other. One could also ask the question whether the systematic approach of chapter 4, when applied to diagnosis, would reveal more criteria than hitherto described.

The task ontology for assignment also provided a detailed account of the conceptualizations used in the description of assignment problems. In KE the task of assignment has been relatively ignored and no good task ontology seems to be available in the literature. The proposed ontology can be regarded as a first step to fill this gap. However, providing a spectrum of criteria for assignment is not that easy. Assignment problems have much more variations compared with diagnostic and classification problem types. Furthermore the preconditions and assumptions on the representation of domain knowledge can vary much more, and it is hard to generate all possibilities in a systematic way. However we have shown how one can compare and order different problems, for example by refining constraints and preferences. This goes well beyond the description of assignment problems as given by Puppe [71].

The principle goal of describing a spectrum of criteria is to be able to compare and structure different problems which fall within the range of the task. Another way of looking at a spectrum of criteria is to see it as the description of all post- and preconditions of the problems of the task. In that sense a spectrum can be seen as a functional characterization of the task. However, this only describes the static nature of the task. In addition one would also like a dynamic characterization of the individual problems and the associated problem solving processes.

Therefore a next step is to use the conceptualizations in such a way that the underlying formal structure of a typical problem becomes visible. We identified this structure every time with the solution space. As an example consider the use of partitions in chapter 8. Here the partition lattice consists of an ordering of possible solutions (partitions) in which problem solving can be described. Also when describing tableaux for classification (chapter 3) the solution space can be identified as the set of classes which are still considered viable.
Hence, where a spectrum of criteria offers a description of what a solution consists of, for different problem variations, the underlying structure of problems is given by the set of possible solutions - the solution space. It is this space which is reduced during problem solving as it becomes known which candidates do, or do not qualify any longer as possible solutions.

In chapters 3, 5 and 8 solutions were identified quite explicitly with logical models and the problem space with a formal structure. For grouping (chapter 8) we identified a partition lattice as the solution space, and a partition as a model of a candidate solution.

This adaptation of the use of tableaux in chapter 3 should not be seen as an attempt to describe new, better or faster methods for classification with analytical tableaux. In fact, our use of tableaux has been quite elementary. Analytic tableaux offer a nice example of how semantics and syntax can be combined. Seen as syntactic structures tableaux can be used to build effective theorem provers [30]. Inspecting the branches of tableaux one finds that they correspond to models of the represented theory. Hence, one can interpret our use of analytic tableaux as an attempt to bridge the gap between a symbol-level and knowledge-level description. The symbol-level being the syntactic operations on the tableaux, and the knowledge-level roughly corresponds to the semantic interpretation of the models depicted in the tableau.

In the chapters were we used tableaux, solutions were associated with formal models. This "model-based" approach was made explicit in chapter 6. This approach of starting the analysis of a problem, or task, with the identification of a semantic structure differs from more traditional approaches in KE. Usually one starts with the choice of some, often rich, representational formalism and then represents the domain theory in it. Then problem solving is performed with the help of some accompanying inference engine.

The approach taken here, differs from this tradition in that it is "model-based". By analysing the problem in terms of searching for a model, the semantic structure can be identified. When this has been done an appropriate language can be selected which is not overly expressive and in which the problem can be accurately and intuitively described. Hence, our approach can be summarized by the slogan "semantics first, syntax later".

As a consequence this thesis does not present "yet another representational language". No "universal problem solving language" or a "one-fits-all" representation was chosen or developed. Instead every task was given a formalization which seemed to match it on the basis of a formalization of the ontology and the typical reasoning of a task method.

A dynamic, model-based approach was explicitly formulated in chapter 6. Problem solving is described as acquiring knowledge about solutions, and knowledge in this respect is strongly linked to the reduction of possibilities. We think that a model-based representation of knowledge intensive problems and tasks is a good way of providing a knowledge-level understanding of them.

In summary, we have used the conceptualizations in two ways. First, to construct a static structure aimed at the description of problem variations and a spectrum of solution criteria. The systematic description of problem variations allows one to order and compare different problems. This can be very beneficial for the development of knowledge based systems. This approach is, at least in spirit, similar to the one advocated by ten Teije [88]. In this work parameters for diagnosis were identified which could be varied in the description of several problem variations. In that way various diagnostic problems could be systematically generated.

Second, the conceptualizations were used to construct a structure of possible solutions which was used in the dynamic description of problem solving on the knowledge-level. This has been described in chapter 6 where problem solving behaviour was described in terms of changing states
of knowledge about possible solutions.

This dynamic, semantic account of problem solving for knowledge intensive tasks has some important advantages. It allows for the easy extensions to other, non-classical forms of reasoning and it provides a good illustration of our view of knowledge.

Knowledge in our view is a semantic concept and therefore a semantic account of knowledge-level reasoning is called for. Hence, instead of focussing on the development of knowledge based systems, or problem solving methods, we have given an analysis of task-oriented problem solving and representation. We feel that an analysis of knowledge intensive tasks should start with an account of what knowledge is required and how it changes during the problem solving process. The explorations presented in this thesis are means to that end.
Appendix A

Problem statement for Sisyphus: models of problem solving

This appendix contains a substantial part of the original problem statement of the Sisyphus I project as published by Linster [57] in 1994. It describes the office assignment problem treated in chapter 7. The text starts with a description of the goal of the Sisyphus project.

We devised the Sisyphus problem to compare different approaches to the modelling of problem-solving processes in knowledge-based systems and the influence of the models on the knowledge acquisition activities. To this purpose we give a short description of a sample problem concerned with office assignment in a research environment.

After a brief description of the settings in which the problem occurs, we describe the organizational structure of the research group and that group's facilities. We will then render a sample annotated protocol of the local expert Siggi D. solving the office assignment problem.

(...) 

I Statement of the sample problem

The members of the research group YQT of the HNE laboratory are moved to a new floor of their chateau (...). Due to the severe cuts in funding they only get a very limited number of offices (...). It will be quite a problem to cram them all in. To complicate matters even further some will have to share an office. After several vain attempts - each ending as nightmares that would have impressed Freddy - the management of HNE is desperate. Sisyphus is their last hope. HNF implores the Sisyphus teams to provide knowledgeable systems that are up to the task.

It is important that the systems' way to solve the problem follow the shining example of the wizard Siggi D., the only one who ever managed to solve the problem. The system developers should be aware of the fact that YQT's members are used to being pampered. They all have their personal preferences and professional peculiarities that had better be observed, as the dungeons of the BABYLON tower are deep and lonely.

II Data on people and offices

Description of the members of YQT

Not all members of YQT can profit from this new office space in the chateau: about half of the group stay in their old offices. Those that are concerned by the new assignment are:
Figure 1
The floor-plan of the chateau.
Hierarchical structures within YQT

Within the subset of members of YQT we have the following organizational structure: Thomas D. is the head of the group YQT; Monika X. and Ulrike U. are the secretaries; Werener L. and Angi W. work together in the RESPECT project; Harry C. Jurgen L. and Thomas D. work in the EULISP project; Michael M. and Hans W. work in the Babylon project; Hans W. is the head of this large project; Marc m., Uwe T. and Andy L. pursue individual projects; Katharina N. and Joachim I. are the heads of larger projects that are not considered in this problem.

The offices on YQT's floor of the chateau

C5-123, C5-122, C5-121, C5-120, C5-119 and C5-117 are large rooms that can host two researchers. Large rooms can be assigned to heads of groups too. C5-113, C5-114, C5-115 and
C5-116 are single rooms. [The shaded rooms in figure 1 are not available as office space - MGJ.]

<table>
<thead>
<tr>
<th>Action by the expert</th>
<th>Self-report transcript (stylized)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Put Thomas D. into office C5-117</td>
<td>1a The head of group needs a central office so that s/he is as close as possible to all the members of the group. This should be a large office. 1b This assignment is defined first, as the location of the office of the head of group restricts the possibilities of the subsequent assignments.</td>
</tr>
<tr>
<td>2 Monika X. and Ulrike U. into office C5-119.</td>
<td>2a The secretaries’ office should be located close to the head of group. Both secretaries should work together in one large office. This assignment is executed as soon as possible, as its possible choices are extremely constrained.</td>
</tr>
<tr>
<td>3 Eva I. into C5-116</td>
<td>3a The manager must have maximum access to the head of group and to the secretariat. At the same time he/she should have a centrally located office. A small office will do. 3b This is the earliest point at which this decision can be taken.</td>
</tr>
<tr>
<td>4 Joachim I. into C5-115.</td>
<td>4a The heads of large projects should be close to the head of group and the secretariat. There really is no reason for the sequence of assignments of Joachim, Hans, and Katharina.</td>
</tr>
<tr>
<td>5 Hans W. into C5-114.</td>
<td>5a The heads of large projects should be close to the head of group and the secretariat.</td>
</tr>
<tr>
<td>6 Katharina N. into C5-113.</td>
<td>6a The heads of large projects should be close to the head of group and the secretariat.</td>
</tr>
<tr>
<td>7 Andy and Uwe T. into C5-120.</td>
<td>7a Both smoke. To avoid conflicts with nonsmokers they share an office. Neither of them is eligible for a single office. This is the first twin-room assignment as the smoker/nonsmoker conflict is a severe one.</td>
</tr>
<tr>
<td>Action by the expert</td>
<td>Self-report transcript (stylized)</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>8 Werner L. and Jürgen L. into office C5-123.</td>
<td>8a They are both implementing systems, both nonsmokers. They do not work on the same project, but they work on related subjects. Members of the same projects should not share offices. Sharing with members of other projects enhances synergy effects within the research group. There are really no criteria for the sequence of twin-room assignments.</td>
</tr>
<tr>
<td>9 Marc M. and Angi W. into office C5122.</td>
<td>9a Marc is implementing systems; Angi isn’t. This should not be a problem. Putting them together would ensure good cooperation between the RESPECT and the KRITON projects.</td>
</tr>
<tr>
<td>10 Harry C. and Michael T. into office C5-121.</td>
<td>10a They are both implementing systems. Harry develops object systems. Michael uses them. This should create synergy.</td>
</tr>
</tbody>
</table>

**Note 1**

Our wizard Siggi D. seems to pursue a general strategy of assigning the head of group and the staff personnel first, followed by the heads of large projects, who through their seniority are eligible to single offices (...). The offices of the head of group and the staff should be close to each other. Heads of projects should, if possible, be allocated offices close to the head of group’s office.

**Note 2**

Twin offices are assigned to the members of the research projects, under the consideration that synergy among projects is boosted. This means that researchers who work on the same project are, if possible not sharing an office. Co-workers who work on related subjects can share an office. It is important not to put smokers and non-smokers together into twin offices.

(...)


Bibliography


References


Samenvatting

Knowledge Engineering is het vakgebied dat zich bezig houdt met het ontwerpen, bouwen en onderhouden van systemen die op één of andere wijze gebruik maken van grote hoeveelheden kennis. Als voorbeeld kan men denken aan systemen die medische diagnoses kunnen uitvoeren of systemen die aangeven op welke plaatsen olie kan worden gevonden.

De verschillende kennis-intensieve problemen die in de loop van de jaren zijn bestudeerd kunnen worden ingedeeld in een aantal verschillende probleem-typen, ook wel taken genaamd. Zo is het stellen van een medische diagnose een voorbeeld van een probleem dat valt onder een диагностische taak. Het vinden van een mankement in een niet werkende PC is dat ook. Naast diagnose worden o.a. ook planning, classificatie, ontwerp en toewijzing (assignment) als verschillende probleemtypen gezien.

Taken zijn analytisch of synthetisch van aard. Onder analytische taken vallen problemen die betrekking hebben op een bestaand 'systeem', terwijl bij synthetische problemen een 'systeem' geconstrueerd dient te worden. Zo is bijvoorbeeld het classificeren van gesteenten een probleem dat valt onder de analytische taak classificatie. Het doel van het probleem is te bepalen tot welke klasse een bepaald object (een gegeven gesteente) behoort. Een toewijzingsprobleem daarentegen behoort tot een synthetische taak omdat de toewijzing (assignment) in zekere zin geconstrueerd dient te worden.

Het onderscheid van problemen in taken heeft grote voordelen voor het ontwerpen, bouwen en onderhouden van kennis-systemen. Allereerst brengt het onderscheid van taken met zich mee dat men de domein-kennis scheidt van de meer procedurele kant van het probleem-oplossingsproces. Zo heeft een arts een grote hoeveelheid medische domein-kennis, terwijl een auto-monteur kennis heeft van auto's. Beide personen kunnen problemen oplossen die tot een diagnostische taak behoren (een arts stelt een medische diagnose m.b.t. een patient, een monteur een mechanische diagnose m.b.t. een auto), en ondanks hun verschillende domeinen wel degelijk iets gemeenschappelijks hebben.

Een ander groot voordeel is dat met elke taak één of meer taak-methoden kunnen worden verbonden. Een taak-methode kan men opvatten als een prototypische methode die gebruikt kan worden als startpunt voor de ontwikkeling van een meer specifieke methode die het betreffende probleem kan oplossen.

Ook kan men met een taak een typische beschrijving van kennis-representatie verbinden. Bij de beschrijving van een taak horen zogenoemde representatie-schema's die op een prototypische manier aangeven hoe kennis kan worden gepercepteerd.

Bij het ontwerp van een kennisstelsel kan men gebruik maken van deze indeling in taken door de problemen die het systeem dient te kunnen oplossen, te classificeren en gebruik te maken van procedures en typische representatie schema's. Zo hoeft de ontwerper een groot aantal beslissingen zelf niet meer te nemen.

De begrippen die gebruikt worden in de beschrijving van een taak worden vastgelegd in een zogenoemde ontologie van de taak. Een taak-ontologie bevat een verzameling definities van be-
grippen die gebruikt kunnen worden in beschrijving van problemen die onder de taak vallen. Zo bevat een taak-ontologie voor classificatie, definities voor begrippen als "object", "observatie", "attribuut" etc. Dergelijke definities zijn vaak informeel.

Dit proefschrift beschrijft hoe bestaande formalismen kunnen worden gebruikt in de beschrijving en karakterisering van individuele taken. Aan de hand van twee taken, een analytische (classificatie) en een synthetische (assignment), illustreren we hoe de conceptualisatie van een taak er uitziet. Vervolgens laten we zien hoe men dergelijke conceptualisaties formeel kan presenteren. De formalisatie wordt hierbij steeds zo gekozen dat de formele eigenschappen van de representatie ons iets leren over de taak. Een derde aspect dat aan de orde komt, is hoe deze representaties te gebruiken om inzicht te krijgen in eigenschappen van de taak in kwestie.

Het proefschrift bevat een aantal exploraties die elk één van bovenstaande punten behandelen. Hoofdstuk 2 beschrijft een verslag van deelname aan het Sisyphus III experiment. Onderzoekers binnen de Knowledge Engineering werden uitgenodigd een systeem te bouwen dat diverse stollingsgesteenteen zou moeten kunnen classifieren. Hiervoor was kennis-acquisitie materiaal, (o.a. in de vorm van interviews met experts) beschikbaar gesteld. Deelnemers werden geacht hun activiteiten bij te houden in een logboek, zodat na afloop de aanpak van verschillende deelnemers systematisch met elkaar konden worden vergeleken. Onze bijdrage aan dit experiment beslaat hoofdstuk 2. In dit proefschrift vervult het de rol van een exploratie die laat zien hoe het modelleren van kennis doorgaans plaats vindt. De aanpak is nog informeel en gericht op het construeren van diverse ontologien. Dit hoofdstuk vormt input voor volgende hoofdstukken waar verschillende aspecten van de classificatie-taak formeel nader worden uitgewerkt. Hoofdstuk 4 bevat een uitgebreide ontologie voor classificatie problemen.

Ook hoofdstuk 7 gaat in op de conceptualisatie van een taak. Ditmaal staat de taak assignment (toewijzing) centraal. Een voorbeeld van een toewijzingsprobleem is het toewijzen van personeel aan werkruimte (kamers). Hierbij dient men rekening te houden met een aantal gegeven eisen (constraints) en wensen (preferences). De meest ideale toewijzing is de oplossing van een dergelijk probleem. In hoofdstuk 7 wordt uitgebreid beschreven welke begrippen van belang zijn bij het beschrijven van toewijzingsproblemen.

De keuze voor een bepaalde formele representatie van een taak is in dit proefschrift steeds afhankelijk van het toekomstige gebruik van de taak-analyse. Zo is in hoofdstuk 3 gekozen voor een propositioneel logische benadering om het gebruik van tableaux voor probleemoplossen te illustreren. In hoofdstuk 4 is gekozen voor een andere representatie van classificatie omdat hier de variëteit van problemen binnen de classificatie-taak beschreven wordt.

In hoofdstuk 5 wordt nader ingegaan op de keuze van een propositioneel logische representatie voor abductieve theorieën. Zoals in hoofdstuk 2 wordt aangetoond kan sterke classificatie worden gezien als een vorm van abductie. Men kan echter vragen stellen bij de juiste interpretatie van de formele representatie van de domein-kennis. Hoofdstuk 5 beschrijft de problemen van de materiële implicatie binnen een abductieve context. Als een mogelijk alternatief wordt abductie met strikte implicatie beschreven.

De verschillende formele representaties van taken kunnen worden gebruikt om een beter inzicht te krijgen in wat een taak nu precies is. We kunnen een onderscheid maken tussen een statisch en dynamisch gebruik van deze representaties.

Het statisch gebruik is het meest duidelijk in hoofdstuk 4. Hier wordt een spectrum beschreven van allerlei mogelijke classificatie-criteria. Een criterium is een bewering die beschrijft wanneer een bepaalde oplossing is bereikt. Het is dus niet moeilijk criteria te verbinden met problemen.
Het beschreven spectrum beschrijft dan ook problemen die allemaal behoren tot de taak classificatie. We laten zien hoe door een aantal basisbegrippen te combineren steeds gecompliceerdere classificatie-criteria kunnen worden beschreven.

In hoofdstuk 7 wordt een overzicht gegeven van verschillende toewijzingsproblemen. Hier wordt dieper ingegaan op de analyse van *constraints* en preferenties. De analyse wordt vervolgens toegepast in de beschrijving van een probleem waarbij werknemers moeten worden toegewezen aan een aantal werkplaatsen op een afdeling.

Het *dynamisch* gebruik heeft betrekking op het probleem-oplossen binnen de taak. Hoofdstuk 3 gaat nader in op een aantal probleem oplosmethoden voor classificatie. Met behulp van analytische (of semantische) tableau laten we zien hoe de methoden van 'sterke' en 'zwakke' classificatie kunnen worden gerepresenteerd. Semantische Tableaux vormen een beslissingsmethode binnen de klassieke logica. Met behulp van tableaux kan men o.a. de geldigheid van bepaalde beweringen nagaan. In dit hoofdstuk worden tableaux gemanipuleerd, waarbij afgeweken wordt van de standaard regels voor het gebruik. Op deze manier kan worden aangetoond dat sterke classificatie overeenkomt met abductie, een manier van redeneren waarbij men verklaringen zoekt voor waargenomen fenomenen.

Hoofdstuk 8 gaat in op een deelprobleem van veel *assignment*-problemen, namelijk gegroepeerd toewijzen (ook bekend als *grouping*). Bijvoorbeeld, het toewijzen van 8 mensen aan 4 kamers met elk 2 plaatsen kan men uitvoeren door elke persoon individueel aan een plaats toe te wijzen. Men kan echter eerst ook groepjes van twee vormen en deze groepjes aan een kamer toewijzen. Deze strategie wordt uitgebreid en formeel geanalyseerd in hoofdstuk 8.

In hoofdstuk 6 wordt een meer algemene aanpak voor een dynamische beschrijving van kennis-intensief probleem-oplossen voorgesteld. Deze aanpak, of raamwerk, bestaat uit een beschrijving van probleem-oplossen in termen van welke kennis beschikbaar is voor, tijdens en na het oplossen van het probleem. De formele beschrijving hiervan maakt gebruik van begrippen als 'kennis-toestanden' en 'toestandsovergangen' die zijn ontleend aan *update* semantiek. Het oplossen van een probleem wordt beschreven als een reeks toestandsovergangen. Men begint met een toestand met weinig of geen kennis m.b.t. de mogelijke oplossing. Gaandeweg neemt kennis over de oplossing toe totdat een toestand wordt bereikt waarin een oplossing gevonden is. Deze toestand wordt gekenmerkt door de acceptatie van het criterium van het probleem.

Dit proefschrift levert door middel van deze exploraties een bijdrage aan een beter begrip van wat een taak nu eigenlijk inhoudt. De formele aanpak resulteert in een precise beschrijving van de begrippen die gebruikt worden in beschrijvingen van de taak. De gebruikte taak-ontologieën zijn dan ook formeler en meer gedetailleerd dan doorgaans het geval is. De uitgebreide beschrijvingen en karakterisering van probleem-variaties stelt men in staat om verschillende problemen met elkaar te vergelijken en te classificeren. Tenslotte, stelt het raamwerk van kennis-toestanden en -overgangen men in staat een beschrijving te geven van de dynamische aspecten van kennis-intensieve taken.
SIKS Dissertatieruks

1998-1  Johan van den Akker (CWI)
DEGAS - An Active, Temporal Database of Autonomous Objects

1998-2  Floris Wiesman (UM)
Information Retrieval by Graphically Browsing Meta-Information

1998-3  Ans Steuten (TUD)
A Contribution to the Linguistic Analysis of Business Conversations within
the Language/Action Perspective

1998-4  Dennis Breuker (UM)
Memory versus Search in Games

1998-5  E.W.Oskamp (RUL)
Computerondersteuning bij Straftoeming

1999-1  Mark Sloof (VU)
Physiology of Quality Change Modelling; Automated modelling of Quality
Change of Agricultural Products

1999-2  Rob Potharst (EUR)
Classification using decision trees and neural nets

1999-3  Don Beal (UM)
The Nature of Minimax Search

1999-4  Jacques Penders (UM)
The practical Art of Moving Physical Objects

1999-5  Aldo de Moor (KUB)
Empowering Communities: A Method for the Legitimate User-Driven Speci-
fication of Network Information Systems

1999-6  Niek J.E. Wijngaards (VU)
Re-design of compositional systems

1999-7  David Spelt (UT)
Verification support for object database design
1999-8  Jacques H.J. Lenting (UM)
Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.

2000-1  Frank Niessink (VU)
Perspectives on Improving Software Maintenance

2000-2  Koen Holtman (TUE)
Prototyping of CMS Storage Management

2000-3  Carolien M.T. Metselaar (UVA)
Sozial-organisatorische gevolgen van kennis technologie; een procesbenader ing en act operspectief.

2000-4  Geert de Haan (VU)
ETAG, A Formal Model of Competence Knowledge for User Interface Design

2000-5  Ruud van der Pol (UM)
Knowledge-based Query Formulation in Information Retrieval.

2000-6  Rogier van Eijk (UU)
Programming Languages for Agent Communication

2000-7  Niels Peek (UU)
Decision-theoretic Planning of Clinical Patient Management

2000-8  Veerle Coupé (EUR)
Sensitivity Analysis of Decision-Theoretic Networks

2000-9  Florian Waas (CWI)
Principles of Probabilistic Query Optimization

2000-10  Niels Nes (CWI)
Image Database Management System Design Considerations, Algorithms and Architecture

2000-11  Jonas Karlsson (CWI)
Scalable Distributed Data Structures for Database Management

2001-1  Silja Renooij (UU)
Qualitative Approaches to Quantifying Probabilistic Networks
2001-2  Koen Hindriks (UU)
Agent Programming Languages: Programming with Mental Models

2001-3  Maarten van Someren (UvA)
Learning as problem solving

2001-4  Evgueni Smirnov (UM)
Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets

2001-5  Jacco van Ossenbruggen (VU)
Processing Structured Hypermedia: A Matter of Style

2001-6  Martijn van Welie (VU)
Task-based User Interface Design

2001-7  Bastiaan Schonhage (VU)
Diva: Architectural Perspectives on Information Visualization

2001-8  Pascal van Eck (VU)
A Compositional Semantic Structure for Multi-Agent Systems Dynamics.

2001-9  Pieter Jan 't Hoen (RUL)
Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes

2001-10 Maarten Sierhuis (UvA)
Modeling and Simulating Work Practice
BRAHMS: a multiagent modeling and simulation language for work practice analysis and design

2001-11 Tom M. van Engers (VUA)
Knowledge Management: The Role of Mental Models in Business Systems Design

2002-01 Nico Lassing (VU)
Architecture-Level Modifiability Analysis

2002-02 Roelof van Zwol (UT)
Modelling and searching web-based document collections
2002-03  Henk Ernst Blok (UT)
Database Optimization Aspects for Information Retrieval

2002-04  Juan Roberto Castelo Valdueza (UU)
The Discrete Acyclic Digraph Markov Model in Data Mining

2002-05  Radu Serban (VU)
The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents

2002-06  Laurens Mommers (UL)
Applied legal epistemology; Building a knowledge-based ontology of the legal domain

2002-07  Peter Boncz (CWI)
Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications

2002-08  Jaap Gordijn (VU)
Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas

2002-09  Willem-Jan van den Heuvel(KUB)
Integrating Modern Business Applications with Objectified Legacy Systems

2002-10  Brian Sheppard (UM)
Towards Perfect Play of Scrabble

2002-11  Wouter C.A. Wijngaards (VU)
Agent Based Modelling of Dynamics: Biological and Organisational Applications

2002-12  Albrecht Schmidt (Uva)
Processing XML in Database Systems

2002-13  Hongjing Wu (TUE)
A Reference Architecture for Adaptive Hypermedia Applications

2002-14  Wieke de Vries (UU)
Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems

2002-15  Rik Eshuis (UT)
Semantics and Verification of UML Activity Diagrams for Workflow Modelling

2002-16 Pieter van Langen (VU)
The Anatomy of Design: Foundations, Models and Applications

2002-17 Stefan Manegold (UVA)
Understanding, Modeling, and Improving Main-Memory Database Performance

2003-01 Heiner Stuckenschmidt (VU)
Ontology-Based Information Sharing in Weakly Structured Environments

2003-02 Jan Broersen (VU)
Modal Action Logics for Reasoning About Reactive Systems

2003-03 Martijn Schuemie (TUD)
Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy

2003-04 Milan Petkovic (UT)
Content-Based Video Retrieval Supported by Database Technology

2003-05 Jos Lehmann (UVA)
Causation in Artificial Intelligence and Law - A modelling approach

2003-06 Boris van Schooten (UT)
Development and specification of virtual environments