Using Expressive and Flexible Action Representations to Reason about Capabilities for Intelligent Agent Cooperation

Gerhard Jürgen Wickler

Ph.D.
University of Edinburgh
1999

Abstract

The aim of this thesis is to address the problem of capability brokering. A capability-brokering agent receives capability advertisements from problem-solving agents and problem descriptions from problem-holding agents. The main task for the broker is to find problem-solving agents that have the capabilities to address problems described to the broker by a problem-holding agent. Capability brokering poses two problems: representing capabilities, for advertisements, and matching problems and capabilities, to find capable problem-solvers.

For the representation part of the problem, there have been a number of representations in AI that address similar issues. We review various logical representations, action representations, and representations for models of problem solving and conclude that, while all of these areas have some positive features for the representation of capabilities, they also all have serious drawbacks. We describe a new capability description language, CDL, which shares the positive features of previous languages while avoiding their drawbacks. CDL is a decoupled action representation into which arbitrary state representations can be plugged, resulting in the expressiveness and flexibility needed for capability brokering.

Reasoning over capability descriptions takes place on two levels. The outer level deals with agent communication and we have adopted the Knowledge Query and Manipulation Language (KQML) here. At the inner level the main task is to decide whether a capability description subsumes a problem description. In CDL the subsumption relation for achievable objectives is defined in terms of the logical entailment relation between sentences in the state language used within CDL. The definition of subsumption for performable tasks in turn is based on this definition for achievable objectives. We describe algorithms in this thesis which have all been implemented and incorporated into the Java Agent Template where they proved sufficient to operationalise a number of example scenarios.

The two most important features of CDL are its expressiveness and its flexibility. By expressiveness we mean the ability to express more than is possible in other representations. By flexibility we mean the possibility to delay decisions regarding the compromises that have to be made to knowledge representation time. The scenarios we have implemented illustrate the importance of these features and we have shown in this thesis that CDL indeed possesses these features.

Thus, CDL is an expressive and flexible capability description language that can be used to address the problem of capability brokering.

Acknowledgements

This work has been performed under a studentship funded as part of the O-Plan project. The O-Plan project is sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory (Rome), under grant number F30602-95-1-0022. The O-Plan project is monitored by Dr. Northrup Fowler III and Mr. Wayne Bosco. The U.S. Government and the University of Edinburgh are authorised to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing official policies or endorsements, either express or implied, of DARPA, Air Force Research Laboratory, the U.S. Government, or the University of Edinburgh.

I am also very grateful for the supervision I have received during my work on this thesis. My principal supervisor was Austin Tate, who has overseen my work for the whole period. Other supervisors included Alan Bundy, Louise Pryor, Julian Richardson, and Brian Drabble, who were involved at various stages of this thesis. Apart from my supervisors I have also received support from various other people at the Department of Artificial Intelligence and the Artificial Intelligence Applications Institute at the University of Edinburgh. Amongst these, I am particularly grateful to Steve Polyak who, amongst other things, was the first to read and comment on this thesis. Needless to say, any remaining mistakes are my own.

Declaration

I hereby declare that I composed this thesis entirely myself and that it describes my own research.

Gerhard Wickler
Edinburgh
April 28, 1999

Contents

Abstract iii

Acknowledgements v

Declaration vii

1 Introduction: Capability Brokering 1
- 1.1 The Problem of Capability Brokering . . . 1
- 1.2 Capability Brokering in Context . . . 7
- 1.3 Criteria for Success . . . 12

2 Capability Brokering: A Literature Survey 15
- 2.1 Software Agents . . . 15
- 2.2 Modelling Capabilities with Logics . . . 28
- 2.3 Actions in AI Planning . . . 39
- 2.4 Models of Problem Solving . . . 52

3 Scenarios, Agents, and Messages 65
- 3.1 The Initial Scenario . . . 65
- 3.2 Inter-Agent Messages . . . 72
- 3.3 More Complex Scenarios . . . 79

4 A Capability Description Language: CDL 85
- 4.1 Problems for Capability Representations . . . 85
- 4.2 Achievable Objectives . . . 91
- 4.3 Performable Actions . . . 106
- 4.4 Other Properties . . . 114
- 4.5 Examples . . . 117

5 Algorithms and Implementation of CDL 129
- 5.1 Basic Capability Evaluation . . . 129
- 5.2 Extended Capability Evaluation . . . 140
- 5.3 Capability Retrieval in JAT . . . 162

6 Further Experiments and Results **179**
6.1 Variations on the Expressiveness Scenario 180
6.2 Variations on the Flexibility Scenario 189
6.3 Performance Issues 196

7 Expressiveness of CDL **203**
7.1 Why more Expressiveness? 203
7.2 Expressiveness of AR Languages 209
7.3 CDL: An AR1 Language 220

8 Flexibility of CDL **227**
8.1 Why Flexible Action Representations? 227
8.2 Defining and Implementing Flexibility 234

9 Related Work and Evaluation **245**
9.1 Comparison with other Brokers 245
9.2 CDL: Expressiveness and Flexibility 261
9.3 Other Domains 273

10 Conclusions **279**
10.1 Possible Extensions 279
10.2 Summary 284

Chapter 1

Introduction: Capability Brokering

The aim of this thesis is to address the problem of capability brokering. For this purpose we will define a new capability description language and show that it has two desirable properties: it is expressive and highly flexible. The first step towards this goal must be a definition of the problem addressed in this thesis. The contribution of this chapter will be a characterisation of the problem of capability brokering and its context, as well as criteria for a successful solution.

1.1 The Problem of Capability Brokering

In this section we will define the problem of capability brokering for intelligent software agents, especially those that may be based on AI planning technology. This is the main problem addressed in this thesis.

1.1.1 Rational Agents and Communication

One approach to achieving artificial intelligence is the *rational agent approach* [Russell and Norvig, 1995, page 7]. In this approach, the field of AI is viewed as the study and construction of rational agents. But what is a rational agent? Unfortunately there is no agreed answer to this question as yet. For example, Russell

and Norvig describe an agent as an entity that perceives and acts. Rationality means that it acts to achieve its goals, given its beliefs.

A more precise characterisation of what an agent is can be found in [Wooldridge and Jennings, 1995, page 116]. They identify four *necessary properties* of an agent which most definitions of agency seem to agree on:

- autonomy
- social ability
- reactivity
- pro-activeness

Social ability, the property we will be most concerned with, means that an agent interacts with other agents (possibly humans) via some kind of agent communication language. Pro-activeness means that an agent should be able to exhibit goal-directed behaviour by taking the initiative. Pro-activeness corresponds to rationality in Russell and Norvig's characterisation above. Taken together, pro-activeness and social ability imply that an agent should communicate not with just any other agent, but specifically with those agents that can help it achieve its goals. For an agent to achieve this behaviour, it will be necessary to first *find* these other agents that can help it achieve its goals. Finding such agents is part of the problem we are addressing in this thesis.

This problem is very similar to what [Davis and Smith, 1983, page 76] have called the *connection problem* in distributed problem solving. One assumption they are making is that the set of agents that exist is fixed. We will assume here that an agent exists in a dynamic environment with other agents. As the environment changes new agents might come into existence or existing agents might disappear. Agent autonomy means that an agent has to operate without the direct intervention of humans, i.e. that it has to find out by itself about other agents that exist, specifically, agents that can help it achieve its goals.

[Genesereth and Ketchpel, 1994, page 51] distinguish two basic approaches to the connection problem: direct communication, in which agents handle their own coordination and *assisted coordination*, in which agents rely on special system programs to achieve coordination. Only the latter approach promises the adaptability required to cope with the dynamic environment we envisage.

[Decker *et al.*, 1997] have recently described a solution space to the connection problem based on assisted coordination. The special system programs for coordination are called middle-agents in their analysis. They identify nine different types of middle-agents depending on which agents initially know about capabilities and preferences of agents. By a preference they mean meta-knowledge about what types of information have utility for a requester. In a solution to the connection problem in which capabilities are initially known to the provider and the middle-agent only, and in which preferences are initially known to the requester and the middle-agent only, the middle-agent is what they call a *broker*. Capability brokering is the main problem addressed in this thesis.

The best known work in AI on agent communication is probably the Knowledge-Sharing Effort [Fikes *et al.*, 1991, Neches *et al.*, 1991]. Part of this effort involved the development of the *Knowledge Query and Manipulation Language* (KQML), a high-level agent communication language which we will describe in more detail in section 2.1.2.3. KQML, like most approaches to the connection problem, advocates *assisted coordination through facilitators and mediators*. While the support offered by KQML for this task is still an active research issue, especially for more complex agents [Kuokka and Harada, 1995b], the communication protocol outlined in the language definition does define the basic behaviour a broker must exhibit.

1.1.2 Defining the Problem

Achieving the basic broker behaviour is what the problem of capability brokering is all about.

Definition 1.1 *The* **task of capability brokering** *is to assist other agents in finding agents that can solve a given problem.*

Capability brokering involves communication between different agents. For a specific instance of this problem we shall distinguish three different *types of agents* according to the roles they play for this problem instance:

1. The Problem-Solving Agents (PSAs): These agents provide the general capabilities that may be called upon by other agents in order to solve their problems. A PSA has to advertise its capabilities to the broker agent (see below) and apply these capabilities when requested to do so by other agents.

2. The Problem-Holding Agents (PHAs): These agents have a problem that they wish to have solved by utilising the capabilities of the PSAs. For an instance of the connection problem there is usually only one PHA involved. A PHA has to describe the problem to the broker agent (see below) and wait for the broker to recommend agents that can help.

3. The Broker: The broker matches the problems of the PHA to the capabilities of the PSAs such that the problems can be solved. It receives capability advertisements from the PSAs and stores them. On receipt of a problem description from a PHA it will use the stored capability descriptions to retrieve one or more PSAs that can solve the given problem. Finally, the broker has to either inform the PHA about the PSAs found or manage the solution of the problem for the PHA by interacting with the PSAs directly.

The basic *exchange of messages* between the different agents that has to take place for capability brokering is illustrated in figure 1.1. Since capabilities are meant to be known by the PSA and the broker initially, it is necessary that the PSAs first advertise their capabilities to the broker. Only once a capability has been advertised to the broker can it be used to address the problem of some PHA. At the time of brokering, problems are meant to be known by the PHA and the

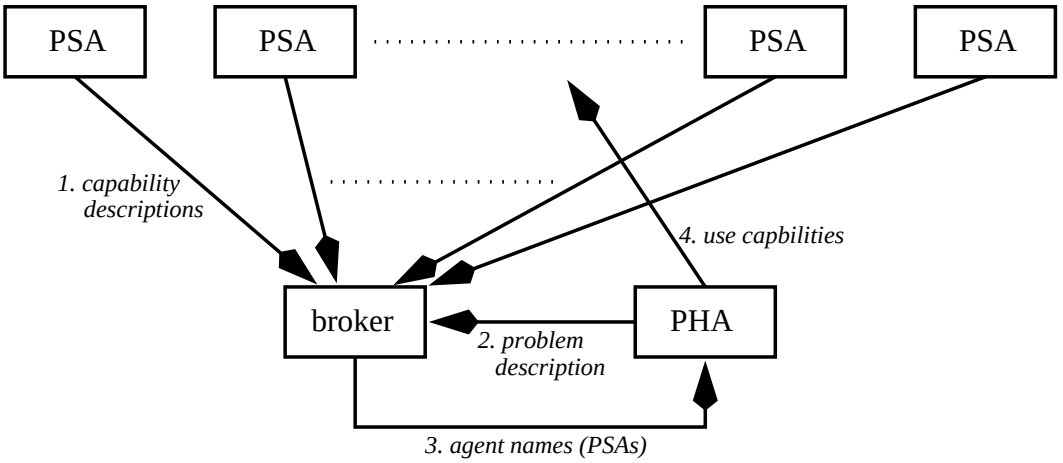

Figure 1.1: Basic message flow in capability brokering

broker and thus, the PHA has to inform the broker about its problem as it arises. If a capability has been advertised to the broker that can be used to address the given problem then the broker should retrieve this capability and inform the PHA about this capability and the agent that has it. Finally, the PHA can use the information from the broker to ask the PSA with the necessary capability to tackle its problem.

As far as the broker is concerned, there are two relatively independent *sub-problems to capability brokering* described in the above message exchange:

- **Capability Storage**: The broker has to store the capability descriptions received by the PSAs. The most important question here is how capabilities can be described or *represented* in a way that is useful to the broker.

- **Capability Retrieval**: The broker has to find PSAs that have the capabilities required to solve the given problem. The most important question here is how capability descriptions can be *reasoned about*.

Not all of the messages outlined above necessarily have to occur in this *order*. The broker should receive at least some capability advertisements from PSAs before it receives a problem from a PHA, but this is not a necessary condition. In fact, capability advertisements from new PSAs could be sent at any time,

i.e. they need not be ordered with respect to the other messages. The remaining messages, the problem description, the reply by the broker, and the utilisation of the capability by the PHA, have to be synchronised though.

To summarise, the problem of capability brokering as defined in definition 1.1 is to achieve the behaviour of the broker outlined in the message exchange schema above.

1.2 Capability Brokering in Context

In this section we will characterise several types of context for capability brokering; we will look at the various combinations of these types and evaluate each in turn to come up with the context for capability brokering that will be used in this thesis. This discussion will clarify what the problem of capability brokering is.

1.2.1 Types of Context

Capability descriptions only make sense in some context. Firstly, there needs to be an agent which has a capability that one wants to describe, even if this agent is only an abstraction in some cases. Capability descriptions are about certain entities. Secondly, there needs to be an agent that wants to evaluate this capability description, be it the described agent for reflective purposes or another agent. We have called these two groups of agents PSAs and PHAs respectively.

The different agents are not the only context for a capability description though. We will now look at additional kinds of context in which capability descriptions can be found.

1.2.1.1 Common Languages and Expressiveness

We will make the assumption here that it is a necessary prerequisite for agents to be able to communicate with each other in order to be able to work together. If the PHA and the PSA "speak" a common language then communication is possible in principle. A more general condition could be to require them to speak equally expressive languages and to necessitate the existence of a translator. We see the ability to communicate in a certain language as (a trivial) part of a capability description.

1.2.1.2 Compile-Time vs. Run-Time Evaluation

Suppose the PHA is looking for a PSA that it wants to use in future to solve a certain type of problem, i.e. the search is done only once. We will call the evaluation of capability descriptions in this context evaluation at compile-time. If the PHA is seeking a PSA every time is has a problem then we will call this evaluation at run-time.

This distinction has two major effects on the capability description required. Firstly, for evaluation at run-time there is a specific problem instance available. To make use of this in the evaluation process the capability description must include sufficient detail and can be expected to use mainly domain terminology. A capability description intended for evaluation at compile-time might have a different emphasis, i.e. it might contain more general information. Secondly, the efficiency with which one expects the description to be evaluable can be very different. Evaluation at compile-time may be slow as it is only done once. Evaluation at run-time may, for example, need to be faster than the average time it takes for the PSA to succeed or fail in an attempt to solve the problem at hand.

1.2.1.3 Agents with or without Domain Knowledge

Another distinction has to be made according to the type of PSA the capability description describes: agents with or without domain knowledge. For example, a general diagnostic agent does not have any domain-specific knowledge whereas a medical diagnostic agent does.

The difference we would expect to see in the respective capability descriptions for agents with or without domain knowledge is linked to the terminology used. Descriptions of agents with domain knowledge will contain domain terminology; they will focus on what problems the agent can address. Descriptions of agents without domain knowledge cannot include domain terminology; they will focus on how the problem is being solved.

1.2.1.4 Coarse-Grained vs. Fine-Grained Agents

Finally, a PSA may be capable of solving the given problem completely. This is what we call a coarse-grained agent as the problem grain-size it deals with is the whole problem. If the agent only contributes a small step towards a solution we will call this a fine-grained agent, as the problem grain-size it deals with is very small. For example, an AI planner deals with a planning problem all at once whereas a temporal constraint manager only deals with some part of this problem. Notice that grain-size depends on the actual problem.

We expect to see that the main effect of this distinction in capability descriptions lies again in the efficiency with which an evaluation is possible. Small steps require relatively little time and the evaluation hence may need to be relatively fast. But this is not the only effect; capability descriptions of fine-grained agents must also include information about expected utility to aid the decision as to whether the agent should be applied or not.

1.2.2 Combinations of the Criteria

What will a meaningful context for capability description and assessment look like in terms of the above criteria? To answer this question, we will look at different ways to combine the values the above criteria can take. We will not consider expressiveness as the values this dimension can take are not known. Grainedness is a dimension with a spectrum of values from complete problem solvers to primitive inference actions and we will mainly consider these extremes.

- **Evaluation at compile-time**:
  - **Choosing from coarse-grained agents**:

    Whether or not the agent has domain knowledge is not important here. The context here is that we are trying to select a problem solver from a set of agents that can all individually deal with the type of problem

we expect to have. This selection is to be made only once, i.e. for a number of future problems.

Why do we need formal capability descriptions in this context? Firstly, it seems questionable to develop a formal capability model for an agent if this is only to be used for the initial choice. Secondly, as the exact problems to be dealt with are unknown at the time the capability description is evaluated, a realistic decision will often be based on rather subjective criteria that are hard to model. We think that a formal capability description in this context would offer little benefit.

- **Choosing from fine-grained agents**:

  Again, the availability of domain knowledge is not important for the argument here. In this context, the aim is to assemble a system from the known capability components.

  In general this context resembles the task of automatic programming. Whereas it is a very worthwhile aim for capability descriptions, it seems that it is also too ambitious at this point. Solving the problem of automatic program generation is unlikely to be achieved solely by developing better capability description languages. This context would only be reasonable under certain assumptions about the solution.

- **Evaluation at run-time**:

  - **Choosing from agents without domain knowledge**:

    We will immediately dismiss this context as it seems unlikely to us that there will be intelligent agents, be it components or complete problem solvers, that do not have at least some domain knowledge at run-time.

  - **Choosing from agents with domain knowledge**:

    * **Choosing from fine-grained agents**:

      The context here is that we have a PHA with a problem at hand and a PSA that might potentially contribute to the solution of this

problem. The aim of the capability description at this point must be to aid in the decision as to whether this primitive inference action should be applied next or not.

This context very much resembles the decision a search controller is frequently faced with. Unfortunately, it appears that only highly relevant and efficiently evaluable capability descriptions would be useful here. As a generic capability description language is unlikely to fulfill these criteria, this context does not seem promising.

* **Choosing from coarse-grained agents**:

  The context here is again that we have a PHA with a problem at hand, but this time we want to use the capability description of other agents to chose one of them that can solve our problem rather than just contribute to the solution. We believe that this is a promising context for the development of a capability description language.

The most fruitful context for the development of a capability description framework by far seems to be the context in which the evaluation of the capability description takes place at run-time, the agents to choose from have domain knowledge, and are coarse-grained, i.e. can solve the given problem.

1.3 Criteria for Success

In this section we will discuss what it means to successfully address the problem of capability brokering. The criteria for success described here will be used for a critical evaluation of the work presented in this thesis in chapters 6 and 9.

Addressing the problem of capability brokering means developing a broker agent that shows the behaviour outlined above. That is, it has to process capability advertising messages and store described capabilities. Later, when a request to recommend a PSA for a given problem arrives, the broker has to search through the capabilities previously advertised to find an appropriate capability for the problem and forward this capability to the PHA. To show that a broker actually achieves this behaviour, our first criterion for success shall be that we can implement a *working prototype* of such a broker.

More specifically, we shall outline a number of scenarios in chapter 3 that describe the messages the broker has to process and generate. We expect the working prototype of the broker to be able to *cope with all of these scenarios*. Furthermore, the broker should also be able to cope with at least a significant portion of variations of these scenarios to show some degree of robustness. However, we do not expect the prototype implementation to be fully debugged and tested or have a well developed user interface.

To be potentially useful in a realistic environment, it will be necessary for the broker to satisfy certain *performance criteria*. Most critical here is the time it takes the broker to respond to a request from a PHA, i.e. the capability retrieval time. The question is what a reasonable response time is. This depends on the time the PSA will take to solve the given problem. If the capability retrieval time is at least an order of magnitude faster than the time the PSA will take to solve the problem, the response time should be adequate. This shall be our criterion for success regarding broker performance.

The criteria for success up to now have all been concerned with the practical aspect of the work presented in this thesis. There is, however, a more theoretical side and criteria for success in this respect need to be defined, too.

The theoretical part of the work described in this thesis centres around the two properties we expect our capability description language to have: expressiveness and flexibility. *Expressiveness* is a property that has been defined in a number of ways for different representations. To show that our capability description language is expressive we thus need to compare it to other knowledge representation languages that were designed for the representation of similar types of entities. Our criterion for success here is that our language should be at least as expressive as most other languages in its class.

The story is quite different for the other property we claim our language to have: *flexibility*. As far as we know there has never been an attempt to formally define what is meant by flexibility. In fact, there are very few languages that have this property. The main problem here is not how to define flexibility though, but how to implement a language that has this property. There are a number of issues arising in this context and our criterion for success here shall be that our language addresses these issue in a way that compares favourably with other flexible languages.

Finally, we need to say a few words about the *brokering mechanism*. We have already defined one related criterion for success: performance, but this shall not be enough. We also want to compare our broker with other generic brokers and we expect that our broker should offer at least those features offered by the other brokers that are required for capability brokering and our scenarios. This shall be our final criterion for success.

Chapter 2

Capability Brokering: A Literature Survey

At this point the general problem of capability brokering that is to be addressed in this thesis has been described and discussed. Our aim is to address this problem with a new capability description language that will be expressive and highly flexible and can be used to reason about capabilities. The next step towards this goal will be to investigate how previous approaches to representing generic capabilities attempted to do so. The contribution of this chapter will be a broad review of approaches to representing and reasoning about knowledge similar to the capability knowledge we need to represent.

2.1 Software Agents

In this section we will look at work in the wider area of intelligent software agents and, more specifically, at approaches to capability brokering found there.

An overview of this section, which provides a conceptualisation of the relationships between the different sub-fields and approaches/systems described in this section, is given in figure 2.1. The figure also contains cross-links to other areas. The most important work for this thesis reviewed in this section includes:

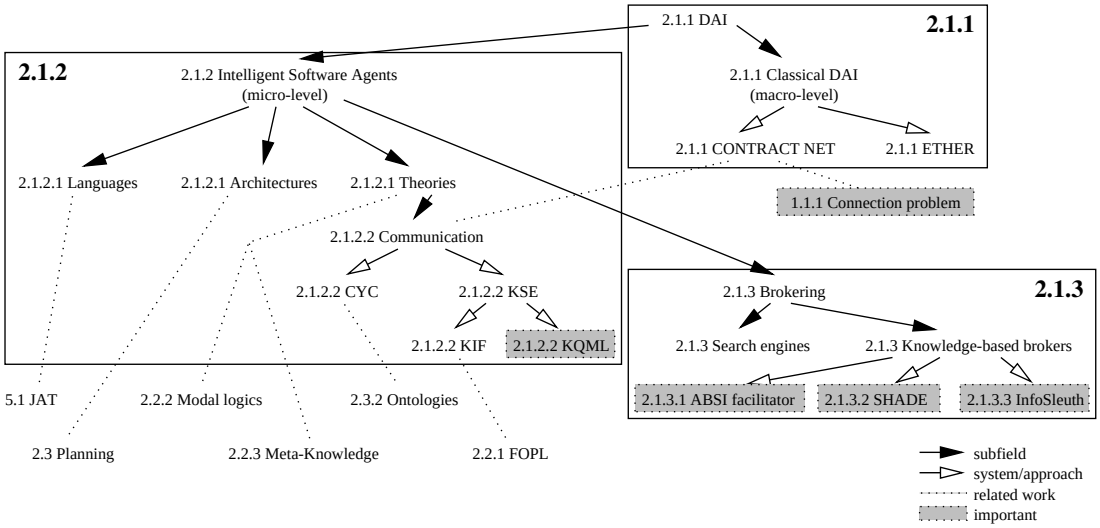

Figure 2.1: Overview of this section

the connection problem defined in the work on the CONTRACT NET; the generic agent communication language KQML; and the brokers described at the end of this section. These areas are also highlighted in figure 2.1 to stress their importance.

2.1.1 Distributed AI

Intelligent software agents are often seen as part of a wider area of *Distributed Artificial Intelligence* (DAI) [Bond and Gasser, 1988, Chaib-Draa *et al.*, 1992, Jennings, 1996], which motivates us to briefly consider this area first. DAI is the subfield of AI that is interested in concurrency in AI computations. Its main concerns have been *distributed problem solving*, i.e. how the task of solving a particular problem can be divided amongst a number of available problem solvers, and *multi-agent systems*, i.e. how a collection of autonomous intelligent agents can coordinate their knowledge, goals, skills, and plans jointly to take action or to solve problems.

DAI has not been very concerned with the problem of capability brokering. As pointed out in [Wooldridge and Jennings, 1995, page 142], the classical emphasis in DAI has mostly been on the *macro-level*, i.e. on social phenomena and the emergent behaviour of a group of problem solvers. This level is of little interest

to us in this thesis. Research in intelligent software agents emphasises the *micro-level*, i.e. the architecture and theories of individual agents. The latter is where the problem of capability brokering has been addressed previously and at which we will look in section 2.1.2.

Two architectures that grew out of DAI are worth mentioning here. Firstly, there is the CONTRACT NET [Smith, 1977, Davis and Smith, 1983]. In the CONTRACT NET architecture a given problem is first decomposed into sub-problems. These sub-problems are treated as contracts and a process consisting of contract announcement, bidding, and contract awarding is used to distribute problem solving. This process of negotiation, i.e. the extensive and explicit use of communication to distribute the problem (cf. section 2.1.2.2), was an important contribution of this work. Another contribution was the definition of the connection problem [Davis and Smith, 1983, page 76] which is essentially the problem we are addressing in this thesis (cf. section 1.1.1).

Secondly, there is ETHER [Kornfeld, 1979, Kornfeld, 1981], a pattern-directed invocation formalism for parallel problem solving. ETHER provides a PLANNER-like language where procedure invocation is driven by pattern matching. Unlike in previous approaches, control over the distribution is not in the hands of the user. Instead, the patterns are used to distribute the problem-solving process. The basic mechanism for the distribution is by broadcasting of patterns. Nowadays virtually all generic brokers use patterns to distribute the problem-solving process (cf. sections 2.1.3 and 9.1), but the specifics of ETHER are not used anymore which is why we will not look at it any further.

2.1.2 Intelligent Software Agents

Intelligent software agents have recently received a lot of attention within AI [Russell and Norvig, 1995, Bradshaw, 1997, Huhns and Singh, 1998]. However, the definition of *agent* is elusive, i.e. there is still considerable lack of consensus on what exactly an agent is or what the research questions are that need to be

addressed. An overview of possible definitions of agency and a comprehensive structuring of the field has been presented in [Wooldridge and Jennings, 1995] and we shall mostly adopt their approach. They distinguish agent theories, agent architectures, and agent languages as the three major subfields of agent research.

2.1.2.1 Languages, Architectures and Theories

Firstly, the subfield of *agent languages* is mainly concerned with tools that allow one to program hardware or software computer systems using the concepts developed in agent theories as outlined below. Such tools include, for example, the Agent Behaviour Language [Wavish, 1992], the agent-oriented programming paradigm [Shoham, 1993], or Concurrent METATEM [Fisher, 1994]. As this area is not concerned with the representation of and reasoning about capabilities, we shall not dwell on it here. We have, however, chosen the Java Agent Template (JAT) to implement the agents to be presented in this thesis and we will describe JAT in section 5.3 in only as much detail as necessary. The particular choice of JAT is of little relevance as none of the tools mentioned above support brokering of capabilities in any way.

Secondly, the subfield of *agent architectures* is concerned with issues surrounding the construction of computer systems that satisfy the properties specified by agent theories (below). The classical approach in AI is the deliberative architecture based on the physical symbol system hypothesis, i.e. an architecture that contains an explicitly represented, symbolic model of the world [Newell and Simon, 1976, Russell and Norvig, 1995]. The main alternative to the deliberative approach is the reactive approach based on the so-called subsumption architecture [Brooks, 1986, Brooks, 1991]. Finally, a number of hybrid approaches to agent architectures have also been attempted. However, none of these architectures explicitly supports capability brokering. Since deliberative agents will need to take well-planned actions it is often assumed that such an agent should be based on AI planning technology [Wooldridge and Jennings, 1995, page

131]. We will look at AI planning more closely in section 2.3 and at existing agents using a planner specifically in section 2.3.4.

Finally, formal *agent theories* are essentially specifications for agents where an agent is described as an intentional system that has beliefs, desires, etc. [Seel, 1989]. Agent theories can be seen as representational frameworks for such attitudes. The dominant approaches are based on modal logics (cf. section 2.2.2) and meta-languages (cf. section 2.2.3). The former lead to the adoption of the possible worlds semantics which has been used to define what it means for an agent to know something and to reason about knowledge and belief [Hintikka, 1962, Kripke, 1963]. Various alternatives were also developed to avoid the problem of logical omniscience [Levesque, 1984, Konolige, 1986]. Similarly, but to a lesser extent, there have been logics of goals or desires [Cohen and Levesque, 1990, Wooldridge, 1994]. Although these approaches have addressed many attitudes of agents, there remains the problem of integrating them into one framework for an all-embracing agent theory. The issue in agent theories we shall be most concerned with here is that of *agent communication* which also addresses the connection problem.

2.1.2.2 Agent Communication

At least two major efforts are currently under way which both assume *knowledge sharing* to be the key to successful agent communication and cooperation.

The first effort addressing the agent communication problem is the CYC project [Guha and Lenat, 1990, Guha and Lenat, 1994, Lenat, 1995]. The basic idea here is that agents need to have a large amount of commonsense knowledge before they can intelligently work together. Since the CYC researchers believe that commonsense knowledge cannot be learned automatically without having a large body of it in the first place, most of the work in CYC has been on hand coding such knowledge and on developing large ontologies using micro-theories. We shall return to the issue of ontologies in section 2.3.2.

The second major effort addressing the agent communication problem is the *ARPA Knowledge Sharing Effort* [Fikes *et al.*, 1991, Neches *et al.*, 1991, Genesereth and Ketchpel, 1994]. They envisage a generic agent communication language as consisting of three parts: the vocabulary, the inner language which carries the content that is being communicated, and the outer language which represents mainly the speech act that this message represents. The vocabulary is to be defined within one or more ontologies that will be shared by the communicating agents [Gruber, 1993b, Gruber, 1993a, Farquahar *et al.*, 1996]. Again, we shall return to the issue of ontologies in section 2.3.2. A generic knowledge representation language called KIF [Genesereth, 1991, Genesereth *et al.*, 1992] to and from which all other content languages should be translatable has been suggested for the content to be communicated, including the content of messages about capabilities (cf. section 2.1.3).

2.1.2.3 The Knowledge Query and Manipulation Language

Research on the outer language mentioned above (section 2.1.2.2) has resulted in the definition of the *Knowledge Query and Manipulation Language* (KQML) [Finin *et al.*, 1992, Finin *et al.*, 1997, Labrou and Finin, 1997]. All the agents described in this thesis use KQML for inter-agent communication and, hence, it is necessary to describe KQML in some detail at this point.

The syntax of KQML is simply based on a balanced parenthesis list. The first element in this list represents the *performative* of this message[1]. The performative indicates the type of speech act this message is. For example, the performative `ask` indicates a question being asked and the performative `tell` indicates a statement being made. For each performative in KQML there is also a protocol that defines with which type of messages other agents should reply to this message, if any.[2] For example, there should always be a reply to an `ask`-message and the

[1] In the literature on KQML and speech acts the term *performative* is sometimes also used to refer to the whole message.

[2] There is currently no agreed formal semantics for KQML available [Cohen and Levesque, 1995].

performative of this reply message should be `tell`. Although there is a set of predefined performatives in KQML it is not meant to be binding. Agents may choose to use this set or invent their own performatives. They may also choose not to implement certain predefined performatives. However, if a predefined performative is used it should be used with the protocol for this performative defined in the KQML specification.

The performative is followed by a number of *keyword-value pairs*. Again, there is a number of predefined keywords like `:sender` or `:content` that all have a fairly obvious meaning. For example, the value following the keyword `:sender` should be the name of the agent sending this message and the value following the keyword `:content` should be the actual content of the message. The content of a KQML message is meant to be opaque to the message, i.e. an interpreter is not supposed to inspect the content while interpreting the message. However, in interpreting a KQML message it is necessary to decide where the content ends and thus, it is necessary to look at the content at least for this. There are also a number of fairly obvious constraints between the different parts of a KQML message. For example, if the language field names a specific content language then the content should be in this language.

KQML, like most approaches to the connection problem, advocates assisted coordination through agents called *facilitators* and *mediators*. A facilitator in KQML is an agent that performs various useful communication services. One of the main services offered by a facilitator is to help other agents find appropriate clients and servers. How client and server agents can find the facilitators is a problem to which KQML does not prescribe a solution. Neither is the mechanism to be used by the facilitators to find appropriate servers for clients specified in KQML. However, there are a number of related performatives and protocols for these performatives that can be seen as the definition of an interface to the facilitators. This interface definition is one of the most important contributions of KQML as far as capability brokering is concerned. Some of the most important performatives

- **advertise**: With this performative the sender informs the receiver (which should be the facilitator) that the sender is willing and able to process certain messages. KQML specifies that the processable message being advertised is given as the content of this message, i.e. the content is a KQML message again. Furthermore, the performative of the content message should be one of a limited set and there are certain basic constraints on the sender and receiver of the advertisement and embedded message. No reply message is required.
- **subscribe**: With this performative the sender informs the receiver (which should be the facilitator) that it wants to be updated every time that the would-be response to the content message is different from the last response to the sender of this message. Thus, like for `advertise` the content must be a KQML message and similar constraints apply. In response, the facilitator should send one reply to the embedded message immediately and further messages as they occur.
- **recommend-one**: With this performative the sender informs the receiver (which should be the facilitator) that it wants to know about one agent that has advertised that it will process the message given as the content of this message. The expected reply to this message is a message with the performative `forward`, the content of which should be an advertising message. The content of this `recommend-one` message and the `advertise` message should be identical.
- **recommend-all**: This performative is like `recommend-one`, only that the reply should name all the agents that have advertised to process the given content message.
- **broker-one**: Again, this performative is like `recommend-one` in its form. The difference is that with this performative the sender asks the facilitator to find an agent that can process the given message and then send it the given message. If there will be a reply to this message, this reply should be forwarded to the sender of the `broker-one` message.
- **recruit-one**: Again, this performative is like `recommend-one` in its form. The difference is that with this performative the sender asks the facilitator to find an agent that can process the given message and then send it the given message. The difference to `broker-one` is that any reply should go directly to the sender of the `recruit-one` message rather than through the facilitator.

Table 2.1: KQML facilitation performatives [Labrou and Finin, 1997]

for brokering in KQML are described in table 2.1. As mentioned above, all the agents described in this thesis use KQML for inter-agent communication and thus, many example messages using these performatives will follow in the remainder of this thesis.

One issue worth noting at this point is that KQML requires the content of an advertisement to be identical to the content of the capability-seeking message. This is very restrictive and, as we shall see, most existing brokers ignore this part of the KQML specification and provide a more sophisticated matching service.

2.1.3 Brokering Agents

Returning to the connection problem, which is the main problem of capability brokering, [Genesereth and Ketchpel, 1994] distinguish two basic approaches to this problem: *direct communication*, in which agents handle their own coordination and *assisted coordination*, in which agents rely on special system programs to achieve coordination. ETHER and the CONTRACT NET, both described in section 2.1.1, fall into the first category. The facilitation approach defined in KQML falls into the second category and this is currently the dominant approach in intelligent agent research.

Before we turn to a survey of existing brokers that facilitate assisted coordination, it is also worth noting that a kind of brokering has been performed on the Internet for some time now by *search engines* [Witten *et al.*, 1994, Howe and Dreilinger, 1997]. However, most search engines match requests, usually only consisting of a few keywords, to text pages on the Internet. The matching is essentially based on a reverse word frequency count algorithm[3] and can hardly be called knowledge-based. This is not the kind of brokering we are interested in here.

Various terms have been used for the special system programs on which assisted coordination relies, some of which we have used in this review, e.g. facil-

[3] Actually, there are also other techniques which are being used in search engines, but none of them is based on what can be considered an understanding of the retrieved document.

preferences initially known by	*capabilities initially known by*		
	provider only	provider + middle agent	provider + middle + requester
requester only	(broadcaster)	"front-agent"	matchmaker/ yellow-pages
requester + middle agent	anonymizer	broker	recommender
requester + middle + provider	blackboard	introducer/ bodyguard	arbitrator

Table 2.2: Middle-agent roles; from [Decker *et al.*, 1997, page 579]

itator or broker. [Decker *et al.*, 1997] have recently suggested a categorisation of what they call *middle-agents*. They use the term middle-agent to mean any special system program used in assisted coordination. They distinguish different kinds of middle-agents according to where preference and capability knowledge resides. Preferences are meta-knowledge about what types of information have utility for a requester and capabilities are meta-knowledge about what types of requests can be serviced by a provider. The table summarising their categorisation is repeated here in table 2.2. According to this categorisation, in a scenario in which the capabilities of problem-solving agents are initially only known to the provider and the middle-agent and the problem of the problem-holding agent are initially only known to the requester and the middle-agent, the middle-agent is called a *broker*.

Brokers are the kind of middle-agent we are most interested in looking at in this thesis. We shall now briefly review some brokers that use explicit representations as the basis for brokering. A detailed comparison of these brokers with the broker developed in this thesis will follow in section 9.1.

2.1.3.1 The ABSI Facilitator

One of the earliest middle-agents that can be considered to be a broker in the above sense is the Agent-Based Software Interaction (ABSI) *facilitator* [Singh, 1993a, Singh, 1993b]. This broker is meant to be used in a system of agents operating in the ABSI architecture [Genesereth and Singh, 1993] and is based on a variant of an early specification of KQML [Finin *et al.*, 1993]. For the facilitator to perform its brokering service, agents must first notify the facilitator of the KQML messages they can process, i.e. they must advertise their capabilities. One important restriction imposed by the ABSI facilitator is that agents must not advertise that they can handle a message which they might subsequently fail to process.

The ABSI facilitator provides performatives for package forwarding, information monitoring, and content-based routing. Content-based routing is basically what we have called capability brokering in section 1.1.2. Of the KQML brokering performatives described in table 2.1, the ABSI facilitator essentially supports `advertise` and `broker-one`. Capabilities are represented by KQML messages as defined in the KQML specification. The content of these capability-representing KQML messages must be in KIF. For capability retrieval, the content of a capability-seeking message and the capability advertisement need not be identical for them to match, as the KQML specification would require. Instead a kind of unification defined by meta-descriptions in the KIF manual [Genesereth *et al.*, 1992] is used to match capabilities and preferences. Additionally, a Prolog-based inference engine can be used to evaluate additional conditions on the matched meta-variables.

2.1.3.2 SHADE and COINS

Two other brokers based on the KQML protocol are the SHADE and COINS matchmakers [Kuokka and Harada, 1995a, Kuokka and Harada, 1995b]. These brokers are implemented entirely as a declarative rule-based program within the MAX

forward-chaining agent architecture [Kuokka, 1990]. As opposed to the ABSI facilitator, it is assumed that SHADE and COINS will make false positive and false negative matches. Thus, part of the work on these brokers was on addressing the problem of recovery after such a false match.

Both, SHADE and COINS, support the full range of KQML performatives described in table 2.1 and more. The difference between SHADE and COINS lies in the capability representations they can handle. In both cases, capabilities are represented as KQML messages, but SHADE works over a formal, logic-based content language and COINS operates over free-text information. Thus, COINS is effectively what we have called a search engine above. SHADE expects either KIF [Genesereth *et al.*, 1992] or MAX [Kuokka, 1990] augmented to support string patterns as terms for its content language. MAX is more appropriate for representing highly structured data such as objects or frames. The actual matching of capabilities and preferences is performed by a Prolog-like unification algorithm. Advertisements and requests must match based solely on their content; there is no knowledge base against which inference is performed. Limited inference for future versions is envisaged though.

2.1.3.3 InfoSleuth

The aim of the *InfoSleuth* project [Bayardo *et al.*, 1997, Nodine and Unruh, 1997, Nodine *et al.*, 1998] is to develop technologies that operate on heterogeneous information sources in an open, dynamic environment. To achieve this flexibility and openness, InfoSleuth integrates agent technology, ontologies, information brokerage, and Internet computing. InfoSleuth's architecture is comprised of a network of cooperating agents communicating in KQML. One of these agents is the broker agent which receives and stores capability advertisements from all other InfoSleuth agents. The task of the broker agent is to provide a semantic match-making service that pairs agents seeking a particular service with agents that can perform that service.

Minimally, every agent must advertise to the broker its location, name, and the language it speaks. Queries must be in KQML using KIF as the content language and "InfoSleuth" as the ontology. Matching is performed as an intersection function between the user query and the data resource constraints in the capability advertisement. The way this works is that KIF sentences, that are the content of capability advertisements and user queries, are translated into the deductive database language LDL++ [Zaniolo, 1991] and maintained in such a database.

Most Important Issues Here

- The work on the CONTRACT NET described in section 2.1.1 gave us the connection problem which is basically the problem addressed in this thesis.
- The inter-agent communication language KQML described in section 2.1.2.3 is probably the most advanced language for this purpose and used for all agents developed in this thesis.
- The KQML-based brokers described in section 2.1.3 perform essentially the same task as the broker that will be described in this thesis and a detailed comparison will follow in section 9.1.

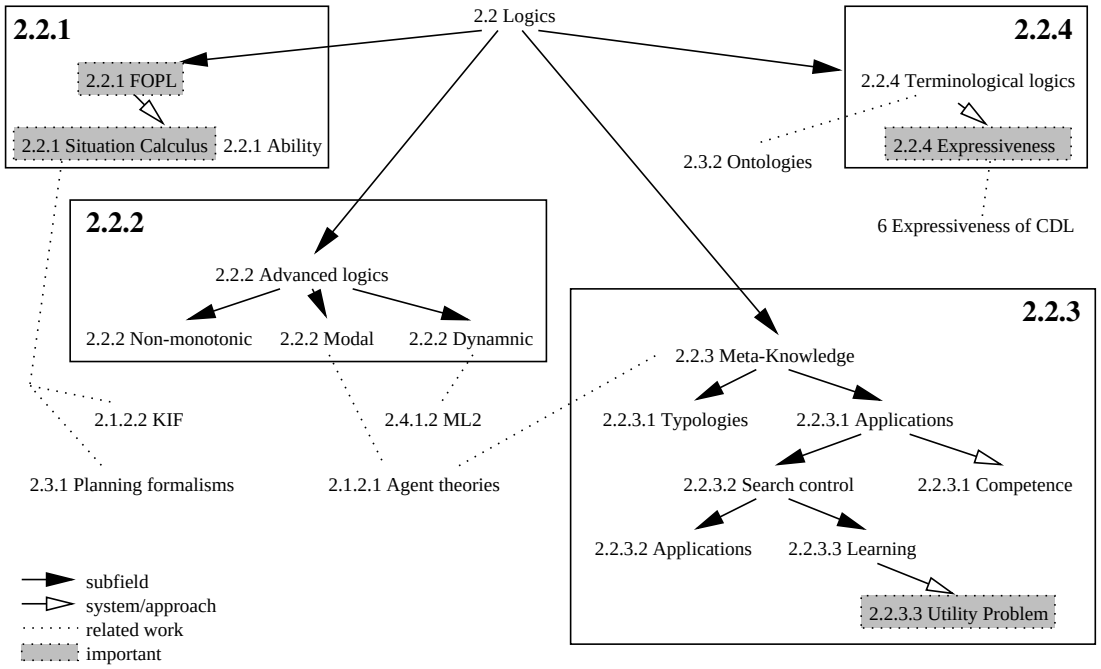

Figure 2.2: Overview of this section

2.2 Modelling Capabilities with Logics

In this section we will look at how some logics have been or could be used to represent the capabilities of intelligent agents.

An overview of this section, which provides a conceptualisation of the relationships between the different sub-fields and approaches/systems described in this section, is given in figure 2.2. The most important areas for our work introduced here are: first-order logic which can be seen as a generic knowledge representation formalism and has been used to represent actions in the situation calculus; representations for meta-level knowledge and the closely connected utility problem; and the theory of expressiveness developed for terminological knowledge representations.

2.2.1 First-Order Predicate Logic

A generic knowledge representation formalism such as *first-order predicate logic* (FOPL) [Chang and Lee, 1973, Loveland, 1978, Gallier, 1986] might well have

turned out to be sufficient for representing and reasoning about capabilities, i.e. what we want to do in this thesis. Advantages of FOPL include its well-defined semantics and the fact that it is probably the best-researched knowledge representation formalism in AI and beyond. This is certainly good enough a reason for us to begin our review of logics as capability representations with FOPL. Furthermore, KIF (cf. section 2.1.2.2), which is supported as a content language by most brokers (cf. section 2.1.3), is essentially a variant of FOPL.

Representations of capabilities in FOPL have been attempted in early approaches to reasoning about actions, e.g. [Green, 1969]. One of these early approaches is the *situation calculus* [McCarthy and Hayes, 1969, Shanahan, 1997] which has actually been an active topic of AI research for more than three decades now. However, its main concern has not been with reasoning about capabilities but with reasoning about actions in general, which can be seen as a much broader area than reasoning about capabilities for brokering.

Very briefly, the ontology of the situation calculus is made up of *situations* which can be thought of as snapshots of some world; *fluents*, which take on different values in different situations and can be thought of as time-varying properties; and *actions*, which can be executed to change the value of fluents. The atomic formula $Holds(f, s)$ is used to denote that the fluent f is true in situation s. Note that the fluent f, although it might look like an atomic formula, is a term, i.e. it represents an object in the domain represented. The function term $Result(a, s)$ is used to denote the situation obtained by executing the action a in the situation s. Sentences in first-order logic called effect axioms can now be written to represent the effects and preconditions of actions.

Unfortunately the effect axioms alone turn out to be epistemologically inadequate and further so-called frame axioms are needed in the representation, leading to the *frame problem* in AI [Hayes, 1974, page 69], [Shanahan, 1997]. Furthermore, the representation of fluents as objects in the domain seems counterintuitive. In summary, the strong point of the situation calculus has traditionally

been the theoretical framework it provides for the representation of actions based on a well-defined semantics.[4] A number of more direct action representations which also address the frame problem have been proposed in AI and we shall review some of them in section 2.3.1. Still, the situation calculus remains a highly expressive action representation with probably the clearest semantics of any such representation.

McCarthy and Hayes' original work was not limited to the representation of and reasoning about actions and their effects, but also included the general concept of *ability* [McCarthy and Hayes, 1969, pages 470–477]. In this work, they have attempted to formalise what it means for an agent to be able to do something by defining a predicate $can(p, \pi, s)$ meaning "agent[5] p can bring about the condition π in situation s." The interesting result here is, as they point out, that it is not at all clear what this proposition means. However, although highly relevant for the epistemological underpinnings of our work, we shall not go into the philosophical problems entailed here.

2.2.2 Advanced Logics

Since we have mentioned the situation calculus and the frame problem, it is also worth noting that there is a group of logics that have been mainly developed to address this problem. These logics are referred to as *nonmonotonic logics* [Ginsberg, 1987, Brewka, 1991], [Davis, 1990, section 3.1]. The classic approaches here are Default Logic [Reiter, 1980] and Circumscription [McCarthy, 1980b, McCarthy, 1980a]. However, as these approaches address the problem by changing the inference mechanism rather than fundamentally changing the representation, they are of little interest to us and we shall not look at them further here.

[4] Recent work described in [Gruninger and Fox, 1994, Gruninger *et al.*, 1997] addresses some practical aspects of reasoning with a formal situation calculus.

[5] They are looking at a world of interacting discrete finite automata for which we will use the term agent here.

Many approaches to agent theories (cf. section 2.1.2.1) are based on *modal logics* [Chellas, 1980, Chagrov and Zakharyaschev, 1997], [Davis, 1990, section 2.7] and the possible worlds semantics [Hintikka, 1962, Kripke, 1963] and thus, we shall have a look at these logics next. Agent theories are specifications of agents. These specifications can be used by agents to reason about other agents. Our aim is to reason about the capabilities of other agents.

A modal logic augments a calculus, e.g. predicate calculus, with a number of operators, called *modal operators*, that take sentential arguments. Modal operators are usually non-extensional, i.e. they do not commute with quantifiers, or are referentially opaque. The semantics of a modal language is based on Kripke structures which consist of a collection of possible worlds, connected by an accessibility relation. We say a possible world $\mathcal{W}_1$ is accessible from a possible world $\mathcal{W}_2$ in a Kripke structure if they are connected by the accessibility relation. In each possible world, a sentence in modal logic can be either true or false, i.e. a sentence may have different truth values in different possible worlds.

For example, in propositional modal logic the truth values of propositions can vary across different possible worlds. Propositions can be connected with the usual connectives (e.g. negation, conjunction, disjunction) to form more complex sentences. The only syntactical extension is the introduction of usually two new, dual types of sentences: $\Box A$ (*necessarily* A) and $\Diamond A$ (*possibly* A), where A is again a sentence in modal logic. The informal semantics is that $\Box A$ is true in a possible world $\mathcal{W}$ if and only if A is true in every possible world accessible from $\mathcal{W}$ and that $\Diamond A$ is true in a possible world $\mathcal{W}$ if and only if A is true in at least one possible world accessible from $\mathcal{W}$. Other modal operators may also be defined.

Modal logics have been used in agent theories mostly to reason about the knowledge of other agents [Wooldridge and Jennings, 1995, section 2]. This is done by interpreting $\Box A$ as a modal knowledge operator, i.e. an agent *knows* A if in every world that is consistent with its knowledge, A is true. Reasoning about

knowledge using modal logics was probably first comprehensively integrated into a framework for reasoning about actions by [Moore, 1985].

A further extension of modal logics are *dynamic logics* [Harel *et al.*, 1982, Harel, 1984]. Dynamic logics were developed to reason about programs and their executions. Syntactically, the only change from normal modal logic is that $\Box A$ is replaced by $[\alpha]A$ and $\Diamond A$ is replaced by $\langle\alpha\rangle A$, where α is a program. A program implicitly defines an accessibility relation, i.e. only those worlds are accessible that are possible states after the execution of the program. $[\alpha]A$ is then defined as true in $\mathcal{W}$ if and only if A is true in every possible world accessible from $\mathcal{W}$ with the accessibility relation defined by α, i.e. if A is necessarily true after the execution of α.

A program is a sequence of performable actions and thus, programs can be seen as capability descriptions. This means that dynamic logics are the first logics introduced here that explicitly include capabilities in the form of programs in their ontology. However, representing knowledge in and automated reasoning over dynamic logics has proven not very practical and thus, we shall not pursue this path any further.

2.2.3 Meta-Level Knowledge

Experiments in [Larkin *et al.*, 1980, Chi *et al.*, 1981], and other work described in [Barr, 1979, Anderson, 1981], have shown that experts in a field often do not have more domain knowledge than novices, but instead they use this knowledge more efficiently; they have more *meta-knowledge*. Being an expert in a domain means to be more competent in this domain or, to be more capable of solving problems in this domain. Thus, there is a strong correlation between the availability of meta-knowledge and capability or competence in a domain. Similar arguments can be found in [Laske, 1986, Lecoeuche *et al.*, 1996, VanLehn and Jones, 1991]. We have argued in [Wickler and Pryor, 1996] that available meta-knowledge can be re-used for competence assessment. The emphasis here is on the re-use aspect

which would make this approach very attractive to our aims as it could save us a lot of work. Thus, we shall now look at meta-knowledge and its representations to see whether this knowledge can be re-used for capability brokering.

2.2.3.1 Types of Meta-Level Knowledge

A number of *classifications of meta-level knowledge* have been attempted. For example, an early classification by [Davis and Buchanan, 1977] distinguishes schemata for reasoning about objects, function templates for reasoning about functions, rule models for reasoning about inference rules, and meta-rules for reasoning about strategies. In [Lenat *et al.*, 1983] there is not so much a categorisation of meta-knowledge, but instead they give a number of examples where such knowledge is being used. These examples include meta-knowledge: for rule selection; to record needed facts about knowledge; for rule justifications; to detect bugs; etc. The last example they give concerns meta-knowledge to describe a program's abilities. Unfortunately they do not describe a representation for this type of meta-knowledge. Similarly, [Maes, 1986] argues that meta-level knowledge is needed for introspection and classifies it by the tasks it is needed for, e.g. in assumption-based reasoning, in learning, in handling inconsistent, incomplete, and uncertain knowledge etc. This shows that there is a need for explicit meta-knowledge in knowledge-based systems.

There are also a number of *examples of systems* that have used explicit meta-knowledge for a number of purposes. For example, [Filman *et al.*, 1983] describe several experiments using meta-language and meta-reasoning to solve problems involving belief, heuristics, and points of view; [Attardi and Simi, 1984] describe a meta-language for reasoning about logical consequence; [Ginsberg, 1986] describes a meta-level framework for the construction of knowledge base refinement systems; [Haggith, 1995] describes a framework for reasoning about conflicts in knowledge bases. Many other systems using explicit meta-knowledge do exist (cf. [Maes and Nardi, 1988]). This illustrates the availability of meta-knowledge

in knowledge-based systems.

Of particular interest to us is work on using meta-knowledge for *competence assessment* [Voß *et al.*, 1990] as this is directly related to capability retrieval. One of their aims was to develop a system that knows when it cannot solve a given problem without having to fail in an attempt to solve it. For this task they extended their problem solver with a number of reflective modules that performed some simple tests. The representation of knowledge in the reflective modules is procedural rather than declarative though, and the modules work for configuration tasks only. Furthermore, competence assessment was internal to the developed system and no external broker-like agent could perform the competence assessment.

Up to this point, there have been few approaches which use meta-knowledge to represent capabilities and certainly no solution that could be used for capability brokering, as we had hoped.

2.2.3.2 Search Control Knowledge

Although many different uses for meta-level knowledge have been suggested there has been one area where the use of meta-knowledge has had the largest impact: *search control* [Davis, 1980, Bundy and Welham, 1981, Georgeff, 1982]. The idea here is that spending some time on where one is going to search in a large search space is more efficient than just searching. As mentioned above (page 32), experts in a domain often distinguish themselves from novices not by having more relevant domain knowledge, but by using it more efficiently. This suggests that the kind of meta-knowledge that is strongly correlated to capability knowledge as we need to represent it is, in fact, search control knowledge. Thus, we shall now look at search control knowledge to see whether this knowledge can be re-used for capability brokering.

There are a number of *domains* for which search control knowledge has been found and employed. For example, [Bundy *et al.*, 1979] describe a system that

uses meta-level inference to solve mechanics problems; [Wilkins, 1982] uses meta-knowledge to control search in chess; [Minton *et al.*, 1985] have used explicit search control knowledge in parsing; [Murray and Porter, 1989] have used knowledge to control search for consequences of new information during knowledge integration. Planning is of particular interest to us (cf. section 2.3.1) and there are a number of planners that use sophisticated search control techniques[6]. For example, [Tate, 1975] describes in his Ph.D. thesis how the structure of a given goal and its sub-goals can be exploited to control search; [Croft, 1985] examines in his work what exactly the choice points are during planning and develops heuristics to control search at these points; [Fox *et al.*, 1989] view planning as a constraint satisfaction problem and develop the concept of problem texture that is meant to aid in controlling search.

Thus, there exists a large body of search control knowledge that might be re-usable as capability knowledge. However, a closer inspection of the approaches described above reveals that the search control knowledge is often built into the system to maximise efficiency, i.e. it is represented only implicitly. In [Wickler and Pryor, 1996] we have attempted to re-use this implicitly represented search control knowledge to assess competence. However, our aim here is an explicit capability representation and the implicitness of the above search control knowledge is unlikely to provide us with insights as to how to represent capabilities.

2.2.3.3 Learning Search Control Knowledge

What we are looking for at this point are systems that contain explicitly represented search control knowledge that can be re-used for capability brokering. Most systems that *learn search control knowledge* belong to this group. This is because techniques from symbolic machine learning are often aimed at constructing an explicit representation of what they are trying to learn. If this learned search

[6] To avoid confusion here, the term meta-planning has been introduced by [Wilensky, 1981] but does not refer to the use of explicit meta-knowledge to control search in planning.

control knowledge could be re-used for capability brokering it would have the added advantage that we would not even have to find the knowledge ourselves. Thus, we shall now look at systems that learn search control knowledge.

Two general problem-solving architectures have been used to investigate this possibility: SOAR [Laird *et al.*, 1987, Rosenblum *et al.*, 1993] and PRODIGY [Minton *et al.*, 1989, Carbonell *et al.*, 1992, Veloso *et al.*, 1995]. The basic learning algorithms in SOAR are chunking and learning from outside guidance [Golding *et al.*, 1987]. In PRODIGY explanation-based learning has been applied to learn explicit search control rules [Minton and Carbonell, 1987]. Explanation-based learning is a technique that has also recently been applied to learning search control rules for a SNLP-like planner [Kambhampati *et al.*, 1996]. The results described there are rather promising as far as the speed-up over SNLP ([McAllester and Rosenblitt, 1991], cf. section 2.3.1.2) is concerned. Similarly, [Ihrig and Kambhampati, 1997] describe the successful application of explanation-based learning to a case-based planner. Inductive learning of search control rules has been described in [Leckie and Zukerman, 1991], and [Eskey and Zweben, 1990] describe their work on leaning search control knowledge for the closely related scheduling problem. This shows that there is sufficient work in this area to provide a large body of explicit search control knowledge that might be re-usable as capability representations.

However, the fact that explicit search control knowledge can slow down problem-solving has not gone unnoticed [Etzioni and Minton, 1992]. The more search control rules have been learned, the more time it takes to evaluate all of them. Early approaches to this *utility problem* have just counted how often a specific search control rule was fired and deleted it if the success-rate went below a certain threshold [Minton *et al.*, 1987]. Later approaches attempted to approximate the learned search control knowledge to save time [Chase *et al.*, 1989]. [Wefald and Russell, 1989] have even tried to theoretically define when a search control rule has no benefit. [Kambhampati *et al.*, 1996] have avoided the utility

problem by only learning provably correct rules, which are not very many.

As far as capability descriptions are concerned, forgetting or approximating search control knowledge means having a less accurate capability description. Considering the advantages of this approach, i.e. no need for a new representation or the manual development of new knowledge, this inaccuracy seems acceptable. However, there are more worrying results that question the usefulness and thus, the availability of explicit search control knowledge in the long term. Specifically, [Ginsberg, 1996a] has looked at a number of problems to which AI systems have been applied and found that, consistently, the most efficient approaches use relatively uninformed search. Why this is the case is not of much interest to us here, but this problem, which is ultimately rooted in the utility problem, has lead us to abandon the re-use approach argued for in this section.

2.2.4 Terminological KR Languages

By a *terminological knowledge representation language* we mean any formalism for defining and reasoning about concepts in the mould of [Brachman, 1979] and KL-ONE [Brachman and Schmolze, 1985]. Such systems are of little direct relevance here as there has not been a comprehensive attempt to represent capability knowledge in such a formalism. That is not to say that it is not possible though. The reason why we want to mention these languages here is that these formalisms provide the framework for the definition of ontologies to which we will return in section 2.3.2. For example, Ontolingua [Gruber, 1992] can be seen as rooted in terminological KR languages.

A final word concerns the *expressiveness* of terminological KR languages. As expressiveness is a claim we would like to make for the capability description language we will introduce in this thesis, it is worth noting that there has been a formal attempt to define the expressiveness for terminological KR languages in [Baader, 1996]. Baader even claims that this kind of approach is generalisable to other types of KR languages and we shall return to this work in chapter 7.

Most Important Issues Here

- The capability description that we will present in this thesis uses first-order predicate logic to describe states and the situation calculus is an important action representation that could be used in KIF-based brokers (cf. section 9.1).

- Although reusing meta-knowledge initially looked like a very promising approach to capability representations because it potentially allows the re-use of a large existing body of knowledge, recent results related to the utility problem discussed in section 2.2.3.3 indicate that this approach is not desirable. However, capability knowledge is still meta-knowledge and thus, this area is still important.

- Finally, the formalisation of a notion of expressiveness in [Baader, 1996] is relevant to our own claim that the capability description language presented in this thesis is expressive.

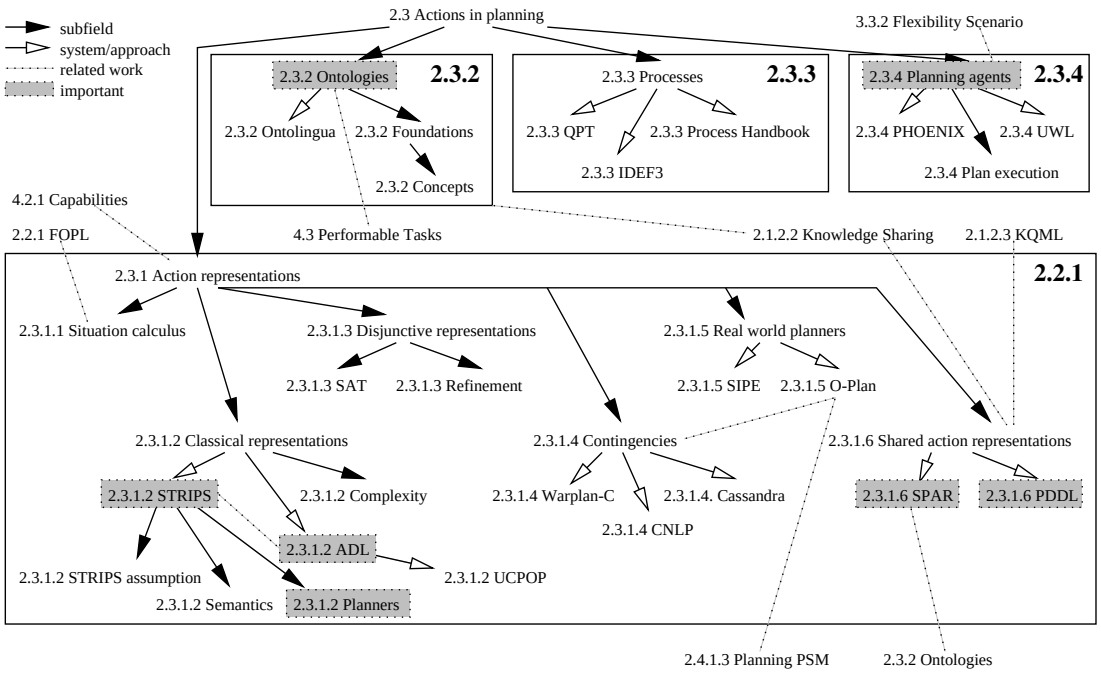

Figure 2.3: Overview of this section

2.3 Actions in AI Planning

In this section we will review approaches to action representations in AI planning, the area we see most closely related to capability modelling.

An overview of this section, which provides a conceptualisation of the relationships between the different sub-fields and approaches/systems described in this section, is given in figure 2.3. The figure also contains cross-links to other areas. The most important areas reviewed here are: classical non-hierarchical action representations, including the STRIPS representation and ADL, on which our capability description language will be based; flexible representations for the communication about actions such as SPAR and PDDL; work on ontologies of actions which provides the basis for the representation of capabilities as performable actions; and work on existing agents based on AI planning technology as our broker will also use a planner.

2.3.1 Action Representation Formalisms

There are *two reasons* why action representations as used by AI planners are of particular importance for our work. Firstly, a primitive action, one of the inputs to the classical planning problem [Tate *et al.*, 1990, page 28], can be interpreted as the representation of a capability (cf. section 4.2.1). The second reason for our interest in action representations and AI planning is more complex. As pointed out before, intelligent agents are often assumed to use a planner to determine a course of action that achieves a given objective [Wooldridge and Jennings, 1995, page 131]. Thus, it is plausible to assume that, for a given objective to be achieved, there exists a planning problem that has a solution if the agent believes it is capable of achieving this objective. This planning problem is characterised by the objective to be achieved, the initial state of the world as perceived by the agent, and the actions the agent believes it can perform. Under this assumption, the problem of capability assessment for an agent may be reduced to the plan existence problem.

2.3.1.1 First-Order Logic and the Situation Calculus

The planning problem was first addressed in AI e.g. in [Green, 1969] and in the situation calculus [McCarthy and Hayes, 1969, Shanahan, 1997].[7] Both of these approaches did not devise a new representation for actions but used *first-order predicate logic* to represent world states, actions, and their effects. Using first-order logic led to a number of problems, most notably, the frame problem [Hayes, 1974, page 69], [Shanahan, 1997]. Although there has been considerable progress towards representations of actions in first-order logic that avoid the frame problem, it can by no means be considered solved. Since we have already discussed first-order logic as a capability representation in section 2.2.1, we shall not go into more detail at this point.

[7] The earliest AI system addressing this problem was probably GPS [Newell and Simon, 1963].

2.3.1.2 Classical Non-Hierarchical Representations

One of the earliest systems in AI to address the planning problem using a task specific representation was the STRIPS system [Fikes and Nilsson, 1971, Fikes *et al.*, 1972]. The STRIPS *representation of actions* basically consists of:

- **an action pattern**: the identifier of the action and some variables describing the parameters;
- **a precondition formula**: a formula that must be true before this action can be applied;
- **an add list**: a list of formulae that will be true as a result of this action; and
- **a delete list**: a list of formulae that will no longer be true as a result of this action.

In the original definition of the STRIPS representation, the different formulae in the representation were allowed to be full first-order logic and a resolution theorem prover was used to reason about world states. However, in a later description [Nilsson, 1980, chapter 7] only conjunctions of literals are permitted, which greatly simplifies the planning process. This later version is what is now generally referred to as the STRIPS representation. The significant advance of this representation over the situation calculus is the STRIPS *assumption*: only what is mentioned in the representation changes when an action is performed, i.e. anything that is not listed in the add or delete list will not change.

One interesting aspect of the STRIPS representation is that there was no *formal semantics* defined for STRIPS for a rather long time. In a classic paper, [Hayes, 1974] pointed out that many representations in AI suffered from this problem, and that formalisms that have no semantics should not be considered representations. Still, it was not until [Lifschitz, 1986] that a semantics for STRIPS

was formally defined. Lifschitz also illustrates how the first intuitive approaches are not always quite the right definitions. It has to be said, though, that the STRIPS representation proved to be an extraordinarily successful action representation. There are still planners being developed today that use exactly this representation.

The STRIPS *planner* on the other hand suffered from many problems that were addressed in a number of subsequent systems, mostly following the STRIPS approach [Georgeff, 1987, Allen *et al.*, 1990, Tate *et al.*, 1990]. The final incarnation of a planner in the mould of STRIPS is probably the partial-order causal-link planner SNLP [McAllester and Rosenblitt, 1991]. However, there has been no significant advance in the representation of actions used by these systems, and this is the aspect we are most interested in here.

One of the more serious limitations of the STRIPS representation is its expressiveness. For example, the situation calculus is considered a much more expressive representation. It was not until [Pednault, 1989] that a serious attempt at exploring the middle ground between these two approaches was made. The result of this work was the new *action description language* (ADL) that combined the expressiveness of the situation calculus with the STRIPS assumption to retain the best of both worlds. The underlying idea in ADL was to exploit the fact that effect axioms in the situation calculus all more or less have the same syntactical format. Pednault used this pattern to define ADL and how ADL expressions should be expanded into situation calculus formulae. In this way, the semantics of ADL was grounded in the situation calculus but the syntax looked much more like STRIPS with precondition, add, and delete formulae.

What Pednault did not do was design a planner for ADL. Although one could translate the representation into first-order logic and reason about it with a theorem prover, this was clearly not the intention. The first planner that was based on a restricted version of ADL was UCPOP [Penberthy and Weld, 1992, Barrett *et al.*, 1995]. The basic extension of UCPOP*'s version of* ADL over the

STRIPS representation was the introduction of conditional effects. Effects are the union of add and delete lists and conditional effects are effects that only occur if certain secondary preconditions hold before the action is executed. Also, conditional effects can occur any number of times with different instantiations for a given action instance. By restricting the domains of all variables to known, finite domains it was possible to extend the basic SNLP algorithm to handle conditional effects.

Complexity of STRIPS Planning As we have mentioned above, one of the reasons why we are interested in AI planning is because the capability assessment problem may be reduced to the plan existence problem. [Bylander, 1994] has shown that the problem of determining whether a given instance of the planning problem has any solution is, even for propositional STRIPS, a PSPACE-complete problem. Thus, assessing capability via plan existence is not a promising route as far as capability retrieval is concerned.

An investigation into whether and why different types of planning algorithms are more efficient than others can be found in [Barrett and Weld, 1994]. In the course of this work, they defined the complexity of a planning problem. We could potentially use such a complexity measure to estimate whether a plan will be found within given resources, i.e. to address the plan existence problem. However, the complexity of the benchmark problems they used was given in terms of the length of the shortest plan solving them, i.e. in general, the complexity was only known once the problem was solved.

2.3.1.3 Disjunctive Representations

In [Kautz and Selman, 1992] a new approach to planning has been suggested. Instead of refining a partial plan through search they have reformulated the planning problem as a *satisfiability problem* to which they applied their stochastic hill climbing algorithm GSAT [Selman *et al.*, 1992]. The difficult task here is the reformulation. [Blum and Furst, 1995] independently found such a reformulation

that led to a significant increase in performance over conventional planners as demonstrated by their planner, Graphplan. This new formulation was later improved and combined with Walksat, an evolution of GSAT, to give even better results [Kautz and Selman, 1996].

Why is it that these satisfiability planners could so drastically outperform all deductive partial-order causal-link approaches? This question has been addressed in [Selman, 1994, Kambhampati, 1997] and they suggest that the essential difference lies in the fact that the representations used by satisfiability planners are capable of representing a *new kind of disjunction in plans*, i.e. sets of plans that contain disjunctions of actions to be included in the final plan. As a response to this finding there are now some deductive planners that also use disjunctive representations, e.g. COPS [Ginsberg, 1996b], UCPOP-D [Kambhampati and Yang, 1996], or Descartes [Joslin and Pollack, 1996]. However, they do not seem to have the performance of satisfiability planners yet.

As far as action representations are concerned, these new planners can be considered a step backwards rather than forward. All the actions the satisfiability planners can reason about are strictly propositional, a limitation that stems from the satisfiability algorithm used. Thus, this work is of little interest to us. The above planners do however reason about disjunction in plans and, as far as plan representations are concerned, this presents a significant advance. This is not an issue here though.

2.3.1.4 Contingencies

An interesting extension of the STRIPS-based action representations presented this far has been introduced in *contingency planning*. Essentially, the idea here is that certain actions may have several alternative outcomes. The first planner to address this problem was Warplan-C [Warren, 1976]. The representation used by Warplan-C was again based on the STRIPS representation. The major difference was that several alternative sets of effects can be specified for an action, each

of which is given a contingency label. Each set of effects was represented as an add and a delete list, as it would be for a normal STRIPS action. The number of contingencies was assumed to be small and not all actions were expected to lead to contingencies. There has also been some work on extending O-Plan (see below) to deal with contingencies [Secker, 1988], but the current version does not contain any such extension.

A more recent contingency planner is CNLP [Peot and Smith, 1992], which is basically a non-linear version of Warplan-C. The underlying action representation did not change from Warplan-C though. A variant of CNLP's algorithm has been used in the Cassandra planner [Pryor and Collins, 1996] which, like UCPOP, also handles conditional effects. In Cassandra's action representation the different contingencies were represented as conditional effects, where the contingency label can be seen as a secondary precondition of the different effects in the different contingencies.

Effectively, contingencies can be seen as introducing disjunctions into effects, thus significantly extending the expressiveness of the representation. Thus, these representations are of great interest to us.

2.3.1.5 Real World Planners

Most of the planners mentioned this far are research vehicles and have not been applied to realistic domains. However, there are at least two planners that have been used outside a research environment: *O-Plan* [Currie and Tate, 1991, Tate *et al.*, 1994, Tate, 1995] and SIPE [Wilkins, 1988]. Both these systems are quite similar in that they support a very rich representation to support planning in realistic domains.

The O-Plan planner essentially consists of: a set of knowledge sources which can address different types of flaws or issues in a plan; a set of constraint managers to evaluate different types of constraints in a plan; and a controller for these modules. The *openness* of O-Plan means new modules can be added

to the planner without too much effort. The representation used by O-Plan is based on the <I-N-OVA> constraint model of activity [Tate, 1996a] which views a plan as a set of constraints on possible behaviour. The actual action representation language used in O-Plan is called O-Plan Task Formalism (TF) [O-Plan TF, 1997, Tate *et al.*, 1998]. O-Plan TF is primarily based on a hierarchical model of action expansion. The representation of primitive actions, the aspect we are most interested in, has a great degree of richness, allowing for a number of constraint types in the representation, e.g. complex temporal constraints, resource constraints, etc. The ability to add new constraint managers as required gives O-Plan the high flexibility needed for realistic domains. Another interesting aspect of the O-Plan planner is that is has been modelled as a CommonKADS problem-solving method [Kingston *et al.*, 1996] (cf. section 2.4.1.3).

When it comes to modelling realistic domains, the richness offered by the representations of these real world planners presents a significant advance over the earlier STRIPS-based representations. However, our aim in this thesis is not to develop a broker for an extremely rich domain which might require the features offered by O-Plan TF or SIPE's representation in its capability representations. What we are aiming for is expressiveness and flexibility and whether more richness necessarily means more expressiveness is an open question. The most interesting aspect of these planners for our work is the openness of O-Plan which gives it its flexibility.

2.3.1.6 Shared Action Representations

Part of the current movement towards knowledge sharing and shared representations (cf. section 2.1.2.2) involves the development of *shared action representations*. In section 2.1.2.3, we have already looked at KQML which can be considered one such language, as a KQML expression represents an action. At least two other efforts with the aim of standardising a common action representation that facilitates knowledge sharing are currently under way. We will look at these next.

One of the latest proposals is the *Shared Planning and Activity Representation* (SPAR) [SPAR, 1997, Tate, 1998]. The principal scope of SPAR is to represent past, present, and possible future activity and the command, planning, and control processes that create and execute plans meant to guide or constrain future activity. It can be used descriptively for past and present activity and prescriptively for possible future activity. The way SPAR aims to facilitate knowledge sharing is not only through a language with an open syntax, but also by providing an ontology of fundamental concepts for representing and reasoning about actions. A brief look at the SPAR ontology will follow in section 2.3.2.

Another shared action representation is the *Planning Domain Definition Language* (PDDL) that was developed as a common format for all competitors in the AIPS'98 planning competition [Ghallab *et al.*, 1998]. The scope of PDDL is far more limited than SPAR: PDDL was only aiming for a representation that covers the representations used by competing planning algorithms. One of the interesting features of this language is that it contains explicit flags for different extensions to the basic language that have to be set in a problem description if the according extension is used. A planner not supporting these extensions can then simply check these flags to see whether it can generate plans for the described problems.

Both representations are only meant as an interlingua and not as representations which are reasoned over directly. Still, both these languages, and KQML, as well, offer very interesting features that our capability representation must also have, such as SPAR's openness and flexibility.

2.3.2 Ontologies of Actions

A logic only defines the syntax and semantics for a representation, but it is the ontology that defines the vocabulary. Approaches to knowledge sharing therefore agree on the need for *shared ontologies* (cf. section 2.1.2.2). Thus, we too will need a shared ontology of actions to represent and reason about capabilities

(cf. section 4.3). One of the best known languages for defining sharable ontologies is Ontolingua [Gruber, 1992], which has been defined as part of the knowledge sharing effort. Methodological issues for developing ontologies are discussed in [Gruber, 1993a, Fernández *et al.*, 1997, Gómez-Pérez, 1998].

Foundations for sharable ontologies of actions are described in [Tate, 1996b]. According to Tate, an ontology can be composed of four major parts. Firstly, there is the meta-ontology which contains fundamental ontological elements used to describe the ontology itself. Secondly, there is the top level ontology which is the minimal ontology used as a framework by all detailed ontologies. Thirdly, there is a library of shared ontological elements which may be shared across a number of detailed ontologies but need not be included. Finally, there are the detailed ontologies which build on the top level ontology and may include ontologies from the library.

A fundamental question is *which concepts* the different parts of a shared ontology of actions should contain. There are a number of such ontologies that have suggested different concepts, mostly for the meta-ontology and the top level ontology. For example, ontologies of actions were defined in the Process Interchange Format (PIF) [Lee *et al.*, 1996, Lee *et al.*, 1998], the Enterprise ontology [Uschold *et al.*, 1996, Uschold *et al.*, 1998], the Core Plan Representation (CPR) [Pease and Carrico, 1997], the Shared Planning and Activity Representation (SPAR) [SPAR, 1997], and recently in work on models of problem solving [Gennari *et al.*, 1998] (cf. section 2.4.2). The SPAR ontology, for example, defines concepts for entities, environments, activities, actions, events, time points, objects, agents, locations, calendars, relationships, activity constraints, world models, plans, processes, objectives, issues, etc. Concepts are related to each other in a semantic network style representation and each concept is defined by a semi-formal description.

Although we believe an ontology of actions to be a considerable aid for the representation of capabilities, the development of such an ontology is beyond the

scope of this thesis. Our capability description language does, however, provide a framework for the representation of and reasoning about ontologies of actions (cf. section 4.3).

2.3.3 Process Modelling

One of the drawbacks of STRIPS-based action representations, as described above, is that they are insufficient for reasoning about processes. This is because they only refer to two states, the one just before the described action and the one just after the action has been completed. Processes cannot be described adequately in this way. A first attempt to reason about simultaneous, interactive processes, characterised by a continuum of gradual change, that may be activated involuntarily, and that take up time, was proposed in [Hendrix, 1973]. This line of work ultimately lead to the *Qualitative Process Theory* (QPT) [Forbus, 1984]. Not only does QPT handle all the above difficulties, it also can be used to come up with useful conclusions even when not all the quantities for a given process are given.

The IDEF3 process capture method has been used to model processes of a different kind [Mayer *et al.*, 1992, Lydiard, 1996]. IDEF3 is part of the IDEF family of methods funded by the US Air Force to provide modelling support for systems engineering and enterprise integration. The IDEF3 method allows different user views of temporal precedence and causality relationships associated with enterprise processes to be captured. The information is presented in a series of diagrams and text, providing both a process-centred view of a system, via the Process Flow Network, and an object-centred view of a system via the Object State Transition Network. This method can tolerate incomplete and inconsistent descriptions and is flexible enough to deal with the incremental nature of the information acquisition process.

[Malone *et al.*, 1997] describe a novel theoretical and empirical approach to tasks such as business process redesign, enterprise modelling, and software development. The project involves collecting examples of how different organisations

perform similar processes, and organising these examples in an on-line *process handbook*. The handbook is intended to help people redesign existing organisational processes, invent new organisational processes, learn about organisations, and automatically generate software to support organisational processes. A key element of the work is an approach to analysing processes at various levels of abstraction, thus capturing both the details of specific processes as well as the "deep structure" of their similarities.

Although the above approaches to process modelling present various interesting ideas, we have chosen not to include a model of processes in our capability representation as the scenarios we envisage do not require such an extension.

2.3.4 Agents Planning with Capabilities

Although it has been argued that deliberative agents should be based on AI *planning* technology [Wooldridge and Jennings, 1995, page 131], most existing agents are not. The earliest agents based on a planner are probably found in [Cohen *et al.*, 1989]'s PHOENIX system which includes planning agents that operate in the domain of situated forest fire management.

We have argued at the beginning of this section (page 40) that the capability assessment problem may be reduced to the plan existence problem under certain assumptions. One of these assumptions was that there will be no problems during the execution of a plan, but we know that this assumption is overly optimistic. Early work that can be seen as the foundation for a planning agent's architecture is presented in [Ambros-Ingerson and Steel, 1988], which describes IPEM, a clear and well-defined framework for the integration of planning, plan *execution*, and execution monitoring. More recent work in the area of plan execution and opportunity recognition with reference features is described in [Pryor, 1996].

Probably the most noteworthy agents that do use a planner are the intelligent softbots developed at the University of Washington [Etzioni *et al.*, 1993, Weld, 1996, Etzioni, 1997]. One of the most interesting aspects of this work for us

is the action representation used by the softbots. They found that STRIPS-based representations lack certain essential features that they needed for their Internet softbots. The action representation language they have developed to model operating system commands, UWL [Etzioni *et al.*, 1992], has two major extensions over conventional languages. Firstly, it allows the modelling of information gathering through goals of the type `(find-out` *literal*`)`. Secondly, one can specify for certain conditions to remain unchanged by an action with a `(hands-off` *condition*`)` goal expression. Arguably, the former is subsumed by reasoning about knowledge as discussed in section 2.1.2 and the latter is a side effect of having to refer to objects by their properties.

Although the agents presented in this thesis will use a planner to reason about their actions, our main concern is with the capability reasoning performed by the broker, prior to the assignment of tasks to problem-solving agents. Hence, problems occurring during plan execution are not addressed in this thesis.

Most Important Issues Here

- Primitive actions in classical non-hierarchical action representations (section 2.3.1.2) like STRIPS and ADL form the basis for the capability description language presented in this thesis (cf. chapter 4).
- Furthermore, our capability description will be open like the O-Plan representation giving it flexibility (cf. chapter 8), it will allow for the representations of ontologies of actions like the SPAR ontology (cf. section 4.3), and it will allow for the flagging of language properties similar to PDDL (cf. section 4.4).
- A planner is used by the PSAs to determine a course of action, but the resulting issues concerning plan execution are not addressed in this thesis. A planner is also used by the broker to combine capabilities of various PSAs to solve a given problem (cf. section 3.3.2).

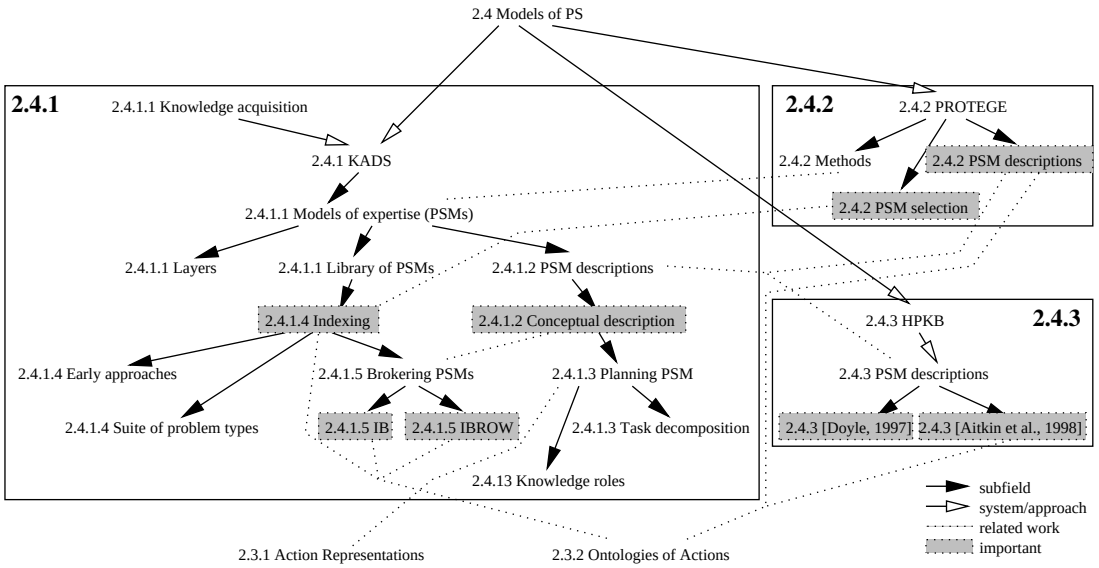

Figure 2.4: Overview of this section

2.4 Models of Problem Solving

In this section we will review approaches to modelling problem-solving methods.

An overview of this section, which provides a conceptualisation of the relationships between the different sub-fields and approaches/systems described in this section, is given in figure 2.4. The most important areas reviewed here are the various representations used to describe problem-solving methods which can be seen as reasoning capabilities and the approaches to the indexing problem which resembles the capability retrieval problem. Furthermore, we shall return to the brokers for problem-solving methods in section 9.1.

2.4.1 KADS-Based Approaches

We were interested in models of problem solving to investigate whether these models can be seen as capability models and thus, can be used for capability brokering. One of the largest and longest-running projects that deals with the modelling of problem-solving methods is the KADS *project* [Wielinga and Breuker, 1986,

Breuker and Wielinga, 1989, Wielinga *et al.*, 1992].[8] Therefore, we shall now look at the KADS representation of models of problem-solving.

2.4.1.1 The KADS Methodology

The KADS methodology is a tool for *knowledge acquisition* and the building of knowledge-based systems (KBSs). Knowledge acquisition is a constructive process in which the knowledge engineer uses data about the behaviour of an expert to make design decisions regarding a KBS to be built. In this view, a KBS is an operational model that is the result of knowledge acquisition. The process of knowledge acquisition consists of knowledge elicitation, knowledge interpretation, and formalisation.

The KADS methodology's first principle is that the knowledge acquisition process should result in a number of *intermediate models*. These are: the organisational model, the application model, the task model, the model of cooperation, the model of expertise, the conceptual model, and the design model. The organisational model and the application model are models of the environment the KBS is meant to be used in and the problem it is meant to address. The task model specifies how the function of the system is to be achieved and contains the task decomposition. The model of cooperation assigns tasks and sub-tasks to agents. The model of expertise specifies the problem-solving expertise required to perform the problem-solving tasks assigned to the system at the knowledge-level [Newell, 1982]. The conceptual model is an abstract description of the objects and operations the KBS should know about. Finally, the design model is a high-level specification of the KBS, the operationalisation of which should be the KBS itself.

The model that we are most interested in here, as it appears to be the closest to a capability model, and that has received the most attention within the KADS community is the *model of expertise*. KADS suggests a decomposition of this

[8] The title of the project was changed to *Common*KADS in later years.

model according to the types of knowledge it contains into several layers:

- **The domain layer**: The domain knowledge embodies the conceptualisation of the domain for a particular application in the form of the domain theory. It contains concepts, properties, and relations between concepts and their properties.

- **The inference layer**: The inference knowledge embodies primitive inference actions over the domain knowledge, also referred to as the knowledge sources. Domain knowledge is mapped into the meta-classes or knowledge roles that represent the generic input and output of the inference actions by the domain view. The inference structure describes the flow of knowledge between the inference actions similar to a data flow diagram.

- **The task layer**: The task knowledge embodies the control knowledge needed to perform reasoning at the inference layer. This includes knowledge of how the overall task is to be decomposed into subtasks.

- **The strategic layer**: The strategic knowledge determines what goals are relevant to solve a particular problem. However, this layer was dropped in KADS-II/CommonKADS.

One of the key issues in the KADS methodology is that it strongly advocates the re-use of knowledge, which is to be achieved through a *library* of such knowledge. The library is divided into two parts: the domain division, which is concerned with generic and re-usable domain knowledge, and the task division, which contains the description of the *interpretation models* or *models of problem-solving*. An interpretation model is a model of expertise with an empty domain layer, i.e. it is a domain-independent description of a problem-solving method (PSM) [Benjamins *et al.*, 1997]. These are exactly the models we are interested in.

2.4.1.2 Descriptions of PSMs

The KADS library contains a number of generic PSMs that represent the experience gained in many years of knowledge engineering [Breuker *et al.*, 1987, Breuker and Van de Velde, 1994]. The *description* of each PSM in the library consists of three parts: a verbal description, a conceptual description, and a formal description. The verbal description is a description in natural language. The conceptual description uses a frame-like language derived from the Conceptual Modelling Language CML [Wielinga (ed) *et al.*, 1994, chapter 3]. The formal description which exists only for a few PSMs is given in ML^2 (see below).

For each PSM the *conceptual description* defines functions which are essentially the primitive inference actions, function structures which are more or less the inference structures, and a control structure which is the control regime applied in this PSM. The conceptual description of a function consists of a description of the dynamic input and output knowledge roles of this function, the static knowledge roles it accesses, its goal, a specification of the relation between input and output, and an operation type which is the type of primitive inference this function performs. The function structure is a collection of the functions this structure is composed of. The control structure is a specification of the control flow over these functions including how the overall task accomplished by this PSM is to be decomposed.

Although the conceptual description contains the right kind of knowledge to be considered a capability representation, it also allows for informal content in many places. Thus, it can not be used as is for automated brokering, but it provided us with insights for designing our own capability representation.

Formal Specifications of Models of Expertise ML^2 is a formal specification language based on KADS models of expertise [van Harmelen and Balder, 1992, Aben, 1995]. It allows different levels of formalism for domain, inference, and task layer. The domain layer is to be specified essentially in typed first-order logic.

The inference layer extends this by allowing the reification of expressions, i.e. a form of meta-expressions, and reflective reasoning about these named expressions. Finally, the specification of the task layer is to be defined in Quantified Dynamic Logic (cf. section 2.2.2). While this formalism is certainly powerful, it has been "claimed that highly trained mathematicians are needed to write, to understand and to verify a formal specification" [Aben, 1995, page 20].

2.4.1.3 Planning as a PSM

As we have pointed out in section 2.3.1, *planning* is one area of particular interest to us because the ability to generate a plan to solve a given problem can be interpreted as the capability of solving this problem. Thus, we will have a brief look at the PSM for planning described in the CommonKADS library of PSMs now [Valente, 1994, Valente, 1995, Barros *et al.*, 1996].

The first step in the description of a PSM is the identification of the *knowledge roles*. For the planning task, four dynamic and two static roles have been identified. The dynamic knowledge roles are the current state, the goal, the plan, and the conflicts. The current state is a description of the initial state of the world, and the goal is a set of conditions to be achieved in a future state of the world. The plan consists of a set of plan steps, ordering constraints, variable bindings, and causal links (cf. section 2.3.1). The conflicts represent the discrepancy between the conditions in the goal and what the plan achieves. The static knowledge roles are the world description and the plan description. The world description consists of the state description, e.g. fluents in the situation calculus (cf. section 2.2.1), and the state changes, effectively the possible actions in the domain. The plan description comprehends the optional plan structure, a hierarchical decomposition of the actions, and the plan assessment knowledge used to evaluate plans.

The *task decomposition* for the planning PSM is summarised in figure 2.5. Tasks are represented by ellipses in this figure and PSMs are represented by boxes.

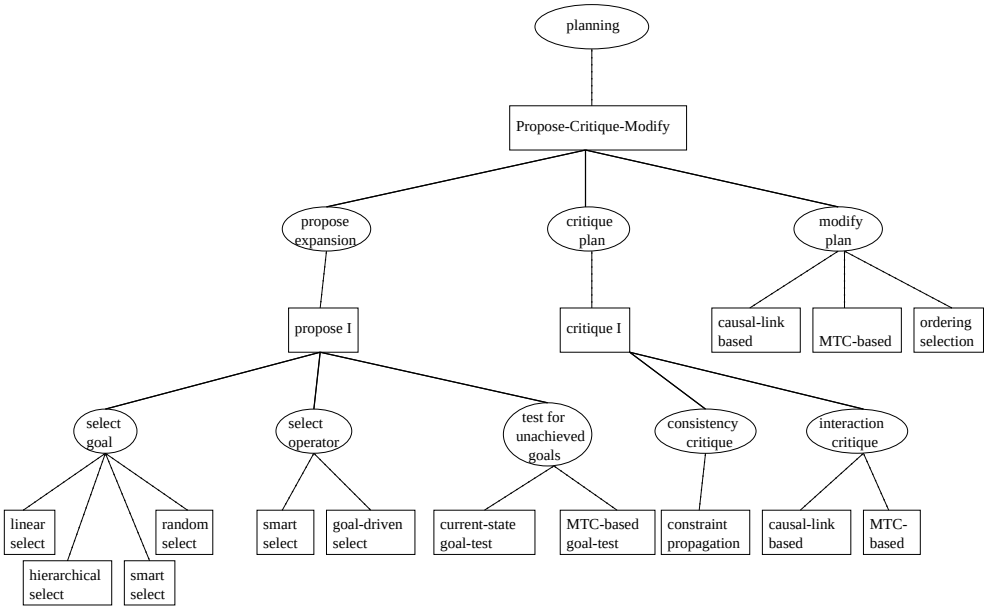

Figure 2.5: Task-Method Decomposition; from [Barros *et al.*, 1996, page 15]

For example, the planning task can be addressed with a propose-critique-modify PSM which leads to three sub-tasks: propose expansion, critique plan, and modify plan that have to be performed in this order. Each of these tasks can again be addressed by some PSM until no further decomposition is possible.

It has been shown that this PSM description does describe, at the knowledge level, the problem-solving behaviour of many modern planners. However, it is difficult to see how this description can be used to decide whether a planner will be able to find a solution for a given planning problem, i.e. how this description can be used to assess capability.

2.4.1.4 Indexing PSMs in the Library

During the knowledge acquisition process the knowledge engineer might identify a PSM from the *Common*KADS library [Breuker and Van de Velde, 1994] as appropriate for the task at hand and then use the respective model from the library to focus the knowledge acquisition process, e.g. by attempting to elicit domain knowledge to fill relevant knowledge roles and by defining the domain view [Brazier *et al.*, 1995]. Thus, the retrieval of an appropriate model from the lib-

rary, also referred to as the *indexing problem*, is an important step in the KADS methodology, just as it is for capability brokering. However, it is also expected that the model from the library will need further refinement before it can be transformed into the design model. This refinement process is called knowledge differentiation in KADS. It is worth noting at this point that the resulting model, the design model, is not meant to be operational in KADS.

One approach to the indexing problem in KADS was the definition of a *taxonomy of generic tasks* which is supposed to help the knowledge engineer to identify an appropriate PSM in the library [Breuker and Van de Velde, 1994, page 59]. The idea was for the knowledge engineer to follow one path down a hierarchy that ends in the most specific PSM suitable for the task at hand. However, in practise this turned out not to be so simple. Another approach tried in the KADS project was the indexing of PSMs in the library with *task features* [Aamodt *et al.*, 1993] for which they have suggested a quite elaborate list of such features. However, this approach also proved insufficient for certain types of problems.

The latest insight seems to be to take a more indirect approach. The basic argument in [Breuker, 1997] is that one is given a problem and different kinds of PSMs might be able to solve this problem. The PSM selection mechanism should reflect this by providing a *suite of problem types* and associating a number of PSMs with each problem type. The selection amongst these could then be by assumptions made by the PSM, by the domain, or by the depth with which the PSM has been modelled. Another, recent criticism of the original indexing mechanism is that it is based only on yes/no distinctions and does not allow gradual refinement [van Harmelen and ten Teije, 1998].

Since we can not use the KADS representation for PSMs to represent capabilities, we also can not use their indexing. However, we have tried to take into account the lessons learned from their work, specifically, the approach to indexing PSMs by the problems they solve.

2.4.1.5 Brokering for PSMs

There are currently at least two approaches in progress that attempt to address the indexing problem with a *broker*. Naturally, we are interested in this work as the problem addressed is very similar to our problem.

IB, the *Intelligent Broker* [Fensel, 1997, Decker *et al.*, 1998], currently under research at the University of Karlsruhe is one such broker. The aim of this broker is not to facilitate agent cooperation, as it is for the brokers described in section 2.1.3, but to find a PSM for a given task on the Internet. Unlike most other brokers reviewed in this chapter, IB is not based on KQML. The approach assumes the availability of an ontology of PSMs which is used for the description of PSMs and which the broker can use for its search. The ontology of PSMs they envisage does not yet exist but might well be based on the taxonomy of PSMs described in the KADS library of expertise models [Breuker and Van de Velde, 1994, page 59]. The language in which they intend to describe PSMs and on which their ontology will be based is not finished yet. This language will be called the Unified Problem-solving Method description Language (UPML), but only a draft specification exists [Fensel *et al.*, 1998a, Fensel *et al.*, 1998b]. Another task envisaged for this broker is the adaption of the selected PSM to the actual task which requires mapping entities in the given problem to the roles of the PSM.

Another project that is closely related to the work on IB is the ESPRIT-funded project IBROW^3 that started in January 1998 [Benjamins *et al.*, 1998, Armengol *et al.*, 1998]. The aim here, too, is to develop a broker that can select, configure, and adapt knowledge components from large libraries on the Internet. For selecting a problem-solving method from a library, the broker will reason about characteristics of the PSM, in particular about their competence and requirements. For this purpose PSMs will have to be described in some language. Although no such language has been selected or proposed yet, it is envisaged that an ontology will be at the heart of the approach. Furthermore, as the people involved with the IBROW^3 project are largely the same as for the IB framework

it is quite possible that the two systems and PSM description languages will be very similar.

2.4.2 PROTÉGÉ

The PROTÉGÉ system [Musen, 1989, Eriksson *et al.*, 1995] addresses a problem very similar to the problem addressed in KADS and thus, we are interested in PROTÉGÉ for very similar reasons.

PROTÉGÉ provides a knowledge engineering environment in which a developer can specify tasks and select PSMs from a library of re-usable *methods*. Developers must identify, at least partially, the task of the system they are designing before they can select and custom tailor preexisting methods. This task-analysis leads to a system-role description in terms of the domain for the system, which serves as the basis for the selection of PSMs that accomplish the task and for the configuration of the selected methods for the task instance. In PROTÉGÉ, methods are actions that accomplish tasks. Methods can delegate problems as subtasks to be solved by other methods. They use the term "mechanism" for primitive methods that cannot be decomposed. In addition to supporting the development of problem solvers for knowledge-based systems, PROTÉGÉ generates domain-specific knowledge acquisition tools that elicit the expertise required by the PSMs to perform the latter's task.

For PSM *selection*, they believe that it will be difficult to make a comprehensive list of factors to consider. However, they do identify a set of recurring factors that are applicable to most tasks. This list of common factors includes the input and output of the task, the domain knowledge available, the solution quality required, the computational time and space complexity, and the flexibility of the method. Once a method has been selected it needs to be configured. This is largely a matter of selecting mechanisms or methods for a method's subtasks and defining the mapping between method terms and domain terms.

An essential part of the *method description language* developed in PROTÉGÉ

is the *method ontology* which includes definitions of all the objects required by the PSM. Ideally, developers of PSMs would share a framework for defining inputs and outputs. [Gennari *et al.*, 1998] have begun to develop a "foundation ontology" for developers of PSMs. In this ontology, a PSM must have a name and a textual description. Furthermore, it contains ontology frames for input and output, a list of subtasks, and a list of constraints across the inputs and output but not among inputs or outputs. The latter are located inside the ontology frame for inputs and outputs, together with key classes and functions in this frame, and the API used which contains information about the ways in which the PSM makes run-time queries for additional information. Subtasks again come with a textual description, inputs, outputs, constraints between those, and information as to whether this subtask is required and whether it has a default implementation. The lowest level of detail in their method description language is the choice of a formal language for expressing the axioms that represent the requirements of the method. The current suggestion is that this language will be based on KIF [Genesereth, 1991, Genesereth *et al.*, 1992].

To summarise, not only are the problems addressed by KADS and PROTÉGÉ very similar, but so are the approaches. Methods in PROTÉGÉ correspond to models of expertise in KADS. Both approaches are based on a library of PSMs and the description languages they use are essentially informal. Furthermore, both approaches address the indexing problem and suggest that an ontology will be the key to the solution. Thus, virtually all of our comments on KADS also apply to PROTÉGÉ.

2.4.3 The HPKB Program

The DARPA-funded *High Performance Knowledge Bases (HPKB) program* is a research programme to advance the technology of how computers acquire, represent and manipulate knowledge [Cohen *et al.*, 1998].[9] The approach taken in

[9] cf. `http://www.teknowledge.com/HPKB/`

HPKB is quite similar to the approach in KADS again. One of its aims is to speed up the development of knowledge-based systems significantly. One way to achieve such a goal is through the enablement of knowledge reuse, including the reuse of PSMs. This might ultimately lead to the fully automated configuration of knowledge-based systems. For this purpose they are interested in developing a language for describing PSMs and a number of groups are currently working on a proposal for such a language. For example, the latest work on PROTÉGÉ is one of the inputs to the HPKB effort.

[Doyle, 1997]'s *proposal* for a PSM description language was one of the earliest contributions for the HPKB program. According to his proposal, a capability description should include: the task addressed by the method; the method ontology; the contextual properties; the behavioural properties; the cognitive properties; relationships to other methods; relationships to implementations; and other annotations. However, as this proposal was still an early draft we shall not go into detail here.

Another interesting input to this part of the HPKB program is the *language proposal* described in [Aitken *et al.*, 1998]. They suggest that a PSM can be viewed as a process or action. In this case process or action representations from AI planning might also work for PSMs. We have reviewed process modelling techniques in section 2.3.3 and we have looked at action representations in section 2.3.1. As a result of this view, the language they propose characterises a PSM or capability in three parts. Firstly, there is the competence of the capability. This includes the goal or objective, the problem type the PSM addresses, a generic solution, the solution components (conclusion, argument structure, and case model), solution properties, and the rationale which can be a textual description of when and why the PSM might be used. Secondly, there is the configuration of the capability. This includes the method ontology, the domain theory consisting of field, ontology/mapping, and representation, and the sub-methods. The third and last part is the PSM process which includes the environment, the resource constraints, the

actor constraints, various world constraints, and sub-activities.

Compared to KADS or PROTÉGÉ, HPKB is still in its infancy. The language proposals are all draft and indicate types of knowledge to be represented rather than defining actual languages. Thus, we can only take these initial ideas into account when designing our own capability description language.

Most Important Issues Here

- Knowledge engineering with models of problem solving is often based on a library of PSMs. This library contains at least semi-formal descriptions of PSMs and it is the description languages suggested by the different approaches we are most interested in.

- Especially KADS and KADS-related work has been concerned with the indexing problem for their library. The indexing problem is closely related to the capability retrieval problem and thus, we must learn from their work.

Summary

In this chapter we reviewed various approaches to *representing and reasoning about capabilities*. In section 2.1 we looked at software agents, the area in which most of the work on brokering has taken place to date. This area has been mostly concerned with the reasoning aspect of capability brokering. This work introduced us to the agent communication language KQML that all agents developed for this thesis will use. The scenarios presented in the following chapter will illustrate our use of KQML.

The remaining sections in this chapter described approaches which were mostly related to the representation aspect of capability brokering. In section 2.2 we looked at the way various logics could have been used to represent capability knowledge. In section 2.3 we looked at how representations of actions, which are very similar to capabilities, have been encoded in AI systems. Finally, in section 2.4 we looked at models of problem solving methods to see whether these models represent capability information, and if so, how it was represented.

In chapter 4 we shall describe our capability description language. In section 4.1 we shall identify several characteristics which we want our capability description language to have. We will then evaluate the approaches we reviewed in this chapter with respect to these characteristics before proceeding with the definition of our own language.

Chapter 3

Scenarios, Agents, and Messages

At this point the general problem of capability brokering has been described and previous approaches to representing generic capabilities have been discussed. Our aim is to define a new capability description language that will be expressive and highly flexible and can be used to reason about capabilities. The next step towards this goal will be to define several scenarios that illustrate the expected behaviour of the different agents involved. The most interesting of these scenarios will be presented in section 3.3. The contribution of this chapter will be a clear definition of the expected problem-solving behaviour and a characterisation of the knowledge that needs to be represented in the message exchanges described.

3.1 The Initial Scenario

In this section we will present an example domain and a simple, initial scenario involving several agents that play different roles. This domain and scenario will reoccur throughout the thesis.

3.1.1 The Domain: Pacifica

Before we describe our example domain it is worthwhile saying what we mean by *domain* and *scenario*. By a *scenario* we mean a reasonably short outline of

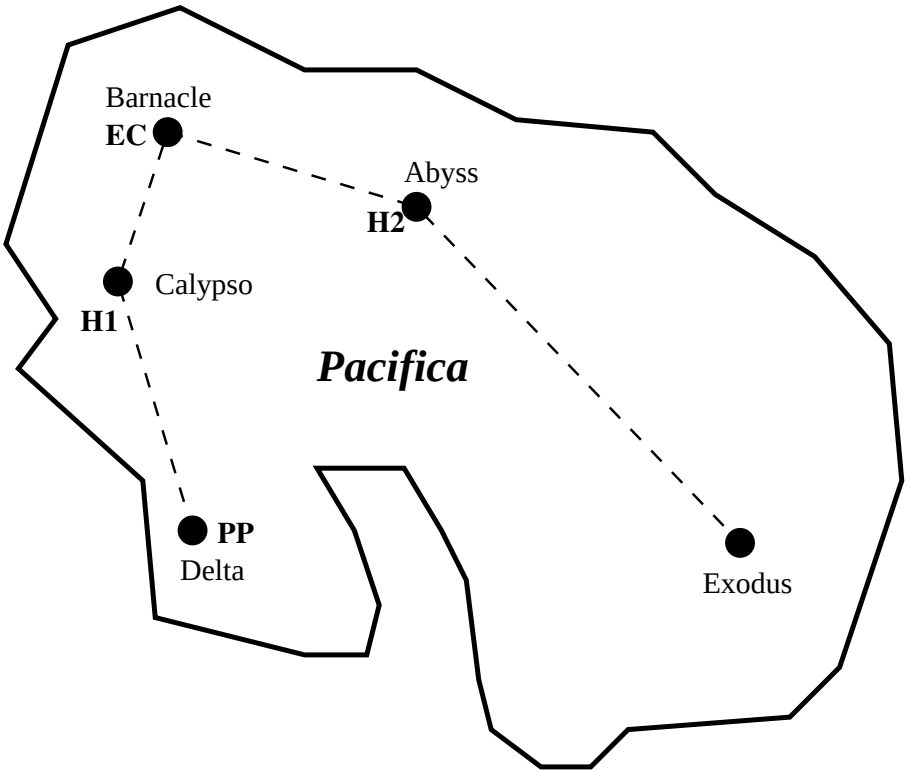

Figure 3.1: A Map of Pacifica

an episode in the life of several agents. The environment in which these agents exist is what we call the *domain*. In other words, a scenario is a kind of informal script and its domain are the surrounding conditions.

Almost all the scenarios presented in this thesis take place on an island called *Pacifica*[1] which constitutes the domain for our scenarios. Figure 3.1 gives a basic map of Pacifica. The initial agents that exist on Pacifica (represented by **EC**, **H1**, **H2**, and **PP**) will be described in section 3.1.2 below.

There are five cities on Pacifica which are called Abyss, Barnacle, Calypso, Delta, and Exodus for simplicity. All five cities are connected by one major road which goes through all the cities and makes up the infrastructural backbone of the island. According to this road Barnacle can be seen as central and Delta and Exodus as the remote extremes on the island.

[1] The idea to use Pacifica as a sample domain has been inspired by the PRECiS Environment [Reece *et al.*, 1994] where this imaginary island state proved to be a very illustrative domain.

3.1.2 Agents on Pacifica

3.1.2.1 The Problem-Solving Agents

We will first describe those agents on Pacifica that play the role of problem-solving agents (PSAs) in the initial scenario, i.e. agents that provide the general capabilities that may be used by other agents to solve their problems (cf. section 1.1.2)[2]. These agents are all *real agents* in the real world. In our scenarios every real agent will have an equivalent software agent that represents the real agent and acts towards other software agents as if it had the capabilities of the real agent. This mechanism simplifies the integration of real agents into an electronically brokered world.

Now, there are three PSAs on the island based in different cities (cf. figure 3.1). These are:

1. an engineering company represented by the **ec**-agent,
2. a hospital represented by the **h1**-agent, and
3. a second hospital represented by the **h2**-agent.

The first PSA is a hypothetical *engineering company* that is based in Barnacle and marked as **EC** in the map. The engineering company employs two engineers that have a truck available that they can use to drive to the other places on the island. Once at a given location with their truck they can repair any type of machine.

The software agent representing the engineering company, the **ec**-*agent*, knows the basic map of the island described above. It also knows its own capabilities and that the other agents on Pacifica exist. However, it does not know what the capabilities of the other agents are. The **ec**-agent uses a planner to plan the actions necessary to complete a given task. Figure 3.2 describes the actions

[2] Further PSAs shall be introduced for the more complex scenarios as we need them.

```
schema drive_to;
;;; go to the place where something is to be repaired:
  vars ?place = ?{type Place}, ?place2 = ?{type Place};
  expands {drive_to ?place};
  conditions
    only_use_for_query {Has Location ECTruck ?place2} = true;
    ;;; add reachability condition here
  effects
    {Has Location ECTruck ?place2} = false,
    {Has Location ECTruck ?place} = true;
end_schema;

schema repair;
;;; repair a machine:
  vars ?machine = ?{type Machine}, ?place = ?{type Place};
  expands {repair ?machine};
  conditions
    only_use_for_query {Has Location ?machine ?place} = true,
    achieve {Has Location ECTruck ?place} = true;
  effects
    {Is ?machine Broken} = false;
end_schema;
```

Figure 3.2: Actions available to the **ec**-agent

available to the planner used by the **ec**-agent in O-Plan-TF[3] [O-Plan TF, 1997]. Note that this description indirectly describes the capabilities of the **ec**-agent.

The next PSA to be described here is the *first hospital* which is based in Calypso and marked as **H1** in the map. This hospital employs several doctors and the usual support staff. The hospital also has an ambulance that can be used to drive to the other places on Pacifica to fetch injured people. Only once the injured people are at the hospital can their injuries be treated.

The software agent representing this PSA, the **h1**-*agent*, knows the basic map of Pacifica, its own capabilities, and that the other agents exist. It is not aware of their capabilities though. Like the **ec**-agent the **h1**-agent uses a planner to plan

[3] There are alternative ways of representing these actions and our representation is not meant to be the most efficient.

```
schema fetch_patient;
;;; drive an ambulance to wherever the patient is, load the
;;; patient, and return to h1:
  vars ?patient = ?{type Person}, ?place = ?{type Place};
  expands {fetch ?patient};
  conditions
    only_use_for_query {Has Location ?patient ?place} = true;
    ;;; add reachability condition here
  effects
    {Has Location ?patient ?place} = false,
    {Has Location ?patient H1} = true;
end_schema;

schema treat_patient;
;;; treat a patient that is at the hospital:
  vars ?patient = ?{type Person};
  expands {treat ?patient};
  conditions
    achieve {Has Location ?patient H1} = true;
  effects
    {Is ?patient Injured} = false;
end_schema;
```

Figure 3.3: Actions available to the **h1**-agent

its actions and figure 3.3 describes the actions available to it in O-Plan-TF.

The final PSA described here, the *second hospital* represented by the **h2**-*agent* is almost identical to the first hospital except for that it is based in Abyss. It has knowledge equivalent to the **h1**-agent's knowledge and the capabilities of these two agents are virtually the same. Thus, we will omit the description of the actions available to its planner here.

3.1.2.2 The Broker

As we have mentioned in the descriptions of the PSAs above, none of these agents actually knows the capabilities of the other PSAs. The *broker* is the agent that has the knowledge about the capabilities of the different PSAs and, on request, it can find a PSA that can solve a given problem. This is all the broker needs to do

in our scenarios.

The broker can be seen as a PSA, too, but we prefer to use the term PSA only for agents that solve problems at the domain level like the agents described above. The broker solves a problem at the meta-level, the problem of capability brokering.

3.1.2.3 The Problem-Holding Agent

The final agent for the initial scenario is the *power plant* on Pacifica. The power plant is located in Delta and marked as **PP** on the map. The power plant supplies the island with electricity. It has a number of generators that generate the electricity and employs a few people that look after the generators during normal operation.

The power plant is represented by the **pp**-agent. It might be a PSA in other scenarios but for the initial scenario discussed here it is the problem-holding agent (PHA), i.e. the agent that has a problem it wants solved by utilising the capabilities of the PSAs (cf. section 1.1.2). Thus, there is no need to describe the capabilities of the **pp**-agent here in detail.

3.1.3 Script for the Initial Scenario

Now, suppose that there has been an accident at the power plant in which a gasket on one of the generators broke and let steam escape. Unfortunately an employee of the plant had been near the generator while this happened and suffered some burns. So there are two problems to be dealt with here:

- a person has suffered burns and needs treatment; and
- a generator is now malfunctioning and needs to be repaired.

The script for the *initial scenario* is as follows:

Example 3.1 (Initial Scenario) *Like the* PSA*s, the* **pp***-agent does not know the capabilities of the other agents on the island. However, it does know that other agents exist and specifically, that the broker can find* PSA*s that can deal with a given problem. Thus, the first thing that has to happen is that the* **pp***-agent contacts the broker to ask for agents that can deal with its problems. Note that this assumes that the broker somehow already knows the capabilities of the different* PSA*s.*

Now the broker has to look through its data base of agents and capabilities to find PSA*s with sufficient capabilities to solve the problems described by the* PHA*. If the broker manages to find such* PSA*s it has to inform the* PHA *about these agents. In the initial scenario the broker will find that the* **ec***-agent can repair the generator and that the* **h1***-agent can deal with the injured person.*[4]

With the knowledge of which PSA*s can solve the* PHA*'s problems the* **pp***-agent can now contact the* **ec***-agent and the* **h1***-agent and ask them to actually solve the problems.*

Finally, the **ec***-agent and the* **h1***-agent can go ahead and solve the given problems. For this initial scenario we shall assume that there were no further complications and all the* PSA*s have to do after completing their tasks is to report success back to the* PHA*.*

As mentioned above, for the scenarios we will describe in this thesis the power plant shall be our PHA. The problem that this agent has will be largely the same for all the scenarios that are going to be presented. The main differences between the different scenarios will be in the capability descriptions for the different agents.

[4] Alternatively, the broker could recommend the **h2**-agent to deal with the injured person, but only one agent is required to address this part of the problem.

3.2 Inter-Agent Messages

In this section we will show what the messages look like that the different agents will need to exchange. These messages are expressed in a high level communication language. We have chosen KQML *for this purpose. This section will also show what knowledge the messages will need to contain, i.e. what needs to be represented in capability descriptions.*

As pointed out in section 1.1.1, social ability is one of the key features of an intelligent agent. Hence, we will now have a closer look at the different messages the agents described in section 3.1.2 will need to exchange to achieve the behaviour described in the initial scenario.

We feel that the communication between software agents has to be in some formal language and we have chosen KQML as the high-level agent communication language (cf. 2.1.2.3). This is mainly because: KQML is one of the best understood languages for this purpose; it is very general by allowing arbitrary content languages; and there is software available that embeds KQML into a number of environments.

3.2.1 Capability Advertisement Messages

The starting point for the message exchange is a situation in which all agents are on-line, i.e. ready to communicate with each other, but they do not know about most of the other agents. Specifically, they do know about the broker, but they do not know about the problem-solving capabilities of the PSAs. Hence, as a first step the PSAs have to tell the broker about their capabilities. Here are the messages we would expect the PSAs to send to the broker advertising their capabilities:

```
(advertise
  :sender   ec
  :receiver ANS
  :ontology capabilities
  :language KQML
  :content  (achieve
              :receiver ec
              :ontology OPlan
              :language CDL
              :content  (<machine fully functional>)))

(advertise
  :sender   h1
  :receiver ANS
  :ontology capabilities
  :language KQML
  :content  (achieve
              :receiver h1
              :ontology OPlan
              :language CDL
              :content  (<injured people treated>)))

(advertise
  :sender   h2
  :receiver ANS
  :ontology capabilities
  :language KQML
  :content  (achieve
              :receiver h2
              :ontology OPlan
              :language CDL
              :content  (<injured people treated>)))
```

Message contents describing capabilities are informal here and are only meant to illustrate what kind of knowledge needs to be represented in our capability description language. The complete messages will be described in section 4.2.5 together with our capability description language CDL.

Let us first look at the *outer part* of these KQML messages. The performative of all these messages must be `advertise`. With a message with this performative the sender tells the receiver that it is capable of processing messages of a certain type. The sender in each case must be the respective agent that is advertising the capability with this message. The receiver must be the Agent Name Server

(ANS). This is because we envisage the broker as an extension of an ANS that supplies agent addresses not only by name but also by capability.

For the agents to share knowledge about capabilities they must have at least one shared ontology which has to be named in the KQML message.[5] The ontology in our examples in called `capabilities`. The content language is again KQML, i.e. there is a KQML message inside a KQML message. This *inner* KQML *message* is given in the content field.

The specification for KQML prescribes that the content field in an advertisement message should contain the KQML message the advertiser commits to processing with this advertisement, i.e. the content of the advertisement and any future message to be processed must be identical [Labrou and Finin, 1997, page 19]. This approach is extremely limited and most existing brokers have extended it to allow the content of the advertisement to be a generalisation of the actual messages the PSA commits to processing.

The performative of the inner message is `achieve` in all three cases.[6] With messages of this type the sender asks the receiver to make something true in its physical environment. Thus, the capability of making a given condition true is what is being advertised here. There is no sender specified in the inner message as this will be the agent requesting the achievement at some later stage. The receiver must be the advertising agent, i.e. the sender of the outer message. The ontology specified in the inner messages is `OPlan` for all of our PSAs. This is only because all of our PSAs use the O-Plan planner to plan their actions. In general any ontology the PSA knows about can be specified here.

We have chosen to allow for a new capability description language that appears in the content of the inner message of the capability advertisement. This language is called CDL and will be described in chapter 4 in detail. At this point we are

[5] The Java Agent Template which is the basis for the implementation of our agents makes a rather unusual use of the ontology field in a KQML message. The details will be explained in section 5.3.

[6] An extension that allows a second performative "`perform`" in the inner message will be described in section 4.3.

only interested in illustrating how this language fits into KQML and how it can be used to make the initial scenario work. Thus, the message contents describing capabilities are only given informally here.

At the heart of this content must be a description of the condition the PSA can make true. For example, the content of the inner message of the capability advertisement of the **ec**-agent must say that it can achieve the condition that a machine is fully functional. It will also often be necessary to qualify such an achievable condition with an applicability condition and we will allow this in CDL. For example, the engineering company can only achieve fully functional states for machines. Achievable conditions amended with applicability conditions will be the core of capability descriptions in CDL.

Another important issue arises from a comment made in section 3.1.2, namely that operators available to a PSA's planner already provide an indirect capability description for this agent (cf. section 2.3.1). However, this might not describe the capabilities the agent wants to advertise. For example, a hospital with an ambulance surely can drive a person from one place to another besides back to the hospital but will normally not want to provide this capability to other agents (i.e. a hospital is capable of providing a taxi service but will not actually offer to do so). Thus, capability descriptions will not necessarily describe actual capabilities, only selected advertised capabilities.

3.2.2 Messages for the Initial Scenario

After receiving the above messages the broker will be aware of these three agents' capabilities. This is the starting point for the initial scenario. The next step is the power plant informing the broker of the problem at hand and asking for agents that can deal with this problem. KQML has a number of performatives that allow for this kind of message. We shall only look at one of those here: `recommend-one`. With this type of message an agent asks the broker to find exactly one agent that can deal with the problem described in the content of this message. For our

example 3.1 there will be two messages describing the two parts of the problem:

```
(recommend-one
  :sender   pp
  :receiver ANS
  :ontology capabilities
  :language KQML
  :content  (achieve
              :sender   pp
              :language CDL
              :content  (<generator fully functional>))

(recommend-one
  :sender   pp
  :receiver ANS
  :ontology capabilities
  :language KQML
  :content  (achieve
              :sender   pp
              :language CDL
              :content  (<injured person treated>))
```

The performative of the outer message is `recommend-one` and the sender is the **pp**-agent, the PHA in the initial scenario. The receiver is the ANS which is the brokering agent as explained above. As in the capability advertisement messages the ontology specified in the outer message is `capabilities` and the content language is KQML.

The inner message in the content field is the message the **pp**-agent wants a PSA to process. The performative is `achieve` because the **pp**-agent wants some condition to be made true. The sender given in the inner message must be the same agent that is sender in the outer message, i.e. the agent seeking the capability. Finally, the content language is CDL and the content must be a description of the condition to be achieved in CDL.

The content of the inner message is basically a description of the problem. Where the CDL expression in the capability advertisement contained an achievable condition, the CDL expression in the capability seeking message contains the condition to be achieved. For example, the power plant wants the condition in

which its generator is fully functional to be made true. There is also the equivalent of qualification here. Namely, conditions which are required by the capability seeker to be true can be used as applicability conditions in the capability advertisement (e.g., requiring that the object is a generator that is to be made fully functional). Conditions to be achieved, amended with conditions provided will be the core of problem descriptions in CDL.

The next step is for the broker to find a capability of a PSA that matches the given task description. How exactly this finding and matching work will be described in chapter 5. For now, let us assume that the broker found the **ec**-agent and the **h1**-agent as PSAs capable of dealing with the described problems. The broker should now forward the capability descriptions of these agents to the PHA as follows:

```
(forward
  :sender   ANS
  :receiver pp
  :ontology capabilities
  :language KQML
  :content  (achieve
              :receiver ec
              :ontology OPlan
              :language CDL
              :content  (<machine fully functional>)))


(forward
  :sender   ANS
  :receiver pp
  :ontology capabilities
  :language KQML
  :content  (achieve
              :receiver h1
              :ontology OPlan
              :language CDL
              :content  (<injured people treated>)
```

Forwarding of messages in KQML is done with the `forward` performative. The sending and receiving agents should be obvious and ontology and language are as before. The content field should be the content of the message that advertised the

capability in the first place. Note that it is necessary to include in this message the name of the PSA so that the PHA will be able to find it subsequently.

Now the **pp**-agent should be able to use the forwarded capability advertisements to formulate messages to the respective PSAs asking them to perform their capabilities on its problems. The according messages should look as follows:

```
(achieve
  :sender   pp
  :receiver ec
  :ontology OPlan
  :language CDL
  :content  (<generator fully functional>))

(achieve
  :sender   pp
  :receiver h1
  :ontology OPlan
  :language CDL
  :content  (<injured person treated>))
```

There is very little protocol for what should happen next in KQML and we do not intend to fully specify it here. Much of the following message exchange obviously depends on the exact nature of the problem and how the PSAs get on with solving it.

3.3 More Complex Scenarios

In this section we will introduce some more interesting agents and scenarios that will be used in the thesis to highlight the usefulness of the two properties of the capability description language: expressiveness and flexibility. This section constitutes a major part for the motivation for the work described in this thesis.

3.3.1 Expressive Capability Descriptions

The two hospitals on Pacifica are based in Calypso and Abyss and their capability descriptions for the initial scenario are virtually identical. One way of adding complexity to the initial scenario to make it more interesting is to divide the island such that one hospital deals with problems in one part and the other hospital deals with problems in the other part of Pacifica. The map of Pacifica (figure 3.1) suggests that the first hospital should deal with cases in Calypso and Delta and the second hospital should deal with cases in Abyss and Exodus. Barnacle lies between the two hospitals and may be served by both.

Such a change would not show in the achievable conditions in the capability descriptions of the two hospitals. It would, however, alter their qualifications, the conditions for applicability of their advertised capabilities. For example, the CDL expression in the capability advertisement for the first hospital should represent that it can achieve states in which injured people in Barnacle, Calypso, or Delta have been treated. In terms of expressiveness, this capability description requires CDL to be able to handle disjunctions in conditions which were not necessary in the initial scenario. Disjunctions require a greater expressiveness of CDL.

To make the scenario even more complex, suppose that Pacifica is somewhere off the coast of Iceland where the weather is often inclement. The first hospital has an ambulance, but if there is snow or ice on the roads then snow chains need to be fitted to the ambulance before it can fetch injured people. Thus, one

applicability condition for the first hospital's capability is to have snow chains. This condition is itself conditional, and its condition is disjunctive.[7] If we want the capability description for the first hospital to reflect this additional condition we need even more expressiveness in CDL.

For the following scenario, let the problems at the power plant be as before: a broken generator and an injured person. Also, let the broker know that the weather is bad, i.e. there may be ice or snow on the roads, and that the first hospital's ambulance has snow chains. Then the script for the *expressiveness scenario* is as follows:

Example 3.2 (Expressiveness Scenario) *The first thing that has to happen is that the* **pp***-agent contacts the broker to ask for agents that can deal with its problems.*

Now the broker has to look through its data base of agents and capabilities to find PSA*s with sufficient capabilities to solve the problems described by the* **pp***-agent. As before, the engineering company can deal with the broken generator.*

For the injured person the two hospitals are potential PSA*s. However, the second hospital does not serve Delta where the power plant is based. The first hospital does, and since its ambulance has snow chains its capability is applicable. Thus, the broker finds the first hospital as capable of dealing with the injured person. According messages will be sent to the* **pp***-agent.*

With the knowledge of which PSA*s can solve the* PHA*'s problems the* **pp***-agent can now contact the* **ec***-agent and the* **h1***-agent and ask them to actually solve the problems.*

Finally, the **ec***-agent and the* **h1***-agent can go ahead and solve the given problems.*

Obviously, a number of variations can be generated from this scenario by varying the road conditions and availability of snow chains to the first hospital's

[7] The formal representation of this capability in CDL will be shown in section 4.5.1.

ambulance, or by dropping the splitting of the island between the two hospitals. The scenario above is one of the most interesting cases, since the conditional applicability condition has to be evaluated, but which part of the disjunctive condition holds, ice or snow on the road, is unknown. The expressiveness of CDL will be discussed in chapter 7. The way that some of the approaches described in chapter 2 would have represented the first hospital's capability and performed in this scenario will be discussed in chapter 9 in detail.

3.3.2 Flexible Capability Descriptions

For the following scenario let us ignore the engineering part of the problem of the power plant and just look at the injured person. The PSAs that can deal with this problem in principle are the two hospitals. For this scenario we shall change their capabilities slightly and introduce a new PSA: an ambulance service. These agents will advertise the following capabilities:

- the first hospital/**h1**-agent: The main capability of this hospital is that it can treat injured people. However, it does not have an ambulance in this scenario and thus, an applicability condition is now that the injured people to be treated must be at the hospital. The hospital is in Calypso.

- the second hospital/**h2**-agent: The main capability of this hospital is also that it can treat injured people. It has an ambulance to transport injured people from Abyss, Barnacle, or Exodus to the hospital in emergencies. Delta and Calypso are considered too far away and the ambulance cannot be spared for such a long time.

- the ambulance service/**as**-agent: The main capability of this agent is that it can transport injured people; it cannot treat them. The ambulance service has an ambulance that can be used only to transport people between any two places. The ambulance is based in Delta.

What is interesting in this scenario is that the different PSAs require different expressiveness in their capability descriptions. The second hospital has a disjunctive applicability condition: injured people must be in Abyss, Barnacle, or Exodus. The first hospital and the ambulance service do not need disjunctions and thus, require a less expressive capability representation.

Let us look at the broker next. The broker accepts and stores capability descriptions from the PSAs. On receipt of a request from a PHA it will try to find agents the capabilities of which it knows about to solve the given problem. Suppose the broker has two options here:

1. Find a single agent that can solve the problem. The broker checks for all PSAs whether they alone have the required capability. Inferences over the capability descriptions are limited and thus the broker can do this for any agent.

2. Find a sequence of agents that can solve the problem. Suppose that the broker has a simple planner built in that it can use to generate partial-order plans involving the capabilities described to it. However, this planner can only handle simple capability descriptions in CDL that do not involve disjunctive conditions.

The fact that the broker has two different ways of finding a solution to the given problem is crucial for this scenario. With these two options the broker has the flexibility to exploit the inferences it can make about less expressive representations.

The final change concerns the way the PHA wants the problem handled. For this scenario it will ask the broker to manage the solution of its problem rather than recommending a PSA the PHA has to contact itself. The script for the *flexibility scenario* is as follows:

Example 3.3 (Flexibility Scenario) *The first thing which happens is that the* **pp***-agent contacts the broker to ask it to deal with its problem, the injured person.*

Now the broker has to look through its data base of agents and capabilities to find PSA*s with sufficient capabilities to solve the problems described by the* **pp***-agent. It will first look for single agents that can deal with the problem. The broker works out that the* **h1***-agent cannot help because it requires the injured person to be at the hospital. Similarly, the* **h2***-agent does not have the necessary capabilities because its applicability conditions state that the injured person must be at Abyss, Barnacle, or Exodus. Finally, the* **as***-agent cannot treat people at all. Hence the broker has failed with its first option to find a single agent that is capable of solving this particular problem.*

The broker will now try its second option, finding a plan involving the capabilities of several agents. In the example this means the broker will exclude the **h2***-agent from its planning attempt to find agents to solve the problem because of the disjunction in its applicability condition, i.e. because this capability corresponds to an operator with a disjunctive precondition which cannot be handled by the broker's planner. Only the* **h1***-agent and the* **as***-agent remain and the broker should be able to work out that their combined capabilities suffice to solve the problem.*

The broker can now contact the different PSA*s involved in this plan and monitor the execution. The* **as***-agent and the* **h1***-agent can go ahead and solve the given problem. Finally the broker reports success to the* **pp***-agent.*

This scenario illustrates the flexibility of CDL because the broker knows what kinds of inferences it will want to make and can look at the CDL descriptions to see whether the inferences are supported. Thus capability descriptions may use different levels of expressiveness which will restrict the inferences the broker can make. This trade-off is not surprising. The flexibility of CDL stems from the fact that it allows arbitrary expressiveness in its descriptions and works out

what inferences it needs to make and whether this is possible only when this is required.

This scenario could be extended to one where the broker has several more planning algorithms (or other methods) available which could be applied depending on the inferences supported by the expressiveness used within the CDL descriptions of different agents, e.g. a planner which could cope with disjunctive preconditions.

It is also worth noting that in the above scenario the availability of the **h1**-agent is crucial for the successful solution of the problem. Due to the disjunction in the capability description of the **h2**-agent the broker cannot generate plans involving this agent, i.e. it cannot combine the **as**-agent and the **h2**-agent to solve the given problem. This could be easily fixed though if the **h2**-agent advertised the additional capability to treat patients that are at this hospital.

Obviously, a number of further variations can be generated from this scenario by changing the capability descriptions of the three PSAs. The scenario above is one of the most interesting cases though as it involves different levels of expressiveness for the different agents' capability descriptions, resulting in a problem that illustrates the need for a flexible capability description language. The flexibility of CDL will be discussed in chapter 8. The way that some of the approaches described in chapter 2 would have performed in this scenario will be discussed in chapter 9 in detail.

Chapter 4

A Capability Description Language: CDL

At this point we have looked at the knowledge we need to represent in the messages exchanged during capability brokering and several areas of AI that need to represent similar knowledge. Our aim is to define a new capability description language that will be expressive and highly flexible and can be used to reason about capabilities. In the next step towards this goal we will define our new capability description language, CDL, *that will be used for capability brokering. The contribution of this chapter will be the definition of the different aspects of* CDL, *including its syntax and various examples to illustrate the language.*

4.1 Problems for Capability Representations

In this section we will look at problems with approaches to representing capabilities, described in chapter 2, when they are used for capability brokering. We will also highlight crucial ideas that we will adopt for our capability description language. This section sets the frame for the capability language that follows.

In the previous chapter we outlined a number of scenarios that involve the representation of and reasoning about the capabilities of various problem-solving

agents (PSAs). Furthermore, we have described how the broker we envisage is supposed to respond to various messages from other agents (cf. section 3.2). What we have omitted in this description is a definition of the format of the capability representations which are the content of the capability advertisement messages. The new *Capability Description Language*, CDL, presented in this thesis provides this.

4.1.1 Desirable Characteristics for CDL

The first step towards a new capability description language must be a characterisation of the *properties* or *attributes* we want this language to have.

The two most important properties we want our capability description language to have are *expressiveness* and *flexibility*. These are exactly the properties the expressiveness scenario (example 3.2) and the flexibility scenario (example 3.3) are meant to illustrate and thus, CDL must have these properties to allow for the realisation of these scenarios.

Our aim is to use CDL for brokering. When designing a knowledge representation language it is important to take into account what kind of reasoning one wants to perform over this language. Thus, another characteristic we would like CDL to have is that it is similar to languages which have been *used for capability brokering* successfully, as this would indicate that CDL, too, can be used for brokering. Likewise, since capabilities can be seen as actions one can perform (cf. section 4.2.1), we would also expect CDL to be similar to representations that have been *used to represent and reason about actions*.

As we expect the broker to perform its services autonomously, it is important that the capability representations are in some *formal* language; CDL must have this attribute. Finally, every representation must *have a semantics* to qualify as a representation in the first place [Hayes, 1974], so we shall pay attention to this property as well.

	2.1 brokers	2.2 logics	2.3 action reps.	2.4 models of PSMs
expressive	medium	high	medium	high
flexible	(yes)	no	some	no
brokered	yes	no	(yes)	(yes)
actions	no	no	yes	(yes)
formal	yes	yes	yes	(no)
semantics	(yes)	yes	(yes)	no

Table 4.1: Properties of different approaches

4.1.2 Preliminary Evaluation

Given this characterisation of desirable properties for CDL, we can now evaluate the approaches described in chapter 2 to identify which of them have the above properties. The results of this preliminary evaluation are summarised in table 4.1. For simplicity, we have only listed the four general areas described in sections 2.1 to 2.4. Each of these areas comprises a number of approaches and the table obviously over-generalises and thus, should be seen as a table of general trends rather than an exact evaluation. A more detailed comparison of CDL with other approaches will follow in chapter 9.

The highest *expressiveness* can be found in logics and models of problem solving.[1] Classical first-order predicate logic [Chang and Lee, 1973, Loveland, 1978, Gallier, 1986] has been used to represent many different kinds of knowledge and can thus be considered an expressive representation. However, many other logics offer still more expressiveness to allow the representation of highly complex circumstances (cf. section 2.2.2). Models of problem solving often allow natural language as at least an aspect of their representation which accounts for their high expressiveness (cf. section 2.4.1.2). Most action representations (section 2.3.1) have only restricted expressiveness as they were designed to be used in formation of plans which in itself is a very complex process. There are, however, some action representations that offer more expressiveness, e.g. ADL [Pednault, 1989]. The representations used by the brokers we have described in section 2.1.3 are

[1] A more elaborate discussion of the expressiveness of CDL will follow in chapter 7.

more difficult to classify as they are vague on what exactly the representation of capabilities they use will look like. Closer inspection reveals that, although they mostly allow KIF [Genesereth, 1991, Genesereth *et al.*, 1992] as at least one possible content language, the restrictions imposed are rather severe (cf. section 9.1).

The highest *flexibility* of the representations we have looked at can be found in brokers and in some action representations[2]. Most brokers are based on KQML (cf. section 2.1.2.3) which specifies that capabilities are to be described as KQML messages that can be processed. Thus, the capability description language is KQML, a language designed to have an opaque content which is expressed in a language specified at the wrapper level. In practise though, most brokers only allow a very limited range of languages that can be used as content in capability descriptions in KQML (cf. section 2.1.3). Most action representations (section 2.3.1) have very little flexibility, but there are a few noteworthy exceptions, e.g. SPAR [SPAR, 1997, Tate, 1998]. Like KQML, these languages allow the plugging in of different content languages which gives them their flexibility. Logics (section 2.2), although they provide a wide range of formalisms do not individually have this flexibility. Finally, models of problem solving (section 2.4) usually allow for some parts of their representations to be natural language descriptions and thus, cannot be considered flexible.

The next aspect we have looked at is whether the representation has been *used for brokering*. Obviously, the KQML-based representations described in section 2.1.3 satisfy this criterion, but they are not the only ones. Action representations (section 2.3.1), in fact, can also be seen as having been used for brokering, as a planner that uses these representations at some point also needs to retrieve an action that can achieve a given effect. This is essentially the task performed during capability retrieval. A similar case could be made for the situation calculus (section 2.2.1) which is based on first-order logic, but it is really the ontology of the situation calculus that facilitates brokering, not the underlying representation.

[2] A more elaborate discussion of the flexibility of CDL will follow in chapter 8.

Thus, we are inclined to say that logics have not been used for brokering, allowing for exceptions. Models of problem solving (section 2.4) are again a borderline case as there are now several projects underway that are aimed at building brokers for problem-solving methods (PSMs) (cf. section 2.4.1.5). However, neither their representations nor their brokering mechanisms are defined yet.

The obvious representations that have been *used for representing and reasoning about actions* are, of course, the action representations (section 2.3.1). Models of problem solving (section 2.4) have also been used to represent and reason about actions, but the actions are usually restricted to the reasoning actions performed by some expert system. Still, reasoning about actions is what these representations were designed for. Brokering representations (section 2.1.3) and logics (section 2.2) both have also been used to represent and reason about actions, but this is not what they were specifically designed for.

As for the *formality* of the representation, the only area that does not qualify here are models of problem solving (section 2.4) because they usually allow for natural language as one aspect of their representation. As usual, there are exceptions, e.g. ML^2 (cf. section 2.4.1.2). However, ML^2 is so heavily logic-based that one could well count it into this area anyway. Closely related is the question of *semantics*. The area that has been most concerned with formal semantics is logics (section 2.2) in which almost every formalism has a well-defined formal semantics, otherwise it does not qualify as a logic. The semantics of action representations (section 2.3.1) and KQML (section 2.1.2.3) have also been defined to some degree, but there remain questions [Kuokka and Harada, 1995b] and descriptions are often informal. Finally, models of problem solving (section 2.4) which are based on natural language fail here.

For the capability description language described in this chapter we want to retain the ideas behind these approaches that made them perform well in certain respects. In summary, we want our representation:

- to preserve the structure found in action representations;

- to benefit from the expressiveness of highly powerful logics and the well-defined formal semantics that comes with them;
- to retain the flexibility of KQML by allowing for opaque content languages and to use the communication approach to brokering;
- to be formal to allow for autonomous brokering.

4.2 Achievable Objectives

In this section we will define the core of our capability description language. This will include basic concepts, syntax, and examples to illustrate the language.

Since we want to base our capability description language CDL, which is to be presented in this chapter, on the structure found in action representations as used for AI planning, it is probably worth first asking what the *difference between an action and a capability* is.

4.2.1 Capabilities and Actions

Most action representations in AI are representations that describe how the state of the world changes when an action is performed and what needs to be true before that action can be executed. Capability descriptions need to *convey very much the same knowledge*, i.e. what changes a capability can bring about and what needs to be true for that capability to be applicable. There are two major differences though:

- **Level of description**: An action is less *abstract* than a capability in the sense that we would expect all its parameters to be instantiated for its execution. However, AI planning systems use operator schemata rather than instantiated actions as input, i.e. they effectively use capability descriptions.
- **Modality**: A capability is an action that *can* be performed (in theory), i.e. it has a different modality. But this is implicitly what an AI planner usually assumes when it generates a plan; that the operator schemata it instantiates and inserts into the plan represent capabilities of some agent (cf. [McCarthy and Hayes, 1969, pages 470–477]).

Despite these differences, the knowledge contained in action representations and capability descriptions is very similar because both representations basically

represent the same types of entities. Thus, the language for representing such entities should be very similar, too, and we have already mentioned that CDL will inherit the structure of action representations.

However, there are some *further differences* between actions and capabilities. For example, capability descriptions, as we envisage them, do not require hierarchical decompositions which are used in many modern planners (cf. section 2.3.1.5) unless the description is meant to also express how to perform a capability. Thus, we will base CDL mostly on non-hierarchical action representations (cf. section 2.3.1.2). Another difference is that the representations used by AI planners are usually not only concerned with actions, but also with representing plans of actions. CDL will not be concerned with the representation of plans. Other differences come with the requirements connected to the intended usage of the representation. Propositional STRIPS planning is already a PSPACE-complete problem [Bylander, 1994] and thus, a more complex action representation is not practical for planning. Other tasks like capability retrieval or assessment have a different complexity and thus, allow for different complexity in the representation. The flexibility of CDL is meant to address this issue.

4.2.2 The Knowledge in Capability Representations

We are now in a position to describe the *knowledge* contained in a CDL capability representation. The core CDL representation for achievable objectives is based on a classical, non-hierarchical operator description (cf. section 2.3.1.2) and consists of the following parts:

- **Inputs**: This part of the capability representation specifies the objects an agent possessing this capability receives as inputs to this capability. How these inputs will be used is unspecified here. This part of the representation will be a syntactically defined expression containing symbolic variables which the actual inputs will have to match.

- **Outputs**: This part of the representation specifies the objects that will be the outputs this capability generates. Again, this will be a syntactically defined expression containing symbolic variables the actual outputs will have to match.

- **Input Constraints**: This part of the capability representation defines the constraints that are expected to hold in the situation before this capability can be performed, i.e. the constraints for the capability to be applicable. Free variables in these constraints can only be from the syntactic expression which describes the inputs.

- **Output Constraints**: This part of the representation defines the constraints that are expected to hold in the situation after this capability has been performed. Free variables in these constraints can be from the syntactic expressions which describe the inputs or outputs.

- **Input-Output Constraints**: This part of the representation defines the constraints across input and output situations that must hold. Free variables in these constraints can be from the expressions describing the inputs or outputs.

The first difference between this representation and classical non-hierarchical representations for operators like the STRIPS representation is that there is *no identifier* for the capability. We believe that the introduction of such an action name at this point would not be epistemologically adequate as described by the knowledge representation hypothesis [Smith, 1982]: the action name might help a human reader of a capability description to understand the capability but is not necessary for reasoning about the capability. We will, however, introduce action identifiers into CDL at a later stage when we have a reason to do so (cf. section 4.3).

The next difference between capability descriptions in CDL and STRIPS-like operator descriptions is that CDL distinguishes *two types of parameters*: inputs and outputs. Parameters are essentially the objects involved in the performance

of a capability and must all be instantiated for the execution of a specific action instance. CDL distinguishes input objects, i.e. objects that exist in the situation before the capability is applied, and output objects, i.e. objects that exist only in the situation that results from the application of this capability in the input situation. Output objects do not exist in the input situation, but input objects may or may not exist in the output situation. The reason for introducing this distinction in CDL is that it simplifies the matching of capabilities and problems slightly (cf. section 5.1.2.2).

For example, consider the capability of sorting the elements in a list. The list itself is an input object to this capability. If the sorting is performed by modifying the given list then there is no output object to this capability. Otherwise there will be an output object, namely the new, ordered list that exists in the output situation only.

Input constraints in CDL *directly correspond to the precondition formula* in classical non-hierarchical action representations. In accordance with modern planning formalisms (cf. section 2.3.1.5) we prefer to view the precondition formula as a constraint on the situation in which the capability can be applied. Notice that input constraints may only mention objects from the inputs as these are the only objects that exist in this situation. *Output constraints in* CDL *correspond to a combined add and delete list*, i.e. to the effects of an action, and represent constraints on the situation that results from the application of this capability. Postconditions that would occur in the delete list in a STRIPS-like representation will be negated in the output constraints in CDL. Finally, output constraints may mention objects that exist in the output situation, i.e. objects from inputs or outputs.

For example, in the list sorting capability mentioned above, the fact that all elements of the given list to be sorted must be elements of the domain of the ordering relation used is a constraint on the input situation, and the fact that the output list is ordered is a constraint on the output situation.

The final set of constraints mentioned above are the *input-output constraints which correspond roughly to secondary preconditions and effects* in ADL [Pednault, 1989] or UCPOP [Penberthy and Weld, 1992, Barrett *et al.*, 1995]. These constraints do not refer to only one situation like the input and output constraints but are constraints across both of these situations. This type of constraint allows one to refer to objects that have different properties in different situations and to expresses a condition on the properties in these different situations.

Returning to the list sorting example where sorting is performed by modifying the original list, one constraint that one must express is that input and output lists contain the same elements. If we sorted by generating a new list we might be able to express this constraint as separate input and output constraints, but if we modified the list this constraint cannot be expressed by referring to one situation only. In the input situation we can only refer to the unsorted list and in the output situation we can only refer to the now ordered list.[3]

4.2.3 Decoupling the Representation

At this point we know what the knowledge is we need to represent in CDL. The next obvious question is *what language* to use to express the different constraints in. As mentioned above, our aim is to inherit the expressiveness and well-defined semantics of logics (section 2.2) in CDL, but we also want to retain the flexibility of KQML (cf. section 2.1.2.3).

4.2.3.1 Integral Action Representations

At heart, many knowledge representation languages are *state representation languages*, i.e. they implicitly assume the world to be in exactly one state or situation at any given time. That is, unless otherwise stated, a set of sentences in such a

[3] Input-output constraints are not required in our example scenarios and thus, we shall only return to them in section 5.2.1 where a slightly modified version of the initial scenario will be introduced.

language is assumed to refer to the same implicit situation. Knowledge representation languages usually also assume that there exist a number of objects in this implicit situation and that certain relations hold between these objects in this situation. The logics described in section 2.2 mostly fall into this category of knowledge representation languages with the notable exception of dynamic logic [Harel *et al.*, 1982, Harel, 1984]. Thus, these logics would qualify as languages which can be used to express constraints on single situations.

Using state representation languages to reason about actions has proven difficult. The most commonly used knowledge representation language that makes the above assumptions is first-order logic [Chang and Lee, 1973, Loveland, 1978, Gallier, 1986]. It is possible to represent and reason about actions in first-order logic as demonstrated by the situation calculus (cf. section 2.2.1), but this leads to a number of problems; most prominently the frame problem. Hence the development of specific action representation languages such as the STRIPS representation, which avoids the frame problem by making the STRIPS assumption, i.e. nothing changes that is not mentioned in the operator description [Tate *et al.*, 1990, page 37]. By adopting the structure of such action representations we have also adopted this convenient approach to the frame problem.

In most conventional action representation languages such as STRIPS, the state representation language is an integral part of the overall representation language. We shall call such languages *integral action representations*. For example, STRIPS [Nilsson, 1980, chapter 7] only allowed conjunctions of positive literals in the input and output constraints of its representation. However, it is relatively trivial to extend the state language to allow for more complex formalisms, e.g. horn clauses, full first-order logic, modal logics, etc. However, with an integral action representation we have to commit to one of these languages and every new state representation language defines a new action representation. It is this inflexibility that we seek to avoid in CDL as it is not clear which would be the right state language for describing arbitrary agent capabilities.

4.2.3.2 Decoupled Action Representations

To allow the arbitrary combination of action and state representation we will define the action representation language independent from the state representation language. We shall call this a *decoupled action representation*, i.e. a full action representation consists of a decoupled action representation combined with a state representation language. Syntactically, decoupling will be achieved using an approach similar to the way KQML allows content expressions to be in some independent content language (cf. section 2.1.2.3), i.e. by having a field that names the content language and one that holds exactly one expression in this language as a sub-expression of the wrapper. CDL will also allow the nomination of a state language in which the different types of constraints are to be expressed, except that there will be several sub-expressions in the named content language in CDL. By decoupling the action from the state representation, CDL will achieve the same, high flexibility that KQML provides.

The obvious *advantage* of such a decoupled action representation over its conventional, integral counterpart is that it allows one to plug different state representation languages into the same decoupled action representation language, i.e. it has flexibility. Thus, whether we are using CDL for states with conjunctions of literals or with full first-order logic plugged in, we are still using the same decoupled action representation. Decoupling of action representations also allows us to compare action representations at two different levels. For example, a decoupled version of STRIPS would be a different action representation from CDL even with the same state representation language, as STRIPS does not allow for input-output constraints.

Defining a decoupled action representation language in this KQML-like way is not the difficult part though. The problem is how to reason over such a combined language, e.g. with a broker. We shall return to the question of how to reason about CDL in chapter 5 and some problems with the implementation of decoupled languages shall be discussed in chapter 8.

```
<cdl-descr> ::= ( capability
                    :state-language <name>
                    :input ( <param-spec>+ )
                    :output ( <param-spec>+ )
                    :input-constraints ( <constraint>+ )
                    :output-constraints ( <constraint>+ )
                    :io-constraints ( <constraint>+ )

<param-spec> ::= ( <name> <term> )

<term>       ::= <constant> | <variable> |
                  ( <constant> <term>+ ) |
<variable>   ::= ?<name>
<constant>   ::= <name>

<constraint> ::= << expression in state-language >>
```

Figure 4.1: Syntax of core CDL in BNF

4.2.4 Syntax of the CDL Core

We are now in a position to define the *syntax of the core* of the capability description language CDL. A number of extensions of this syntax will be described in the following sections. The syntax will be based on a KQML-like balanced parenthesis list and the BNF of CDL is given in figure 4.1.

A *capability description* in CDL begins with an open bracket "(", which is followed by the word `capability`, indicating that this is the description of a capability held by some agent. This is followed by a number of keyword-value pairs as in KQML. The keyword `:state-language` must be followed by a state language identifier. Although the BNF does not indicate this, all but the first keyword-value pair are optional in CDL. Even the state language specification could be omitted if there were no constraints specified for this capability, but this does not appear to be a useful capability. The remainder of the CDL description specifies the keyword-value pairs for the inputs, the outputs, and the various

types of constraints explained above.

Both inputs and outputs are lists of *parameter specifications* similar to the parameters of a STRIPS-like operator description. A parameter specification in CDL is a pair consisting of an identifier `<name>`, and a term. The identifier specifies which role [Brachman, 1979] this parameter plays for this capability. The term specifies the object that will fill this role. In capability descriptions these terms will usually be variables or function terms containing variables. Thus, the specification of the role filler in a parameter specification in CDL is not dissimilar from the specification of the arguments in the definition of a Prolog predicate. The main difference is that arguments are explicitly named by the role name. This allows one to specify the parameters to this capability in an arbitrary order.

For example, the specification `:input ((BrokenMachine ?machine))` for the inputs of the capability of the engineering company in the initial scenario specifies that there is just one input parameter to this capability, that this parameter plays the role of the `BrokenMachine` for this capability, and that the object that will fill this role is represented by the variable `?machine` in the constraints of this capability description.

Finally, the BNF of CDL specifies that the various types of *constraints* in the capability description will in fact be lists of constraint expressions, but it does not define the syntax for them. Of course, this is because this is the point where CDL allows the plugging in of an independent state description language, i.e. the syntax of CDL is open at this point because CDL is a decoupled action representation. The only indication of what the constraints will look like is given as the value of the `state-language` field which names the language in which all constraints have to be expressed. Thus, the syntax of CDL is described completely at this point.

However, to be able to actually write capability descriptions in CDL it is, of course, necessary to define at least one *state language* that can be used to represent the constraints in a CDL description. For the capability descriptions in the initial

```
<formula>    ::= ( <quant> <c-form> ) | <c-form>

<quant>      ::= ( <quantifier> <varspec>+ )
<quantifier> ::= forall | exists
<varspec>    ::= <variable>

<c-form>     ::= <literal> |
                 ( not <formula> ) |
                 ( and <formula> <formula>+ ) |
                 ( or <formula> <formula>+ ) |
                 ( implies <formula> <formula> ) |
                 ( iff <formula> <formula> ) |
                 ( xor <formula> <formula> ) |

<literal>    ::= <constant> |
                 ( = <term> <term> )
                 ( <constant> <term>+ )
```

Figure 4.2: Syntax of FOPL in BNF

scenario we have implemented a language that is essentially first-order predicate logic [Chang and Lee, 1973, Loveland, 1978, Gallier, 1986]. The syntax of this language resembles a subset of KIF [Genesereth, 1991, Genesereth *et al.*, 1992] and is given in figure 4.2. We shall not describe the meaning of the different syntactical categories here as they are all fairly intuitive.

One caveat here is that the syntax of the state language refers back to the syntax of CDL for the definition of *terms*. This is because we allow terms in the parameter specifications of CDL descriptions and in the content language. Thus, terms are shared across the wrapper and the content level of CDL. However, the underlying assumption made by CDL at this point is that the state language will consist of expressions that relate objects to each other and these objects are described by sub-expressions called terms. We believe this to be very reasonable since most knowledge representation languages are based on a semantics that

satisfies this assumption. An exception is propositional logic and it is not clear to us what the parameter specification in a propositional action representation could mean. Thus, we believe the sharing of terms between CDL and its content language to be acceptable.

However, the above definition of CDL and its content language requires more than the state language to consist of expressions that relate objects to each other, it requires a *shared syntax* for terms. This problem could easily be addressed by defining a separate term definition language that has to be plugged into the decoupled action representation just like the state language. Although this would allow for even greater flexibility in the action representation, we believe that the added complexity in the expressions is not worth the effort because most knowledge representation languages have epistemologically very similar term specifications anyway. Hence we have chosen to implement CDL with a shared syntax for terms as described above.

4.2.5 Examples from the Initial Scenario

Now that we have defined the core of CDL it is time to look at some *simple examples* that illustrate how capabilities can be represented in CDL. All the examples in this section are the content of KQML messages required for the initial scenario as described in section 3.2. Further examples will follow in section 4.5.

4.2.5.1 Capability Advertisements

The first group of messages required for the initial scenario are the *capability advertisement messages* with which the problem-solving agents (PSAs) inform the broker of their capabilities. We have already described these KQML messages at the wrapper level in section 3.2.1. The content describing the actual capability was only given informally at this point simply because CDL was not yet defined. Now we are in a position to specify the content formally. The first PSA we present

which advertises a capability is the engineering company represented by the **ec**-agent. The content of its capability advertisement in CDL is as follows:

```
(capability
  :state-language fopl
  :input ((BrokenMachine ?machine))
  :input-constraints (
    (elt ?machine Generator)
    (Is ?machine Broken)
    (Has Location ?machine Pacifica))
  :output-constraints (
    (not (Is ?machine Broken))))
```

This CDL expression represents a capability and uses the content language `fopl` to represent constraints on states. It expects just one object as input which plays the role `BrokenMachine` for this capability. To be able to apply this capability three constraints must hold in the situation before the capability can be performed: the object represented by the variable `?machine` must be a generator: `(elt ?machine Generator)`; it must actually be broken: `(Is ?machine Broken)`; and it must be on Pacifica: `(Has Location ?machine Pacifica)`. As a result of the application of this capability the given `?machine` will no longer be broken: `(not (Is ?machine Broken))`, i.e. this constraint will hold in the state after the capability has been performed. This CDL expression does not specify any outputs or input-output constraints.

Similarly, the content of the KQML message that is the capability advertisement of the **h1**-agent will be expressed as:

```
(capability
  :state-language fopl
  :input ((InjuredPerson ?person))
  :input-constraints (
    (elt ?person Person)
    (Is ?person Injured)
    (Has Location ?person Pacifica))
  :output-constraints (
    (not (Is ?person Injured))))
```

This capability description also uses the content language `fopl` and it expects one input which plays the role `InjuredPerson`. Furthermore, three con-

straints must hold in the input situation: the object represented by the variable `?person` must be a person: `(elt ?person Person)`; the person must be injured: `(Is ?person Injured)`; and the person must be on Pacifica: `(Has Location ?person Pacifica)`. After the application of this capability the person will not be injured: `(not (Is ?person Injured))`. The second hospital represented by the **h2**-agent advertises an identical capability and there is no need to repeat this message here.

4.2.5.2 Messages for the Initial Scenario

The first pair of messages in the initial scenario are the messages with which the power plant represented by the **pp**-agent asks the broker to recommend PSAs that can solve its problem. The *problem* itself consists of two parts, a broken generator and an injured person. Hence the **pp**-agent has to send the two messages already described in section 3.2.2 to the broker. As before, the content was left informal at this point in the description and shall be given here. However, the contents of these messages are not capability descriptions but problems. We will use CDL to represent problems, too, only that these have to begin with the word `task` instead of `capability`. Thus, the message that describes the engineering part of power plant's the problem is expressed as:

```
(task
  :state-language fopl
  :input-constraints (
    (elt generator1 Generator)
    (Is generator1 Broken)
    (Has Location generator1 Pacifica))
  :output-constraints (
    (not (Is generator1 Broken))))
```

This CDL expression describes a problem using `fopl` as the state description language. There are three constraints given that hold in the input situation: `generator1` is a generator: `(elt generator1 Generator)`; it is broken: `(Is generator1 Broken)`; and it is on Pacifica: `(Has Location generator1`

`Pacifica)`.[4] The only constraint on the situation that should hold after the sought for capability has been applied is that `generator1` should no longer be broken: `(not (Is generator1 Broken))`.

Notice that, the input and output constraints in this problem are almost identical to the ones in the capability advertisement of the **ec**-agent. The reason for this is simply that, here, we are trying to illustrate what CDL expressions look like, not how the matching works. More interestingly, notice that problem specifications in CDL usually do not specify any parameters. In this example, it is the task of the broker to work out that generator1 has to play the role of the `BrokenMachine` for the **ec**-agent's capability to be applicable to this problem.

The content of the message describing the second part of the **pp**-agent's problem is quite similar to the above:

```
(task
  :state-language fopl
  :input-constraints (
    (elt JohnSmith Person)
    (Is JohnSmith Injured)
    (Has Location JohnSmith Pacifica))
  :output-constraints (
    (not (Is JohnSmith Injured))))
```

Again, the state language is `fopl` and the input constraints specify that `JohnSmith` is a person, injured, and on Pacifica. The only output constraint the seeked capability has to satisfy is that `JohnSmith` must not be injured after the application of the capability.

The next set of messages in the initial scenario are from the broker to the **pp**-agent. KQML specifies that these messages forward the capability advertisement from the PSA that can solve the described problem to the problem-holding agent (PHA). Thus, the content that has again been informal in section 3.2.2 is, in fact, exactly the same as in the capability advertisements described in section 4.2.5.1.

[4] We are aware that this representation might not be epistemologically adequate, but as an illustrative example for CDL it will do for now.

Finally, the PHA, the **pp**-agent in the initial scenario, can send messages to the PSAs asking them to solve the two parts of the problem. Again, the necessary KQML messages have been described with an informal content in section 3.2.2. The content of these messages is the same problem description that was originally sent to the broker, but as this part of the communication is not strictly part of the capability brokering process, other message formats are conceivable at this point.

4.3 Performable Actions

In this section we will extend CDL *to allow for the representation of performable actions which will be based on the description of achievable objectives. This will include basic concepts, syntax, and examples to illustrate the language.*

4.3.1 Achieving Objectives or Performing Actions?

Every capability can be described as *achieving an objective or as performing an action.* For example, the sorting capability mentioned before can be described as achieving a state in which the elements of the given list are ordered. Alternatively, it can be described as sorting the list, i.e. the performance of an action of type sorting on the given list. The former description can be regarded as an objective-centred description and the latter is an action-centred description. Natural language allows us to describe every capability in both ways, although some descriptions might sound awkward to us. Performing an action can be described as achieving a state in which the action has been performed. Achieving an objective can be described as performing an action of type achieving for the given objective. Thus, both descriptions are effectively equivalent.

In the core of CDL described above we have chosen to represent capabilities through objectives they can achieve. Now is the time to briefly *reflect* on this decision.

Capabilities and actions are usually described by *verbs* in natural language because these are things we can do. In fact, most verbs describe actions. The fact that we use this major syntactic category to communicate about capabilities and actions is because it usually is the way we think about these entities. If this is the case then an action-centred description of capabilities can be considered a more direct representation. This in turn can be interpreted as evidence that we should have based CDL on performable actions. However, in classical non-hierarchical action representations the verbs describing the actions usually take

the place of the action name, but in section 4.2.2 we have argued that such an action name does not add to the representation of an action in terms of its objectives. Furthermore, in [Hayes, 1974, section 2] it was argued that the so-called directness of a representation is a questionable concept that does not indicate whether a representation is adequate or not.

However, there is further evidence that we should have based our capability representation on performable actions rather than achievable objectives: most *modern planners* (cf. section 2.3.1) do not use explicit goals in the statement of a planning problem. Instead, they accept an incomplete plan as input, i.e. a complex description of an action. The task for the planner is to refine the given plan until it contains no more flaws. Thus, at this level of description there is no mention of objectives at all. Objectives do however occur in plans as the preconditions of actions or as flaws. Thus, modern planners use an action-centred representation. While this representation is convenient for the planning process itself, it does require a rather awkward specification of a planning problem [Tate *et al.*, 1990, page 28]: every plan contains at least two dummy steps at the very beginning and end which do not represent actual actions that are part of the plan. Thus, the reason for the action-centred view in planning lies in the planning process and not in the epistemological adequacy of this view.

If every action can be described in both ways, we still need to explain *why* we have chosen to base our capability representation on achievable objectives.

One answer is that we have chosen to adopt the *deliberative agent architecture* (cf. section 2.1.2.1) which assumes that every action performed by an agent is goal-directed, i.e. first there is the objective, then there is the action. Furthermore, it is sometimes meaningful to talk about objectives for which there are no capabilities that will achieve them, but ultimately every capability of an agent can be assumed to have some objective, even if this objective or motive is difficult to pin down as it is for altruistic actions. Thus, we consider achievable objectives as more fundamental than performable actions and have based capability descriptions in

CDL on objectives.

A second argument for achievable objectives as the basis of CDL can be found in work on the *indexing problem* for libraries of PSMs (cf. section 2.4.1.4). The indexing problem is very similar to the capability retrieval problem faced by our broker and thus, the experience gained there is relevant to our work here. The initial approach to the indexing problem was to form a hierarchy of PSMs [Breuker and Van de Velde, 1994, page 59]. Nodes in this hierarchy are labelled with actions that characterise the capability this class of PSMs performs. Thus, this approach can seen as based on an action-centred representation. This approach turned out to be inadequate for the indexing problem though. A later approach was to associate PSMs with the problems they can solve [Breuker, 1997]. The problem to be solved was described in terms of objectives that need to be achieved. Thus, this improved approach can be seen as based on an objective-centred representation and is now considered more appropriate for a problem very similar to the problem addressed in this thesis.

Partially, the problem is that *natural language is misleading* when it is used to express tasks with verbs. For example, when we want our generator to be repaired and give this as the action to be performed to the engineering company, we do not really mean that we want the agent to necessarily perform an action of type repairing. What we are really interested in is getting the generator into a fully functional state, i.e. this is our objective. Thus, we have decided to base capability representations in CDL on achievable objectives as described in section 4.2, but we shall provide for a capability description based on performable actions, too.

4.3.2 Extending the Syntax

To think of capabilities in terms of performable actions as opposed to achievable objectives has one major advantage: one can *define a new capability* in terms of other, more primitive capabilities. For example, suppose the broker knew the description of a general sorting action. If a new agent now wants to advertise

the capability that it can sort lists of integers, and this new agent is aware of the broker already knowing about the description of a sorting action, then the new agent could advertise its integer sorting capability based on the description of the sorting action already known to the broker. All the new agent needs to do in this case is refer to the broker's existing description of a sorting action and modify it by stating the additional constraint that the elements of the given list must all be integers.

The knowledge the broker would need to achieve this kind of behaviour is effectively an *ontology of actions* (cf. section 2.3.2). It is conceivable that a broker knowing about a number of primitive actions in an ontology would be much easier to communicate with, as it would not be necessary to represent every new capability completely from scratch. In fact, the two brokers for PSMs described in section 2.4.1.5 both consider an ontology of actions to be at the heart of their brokering mechanism.

Thus, we shall now extend CDL to allow for the representation of performable actions. If the broker has an ontology of actions and another agent wants to define a new capability in terms of an action in this ontology, it needs to be able to refer this action in the ontology in some way. For this purpose we need to introduce *two new keyword-value pairs* into the capability representation in CDL:

- **a capability identifier**: this field allows the specification of a unique action name for a capability, i.e. exactly what we have argued as being epistemologically inadequate above; and
- **a capability inheritance link**: this field allows the naming of an action from which this capability will inherit the description.

The *extended syntax of* CDL that includes these fields is given in figure 4.3. The syntactic categories not mentioned there have not changed from the definition in figure 4.1. The `:action` field is used to specify the name of this action, i.e. the name that can be used in future to refer to this action description, e.g. to inherit

```
<cdl-descr> ::= ( <ctype>
                    :state-language <name>
                    :action <name>
                    :isa <name>
                    :input ( <param-spec>+ )
                    :output ( <param-spec>+ )
                    :input-constraints ( <constraint>+ )
                    :output-constraints ( <constraint>+ )
                    :io-constraints ( <constraint>+ )

<ctype>     ::= capability | task
```

Figure 4.3: Syntax of CDL including performable tasks in BNF

its description. The `:isa` field is used to specify from which action this action inherits, i.e. of which action it is a specialisation.

When a new capability description inherits from an action description in the broker's action ontology, the description of the new capability is effectively a description of how to modify the inherited action description inherited from to obtain the new capability description. We shall call a CDL expression that describes a capability by inheriting from some action a *modification description*. Without further extending the syntax, three principal types of modification possible are:

- **New parameters**: The modification description can specify additional parameters for input and output in the inheriting capability description.
- **Instantiated parameters**: The modification description can give values for parameters defined in the description inherited from, i.e. these parameters are instantiated in the inheriting description.
- **New constraints**: The modification description can specify additional input, output, or input-output constraints involving all the new parameters as well as inherited parameters.

4.3.3 Examples

To illustrate modification descriptions and the inheritance mechanism outlined above we shall now look at some *simple examples*. The following examples represent a minor extension of the initial scenario described in example 3.1. The first thing we need is an ontology of actions known to the broker. For simplicity, we shall describe only one action in this ontology: a moving action. This action will be described as follows:

```
(capability
  :action move
  :state-language fopl
  :input ((Thing ?thing)(From ?p1)(To ?p2))
  :input-constraints (
    (Has Location ?thing ?p1))
  :output-constraints (
    (not (Has Location ?thing ?p1))
    (Has Location ?thing ?p2)))
```

The three parameters are the object that is to be moved (`?thing`), the place from where it is to be moved (`?p1`), and the place to which it is to be moved (`?p1`). The sole constraint on the input situation is that the thing to be moved is at the place from where it is to be moved: `(Has Location ?thing ?p1)`. The output constraints state that `?thing` will not be at the initial location anymore after the action has been performed: `(not (Has Location ?thing ?p1))`; and that it will be at the location it was to be moved to: `(Has Location ?thing ?p2)`. The name of this action is given as `move`. We shall now assume that this action description is known to the broker before it receives any capability advertisements.

Now, suppose the second hospital also wants to *advertise* the capability that it can move patients to the hospital. Of course, this could be done by simply defining a new capability, but it can also be described as a modification of the moving action already known to the broker. Thus, the **h2**-agent could send a second capability advertisement message to the broker with the following content:

```
(capability
  :isa move
  :state-language fopl
  :input ((To Hospital2)(Ambulance ?a))
  :input-constraints (
    (elt ?thing Person)
    (Is ?thing Injured)))
```

This CDL description first states that it inherits from the move action in the broker's action ontology. This action is modified by instantiating the input parameter `(To ?p2)` to `Hospital2`, i.e. the capability can only move objects to this hospital. The description also adds one more input parameter, the `Ambulance` that is to be used in the application of this capability. Thus, the three input parameters of the new capability described here are the object to be moved (i.e. the patient) and the place it is to be moved from, both inherited from the move action, and the ambulance with which the patient is to be moved. The capability description also extends the input constraints, specifying that the object to be moved must be a person: `(elt ?thing Person)`; and that this person must be injured: `(Is ?thing Injured)`. It also inherits the input constraint, `(Has Location ?thing ?p1)`, and the first output constraint, `(not (Has Location ?thing ?p1))`, from the move action. The second output constraint, however, is modified to `(Has Location ?thing Hospital2)` because the input parameter `To`, which is represented by the variable `?p2` in the description of move, has been instantiated to `Hospital2` in the input of the modification description.

The new capability can be matched against problems by the broker just like any other capability. For example, a request to recommend an agent that can deal with the following problem would result in the retrieval of this capability:

```
(task
  :state-language fopl
  :input-constraints (
    (elt JohnSmith Person)
    (Is JohnSmith Injured)
    (Has Location JohnSmith Pacifica))
  :output-constraints (
    (Has Location JohnSmith Hospital2)))
```

Modification descriptions not only apply to new capabilities but also to task descriptions. It is possible in CDL to define tasks by inheriting from actions in the broker's ontology. Thus, another way of specifying the above problem would be the following:

```
(task
  :isa move
  :state-language fopl
  :input ((Thing JohnSmith)(From PowerPlant)(To Hospital2))
  :input-constraints (
    (elt JohnSmith Person)
    (Is JohnSmith Injured)))
```

This CDL expression states that the PHA is looking for an agent that can move `JohnSmith` from `PowerPlant` to `Hospital2` where `JohnSmith` is a person: `(elt JohnSmith Person)`; and he is injured: `(Is JohnSmith Injured)`.

As can be seen from these examples, the specification of a problem by inheriting from an action usually involves the specification of at least some of the input parameters. Previous problems had been described in terms of input and output constraints only, and it was the task of the broker to fill the different roles of a capability to decide whether the described capability can solve the given problem. While this seems to complicate the description of a problem, inheritance does in fact simplify the description by inheriting the constraints that come with the specified action. The way that different types of problems and capabilities will be matched against each other will be described in section 5.2.3.

```
<cdl-descr> ::= ( <ctype>
                    :state-language <name>
                    :action <name>
                    :isa <name>
                    :properties ( <name>+ )
                    :input ( <param-spec>+ )
                    :output ( <param-spec>+ )
                    :input-constraints ( <constraint>+ )
                    :output-constraints ( <constraint>+ )
                    :io-constraints ( <constraint>+ )

<ctype> ::= capability | task

<param-spec> ::= ( <name> <term> )
<term>       ::= <constant> | <variable> |
                  ( <constant> <term>+ ) |
<variable>   ::= ?<name>
<constant>   ::= <name>

<constraint> ::= << expression in state-language >>
```

Figure 4.4: Final syntax of CDL in BNF

4.4 Other Properties

In this section we will show how an agent advertising its capabilities can be represented in CDL *and we describe additional properties which are required to accomplish this.*

The final extension of CDL concerns the fact that there are a number of simple *properties* that an agent might have and which it might want to include in the capability description. For example, an agent might want to advertise that its problem-solving behaviour is complete, i.e. that, if there is a solution to a problem, this agent will find it. This information can be added to a capability description in CDL simply through a list of propositional symbols attached to the capability description. Syntactically this extension leads to another, optional keyword-value

pair with the keyword `:properties`. This keyword must be followed by a non-empty list of propositions. The complete and final syntax of CDL including this extension is given in figure 4.4.

To illustrate this feature of CDL let us reconsider the new capability of the second hospital described in the previous section, namely that it can move patients to the hospital. It also advertises the original capability described in the initial scenario, namely that it can treat injured people. Now, the **h2**-agent cannot be certain that its original capability will have the described result, i.e. that the person the capability has been applied to will no longer be injured. However, the **h2**-agent might be certain that it can at least get an injured person to the hospital, i.e. it might consider its problem-solving behaviour complete with respect to this capability. To state this in its capability advertisement, the **h2**-agent could use the properties feature. The modified capability description for its second capability would thus look as follows:

```
(capability
  :isa move
  :properties (complete)
  :state-language fopl
  :input ((To Hospital2)(Ambulance ?a))
  :input-constraints (
    (elt ?thing Person)
    (Is ?thing Injured))))
```

Note that the properties are properties of the PSA and thus, they should not be expressed as part of the input or output constraints. Task descriptions in CDL can also mention properties, and there the properties are interpreted as properties that the PSA for the described problem must have. For example, the following task description would require the capability described above:

```
(task
  :isa move
  :properties (complete)
  :state-language fopl
  :input ((Thing JohnSmith)(From PowerPlant)(To Hospital2))
  :input-constraints (
    (elt JohnSmith Person)
    (Is JohnSmith Injured)))
```

Properties can be used to express various facts about the PSA. In the above example they have been used to represent the completeness of the problem-solving behaviour of the PSA. Other useful information the property list may convey is the content languages the PSA can handle. Although CDL would detect a PSA not being able to handle a given problem anyway, the properties are a far more efficient way of testing this. There is also a certain similarity between this feature of CDL and the explicit naming of used extensions in PDDL [Ghallab *et al.*, 1998]. However, the property list introduced here refers to properties of the agent while the named extensions in PDDL refer to properties of an expression in PDDL. This is handled automatically in CDL as the reasoning that can be performed over a given state language will automatically determine the reasoning the broker can perform.

4.5 Examples

In this section we will show how the scenarios introduced in chapter 3 and the agents involved in them can use CDL *to represent their capabilities. The content of the messages described in section 3.3 will be given in* CDL *here.*

4.5.1 Expressiveness Scenario

The first scenario we will look at is the *expressiveness scenario* described in section 3.3.1. The agents for this scenario are the same as for the initial scenario. The complexity of this scenario essentially lies in the expressions required to represent certain constraints: The first hospital only treats patients from Barnacle, Calypso, or Delta, and the second hospital only treats patients from Abyss, Barnacle, or Exodus. Thus, both hospitals require disjunctions in their applicability conditions. The capability description of the second hospital was complicated even further by the fact that it requires snow chains for its ambulance if there is snow or ice on the roads.

The first messages of interest to us again are the capability advertisements. As the capability of the engineering company remains unchanged, so does its capability advertisement message:

```
(advertise
  :sender ec
  :content
    (achieve
      :receiver ec
      :ontology OPlan
      :language CDL
      :content
        (capability
          :state-language fopl
          :input ((BrokenMachine ?machine))
          :input-constraints (
            (elt ?machine Generator)
            (Is ?machine Broken)
```

```
            (Has Location ?machine Pacifica))
          :output-constraints (
            (not(Is ?machine Broken)))))
  :ontology capabilities
  :receiver ANS
  :language KQML)
```

The next PSA advertising its capabilities is the **h1**-agent. One way of expressing the condition that the person to be treated must be in Barnacle, Calypso, or Delta, is simply to add this as a new disjunctive input constraint. Since we are using first-order logic as the state language within the CDL expression describing the first hospital's capability, this presents no problem. The complete capability advertising message for the **h1**-agent is expressed as:

```
(advertise
  :sender h1
  :content
    (achieve
      :receiver h1
      :ontology OPlan
      :language CDL
      :content
        (capability
          :state-language fopl
          :input ((InjuredPerson ?person))
          :input-constraints (
            (elt ?person Person)
            (Is ?person Injured)
            (or (Has Location ?person Barnacle)
              (Has Location ?person Calypso)
              (Has Location ?person Delta)))
          :output-constraints (
            (not(Is ?person Injured)))))
  :ontology capabilities
  :receiver ANS
  :language KQML)
```

An alternative way of advertising this capability avoiding first-order logic would be to describe it as three separate capabilities, one for each of the disjuncts in the disjunctive input constraint (cf. [Russell and Norvig, 1995, page 383]).

The next capability advertiser is the second hospital. We will express the condition that the injured person must be in Abyss, Barnacle, or Exodus with a disjunctive input constraint, as we did for the first hospital. However, there is one more condition to represent for this hospital, namely that its ambulance must have snow chains if there is snow or ice on the road. In first-order logic, this can be represented as an implication with a disjunction as its left hand side. The resulting capability advertisement message is shown here:

```
(advertise
  :sender h2
  :content
    (achieve
      :receiver h2
      :ontology OPlan
      :language CDL
      :content
        (capability
          :state-language fopl
          :input ((InjuredPerson ?person))
          :input-constraints (
            (elt ?person Person)
            (Is ?person Injured)
            (or (Has Location ?person Abyss)
              (Has Location ?person Barnacle)
              (Has Location ?person Exodus))
            (implies (or(on Road Ice)(on Road Snow))
              (have Ambulance SnowChains)))
          :output-constraints (
            (not(Is ?person Injured)))))
  :ontology capabilities
  :receiver ANS
  :language KQML)
```

The next message in this scenario comes from the power plant which asks the broker to recommend a PSA that can deal with the engineering part of its problem. This message is the same as in the initial scenario and the complete message is shown below:

```
(recommend-one
  :sender pp
  :content
```

```
    (task
      :state-language fopl
      :input-constraints (
        (elt generator1 Generator)
        (Is generator1 Broken)
        (Has Location generator1 Pacifica))
      :output-constraints (
        (not(Is generator1 Broken))))
  :ontology capabilities
  :receiver ANS
  :language CDL)
```

As in the initial scenario, in reply to this request the broker will forward the capability advertisement of the **ec**-agent to the **pp**-agent. This is because this was the only capability advertisement matching the described problem. Note that the content of this message is generated from the internal representation of the broker which uses unique names for all variables. The complete reply message is expressed as:

```
(forward
  :sender ANS
  :content
    (achieve
      :receiver ec
      :ontology OPlan
      :language CDL
      :content
        (capability
          :state-language fopl
          :input ((BrokenMachine ?machine_3))
          :input-constraints (
            (elt ?machine_3 Generator)
            (Is ?machine_3 Broken)
            (Has Location ?machine_3 Pacifica))
          :output-constraints (
            (NOT(Is ?machine_3 Broken)))))
  :ontology agent
  :receiver pp
  :language KQML)
```

The more interesting part of the problem is, of course, the injured person. The next message from the **pp**-agent to the broker describes this problem to

the broker. There are two minor differences between this message and the corresponding message in the initial scenario. Firstly, the location of the injured person is given as Delta here. This should render only the **h1**-agent capable of solving the problem for the **pp**-agent. Secondly, the performative for this message is `recommend-all`, i.e. the **pp**-agent wants to know about all PSAs that can deal with the described problem. We have made this change to illustrate that only the first hospital has the desired capability. Thus, the complete message is expressed as:

```
(recommend-all
  :sender pp
  :content
  (task
    :state-language fopl
    :input-constraints (
      (elt JohnSmith Person)
      (Is JohnSmith Injured)
      (Has Location JohnSmith Delta))
    :output-constraints (
      (not(Is JohnSmith Injured))))
  :ontology capabilities
  :receiver ANS
  :language CDL)
```

In reply to this message the broker will first forward the capability advertisement of the **h1**-agent to the **pp**-agent, thereby indicating that this agent will be capable of solving the given problem. The complete message is given below:

```
(forward
  :sender ANS
  :content
    (achieve
      :receiver h1
      :ontology OPlan
      :language CDL
      :content
        (capability
          :state-language fopl
          :input ((InjuredPerson ?person_4))
          :input-constraints (
            (elt ?person_4 Person)
```

```
            (Is ?person_4 Injured)
            (OR (Has Location ?person_4 Barnacle)
              (Has Location ?person_4 Calypso)
              (Has Location ?person_4 Delta)))
          :output-constraints (
            (NOT(Is ?person_4 Injured)))))
  :ontology agent
  :receiver pp
  :language KQML)
```

If the broker found the **h2**-agent also capable of solving the **pp**-agent's problem, it should also forward its capability description to the **pp**-agent at this point. However, since the injured person is in Delta the second hospital's capability description should not match the problem and thus, the capability description should not be forwarded. The final message from the broker to the **pp**-agent in reply to the described problem indicates that all the matching capability descriptions have been forwarded at this point. This is done with the following simple KQML message:

```
(eos
  :sender ANS
  :ontology agent
  :receiver pp)
```

While this concludes the expressiveness scenario from the broker's point of view, there remains the additional condition of the second hospital which has not been used for this scenario. Thus, we have decided to alter the problem description slightly to test the brokering for the **h2**-agent's capability. Although the power plant is located at Delta, we have moved the injured person in the problem description to Exodus. Furthermore, we have added the knowledge that there is ice on the road and that the ambulance has snow chains. The resulting problem description is given in the following message:

```
(recommend-all
  :sender pp
  :content
    (task
```

```
      :state-language fopl
      :input-constraints (
        (elt JohnSmith Person)
        (Is JohnSmith Injured)
        (Has Location JohnSmith Exodus)
        (on Road Snow)
        (have Ambulance SnowChains))
      :output-constraints (
        (not(Is JohnSmith Injured))))
  :ontology capabilities
  :receiver ANS
  :language CDL)
```

Since this problem description does satisfy all the second hospital's input constraints the broker should forward the capability description to the PHA, the power plant. The first hospital's capability is not applicable here because of the injured person's location. Thus, there will be two reply messages from the broker to the **pp**-agent which will be expressed as:

```
(forward
  :sender ANS
  :content
    (achieve
      :receiver h2
      :ontology OPlan
      :language CDL
      :content
        (capability
          :state-language fopl
          :input ((InjuredPerson ?person_5))
          :input-constraints (
            (elt ?person_5 Person)
            (Is ?person_5 Injured)
            (OR (Has Location ?person_5 Abyss)
              (Has Location ?person_5 Barnacle)
              (Has Location ?person_5 Exodus))
            (IMPLIES (OR(on Road Ice)(on Road Snow))
              (have Ambulance SnowChains)))
          :output-constraints (
            (NOT(Is ?person_5 Injured)))))
  :ontology agent
  :receiver pp
  :language KQML)
```

```
(eos
  :sender ANS
  :ontology agent
  :receiver pp)
```

4.5.2 Flexibility Scenario

The second scenario we want to look at in this section is the *flexibility scenario* described in section 3.3.2. For this scenario we have decided to ignore the engineering part of the power plants problem. The complexity of this scenario lies in the fact that different agents use differently expressive state languages. With capabilities using the less expressive state language, the broker will still be able to form plans involving those agents' capabilities. Otherwise the broker can only determine whether an agent will be able to solve the given problem alone.

The first messages we need to look at in this scenario are again the capability advertisements, beginning with that of the **h1**-agent:

```
(advertise
  :sender h1
  :content
    (achieve
      :receiver h1
      :ontology OPlan
      :language CDL
      :content
        (capability
          :state-language lits
          :input ((InjuredPerson ?person))
          :input-constraints (
            (elt ?person Person)
            (Is ?person Injured)
            (Has Location ?person Hospital1))
          :output-constraints (
            (not(Is ?person Injured)))))
  :ontology capabilities
  :receiver ANS
  :language KQML)
```

There are two important changes in the capability advertisement of the first hospital compared to the previous scenarios. Firstly, the injured person must

be at the hospital for the capability to be applicable in this scenario. This is reflected in the final input constraint. The underlying reason for this restriction is the assumption that the first hospital has no available ambulance in this scenario. Secondly, the content language used within CDL is specified as `lits` in this message. The reason for this is that all the constraints specified in this capability description are literals only, i.e. there was no need for full first-order logic in this capability description. Whereas we have defined the syntax of first-order logic in figure 4.2, we have not yet defined the language specified in this message: `lits`. The broker will be in the position in which it has never seen this language before. Thus, it will send a message to the sender of the original message containing the unknown language asking it where to find this language. This message will be expressed as:

```
(evaluate
  :sender ANS
  :content
    (ask-resource
      :type language
      :name lits)
  :ontology agent
  :receiver h1
  :language KQML)
```

Now, since the **h1**-agent used this language in its capability advertisement we can safely assume that it knows the language in this scenario. Thus, it can tell the broker where to find this language with the responding message below:[5]

```
(evaluate
  :sender h1
  :content
    (tell-resource
      :type language
      :value (http://www.dai.ed.ac.uk/students/gw/jat/classes
        JavaAgent.resource.fopl.LitLObject)
      :name lits)
  :ontology agent
```

[5] The treatment of languages as resources managed by an agent based on the Java Agent Template will be explained in section 5.3.1 in detail.

```
  :receiver ANS
  :language KQML)
```

The second hospital still uses `fopl` as the content language in its CDL capability description language, and, as for the previous scenarios, we shall assume that the broker knows where to find this language. The reason why the **h2**-agent still uses first-order logic as its state language is the final input constraint. As in the expressiveness scenario, this hospital will only transport injured people from Abyss, Barnacle, or Exodus to the hospital for treatment. This is expressed as a disjunction and thus, it requires first-order logic. The complete capability advertisement message of the **h2**-agent is shown here:

```
(advertise
  :sender h2
  :content
    (achieve
      :receiver h2
      :ontology OPlan
      :language CDL
      :content
        (capability
          :state-language fopl
          :input ((InjuredPerson ?person))
          :input-constraints (
            (elt ?person Person)
            (Is ?person Injured)
            (or (Has Location ?person Abyss)
              (Has Location ?person Barnacle)
              (Has Location ?person Exodus)))
          :output-constraints (
            (not(Is ?person Injured)))))
  :ontology capabilities
  :receiver ANS
  :language KQML)
```

The last capability advertisement comes from a new agent which we have introduced for this scenario: an ambulance service. Essentially, the capability the **as**-agent advertises is that it can transport injured people from any place to any other place. Since this capability is again reasonably simple, it also is based on `lits` as the state language within CDL. The actual message looks as follows:

```
(advertise
  :sender as
  :content
    (achieve
      :receiver as
      :ontology OPlan
      :language CDL
      :content
        (capability
          :state-language lits
          :input ((InjuredPerson ?person)(From ?p1)(To ?p2))
          :input-constraints (
            (elt ?person Person)
            (Is ?person Injured)
            (Has Location ?person ?p1))
          :output-constraints (
            (not(Has Location ?person ?p1))
            (Has Location ?person ?p2))))
  :ontology capabilities
  :receiver ANS
  :language KQML)
```

Notice that on receipt of this message the broker should not need to ask the sending agent where to find the state language `lits`, as it already knows where to find this language from the communication following the capability advertisement of the **h1**-agent.

Thus, the next message in this scenario will be the problem description from the **pp**-agent:

```
(broker-one
  :sender pp
  :content
    (task
      :state-language lits
      :input-constraints (
        (elt JohnSmith Person)
        (Is JohnSmith Injured)
        (Has Location JohnSmith Delta))
      :output-constraints (
        (not(Is JohnSmith Injured))))
  :ontology capabilities
  :receiver ANS
  :language CDL)
```

This problem is essentially the same as in the previous scenarios. The only important difference is the performative used here: `broker-one`. With this performative the PHA does not ask the broker to recommend PSAs that can solve the described problem as before, but to manage the solution of the problem for the PHA.

As described in section 3.3.2, the broker actually has several mechanisms for brokering available. The first mechanism, to find a PSA that can solve the described problem is the one we have used up to now. This is also the first mechanism the broker will try here. However, none of the PSAs the broker knows about have advertised a capability that matches the described problem, i.e. none of the PSAs has the capability to solve the described problem on its own. In this case the broker will try its second mechanism: finding a plan. This mechanism will only be invoked with the `broker-one` performative as a plan cannot be the reply to a recommendation performative in KQML. In this example the broker will find a plan that involves firstly, applying the **as**-agent's capability to move the injured person to the first hospital, and secondly, applying the **h1**-agent's capability to treat the patient.

The messages to the PSA to execute this plan are not included here simply because the current implementation does not provide a plan execution framework. Partially the reason for this is the fact that this would not add to the actual brokering process that is the focus of this thesis, and partially it is because KQML and JAT specifically do not provide sufficient support for such a framework. For a review of work on agents that execute plans see section 2.3.4.

Chapter 5

Algorithms and Implementation of CDL

At this point we have defined the capability description language CDL *that will be used to represent general capability knowledge. Our aim is now to show that* CDL *can be used to reason about capabilities as illustrated in our scenarios and that it is indeed expressive and flexible. The next step towards this goal will be to show how specific problems can be evaluated against capability descriptions in* CDL*. The contribution of this chapter will be the description of the algorithm used to perform this evaluation and its integration into the agent framework chosen for the implementation.*

5.1 Basic Capability Evaluation

In this section we will show how simple capability descriptions in CDL *are represented internally and can be evaluated against simple task descriptions.*

5.1.1 Internal Representation

The broker and CDL are both implemented in the object-oriented programming language Java [Eckel, 1997, Campione and Walrath, 1998]. CDL expressions are

handled as objects in this implementation. Thus, we will first have a brief look at the *structure of* CDL *descriptions*, i.e. what other objects constitute a CDL description.

Definition 5.1 (CDL Description) *A capability description of a capability* $\mathcal{C}$ *in* CDL *is a tuple* $\prec A^{\mathcal{C}}, S^{\mathcal{C}}, id^{\mathcal{C}}, sup^{\mathcal{C}}, I^{\mathcal{C}}, O^{\mathcal{C}}, C_I^{\mathcal{C}}, C_O^{\mathcal{C}}, C_{IO}^{\mathcal{C}}, P^{\mathcal{C}} \succ$ *where:* $A^{\mathcal{C}}$ *is the agent that has capability* $\mathcal{C}$*;* $S^{\mathcal{C}}$ *is the name of the state language used within this capability description;* $id^{\mathcal{C}}$ *is the identifier of this capability;* $sup^{\mathcal{C}}$ *is the identifier of the action from which this action inherits;* $I^{\mathcal{C}}$ *is a set of* $i_{\mathcal{C}}$ *input parameter specifications:* $\{I_1^{\mathcal{C}}, \ldots, I_{i_{\mathcal{C}}}^{\mathcal{C}}\}$*;* $O^{\mathcal{C}}$ *is a set of* $j_{\mathcal{C}}$ *output parameter specifications:* $\{O_1^{\mathcal{C}}, \ldots, O_{j_{\mathcal{C}}}^{\mathcal{C}}\}$*;* $C_I^{\mathcal{C}}$ *is a set of* $k_{\mathcal{C}}$ *input constraints:* $\{C_{I1}^{\mathcal{C}}, \ldots, C_{Ik_{\mathcal{C}}}^{\mathcal{C}}\}$*;* $C_O^{\mathcal{C}}$ *is a set of* $l_{\mathcal{C}}$ *output constraints:* $\{C_{O1}^{\mathcal{C}}, \ldots, C_{Ol_{\mathcal{C}}}^{\mathcal{C}}\}$*;* $C_{IO}^{\mathcal{C}}$ *is a set of* $m_{\mathcal{C}}$ *input-output constraints:* $\{C_{IO1}^{\mathcal{C}}, \ldots, C_{IOm_{\mathcal{C}}}^{\mathcal{C}}\}$*; and finally,* $P^{\mathcal{C}}$ *is a set of* $n_{\mathcal{C}}$ *properties of capability* $\mathcal{C}$*:* $\{P_1^{\mathcal{C}}, \ldots, P_{n_{\mathcal{C}}}^{\mathcal{C}}\}$.

The agent $A^{\mathcal{C}}$, which has the capability $\mathcal{C}$, is represented by its name as part of the capability description. The state language $S^{\mathcal{C}}$ is an instance of the class `Language`, a special resource provided by the Java Agent Template (JAT) described in section 5.3. Note that this feature of Java, the explicit representation of the class of an object as an object itself, allows the reflective reasoning over the state language within a CDL expression which is necessary to permit the plugging in of arbitrary, opaque state languages (cf. section 4.2.3). The identifier $id^{\mathcal{C}}$ of this capability can be used to refer to this description in future, and the identifier $sup^{\mathcal{C}}$ names the action of which this capability description is a specialisation (cf. section 4.3.2). Both these identifiers may be undefined.

The inputs $I^{\mathcal{C}}$ and outputs $O^{\mathcal{C}}$ are both potentially empty sets of parameter specifications, where a parameter specification consists of a role name and a term describing the object that will play this role for the described capability $\mathcal{C}$ (cf. section 4.2.4). The input constraints $C_I^{\mathcal{C}}$, the output constraints $C_O^{\mathcal{C}}$, and the input-output constraints $C_{IO}^{\mathcal{C}}$ are all sets of objects that belong to the class

specified in $S^{\mathcal{C}}$, which is, as mentioned above, a specialisation of the JAT `Language` class. Any of the sets of constraints may be empty. Finally, there is the set $P^{\mathcal{C}}$ of properties associated with this capability, as explained in section 4.4, where a property is represented by a propositional symbol.

This concludes the introduction of the internal representation used for CDL expressions and we will now turn to the problem of reasoning over CDL.

5.1.2 Capability Evaluation

We will first consider the slightly *restricted case* where capabilities are represented as achievable objectives (cf. section 4.2) and are not allowed to have input-output constraints. Task descriptions shall only contain input constraints and output constraints here. Note that these restrictions are not severe, as most other capability and task descriptions can be reduced to such a representation. How this can be done will be shown in the extensions of the basic capability evaluation algorithm that will follow in section 5.2.

5.1.2.1 Basic Capability Subsumption

The essential question the capability evaluation has to answer is whether a capability represented by the CDL description $\mathcal{C}$ can be used to solve a problem represented by the CDL description $\mathcal{T}$. We will say that capability $\mathcal{C}$ *subsumes* task $\mathcal{T}$ if this is indeed true, i.e. if the capability represented by $\mathcal{C}$ can be used to solve the problem described by $\mathcal{T}$. Now, capability $\mathcal{C}$ subsumes a task $\mathcal{T}$ if:

1. in the situation that is the result of performing $\mathcal{C}$, all the output constraints of $\mathcal{T}$ ($C_O^{\mathcal{T}}$) are satisfied, i.e. if the capability achieves the desired state; and

2. in the situation that precedes the performance of $\mathcal{C}$, all the input constraints of $\mathcal{C}$ ($C_I^{\mathcal{C}}$) are satisfied, i.e. if the capability is applicable.

We will refer to these conditions as the *output match condition* and the *input match condition* respectively. Both conditions rely on a notion of certain constraints being satisfied in a given situation, but we cannot say anything about these constraints since we do not know the state languages in which they are expressed.

To be able to formally define what we mean by a capability subsuming a task (cf. definition 5.2) we will thus make the assumption that there is a model-theoretic semantics defined for every state language we will encounter. As pointed out in [Hayes, 1974], knowledge representation languages that do not have a formal semantics do not really represent anything, and thus, we consider this assumption very reasonable. Assuming that there exists a model-theoretic semantics, we can easily associate models with situations and we have defined constraints as expressions in the state language. Thus, we can define that *a constraint is satisfied in a situation* if the model corresponding to the situation is a model of the expression representing the constraint.

The next problem is that *situations* are never explicitly mentioned in the capability or in the task description. However, a model-theoretic semantics essentially defines a mapping from expressions in a given state language into the power set of models for this language. An expression in the language is mapped to the set of all models in which this expression is considered true. Thus, the input constraints $C_I^{\mathcal{T}}$ of the task $\mathcal{T}$ define a set of models, one of which corresponds to the actual situation before the capability is to be applied. The input match condition is obviously satisfied if every model of the task's input constraints $C_I^{\mathcal{T}}$ is also a model of the capability's input constraints $C_I^{\mathcal{C}}$. Similarly, the output constraints $C_O^{\mathcal{T}}$ of the task $\mathcal{T}$ define a set of models, all of which correspond to situations in which the objective has been achieved, and the output match condition is obviously satisfied if every model of $C_O^{\mathcal{C}}$ is also a model of $C_O^{\mathcal{T}}$.

For simplicity, one can define the *meta-relation* $\models$ between expressions as the subset relation of the models of the related expressions. This meta-relation can

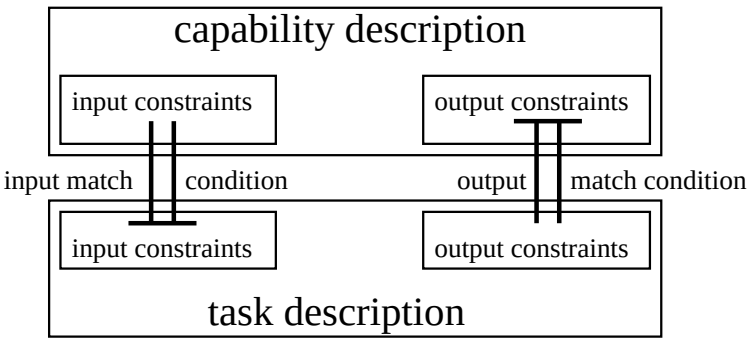

Figure 5.1: Matching Capabilities and Tasks

now be used to define subsumption as illustrated in figure 5.1 graphically. Notice that this meta-relation, defined in this way, also relates expressions in different languages as long as they have a model-theoretic semantics. However, the meta-relation $\models$ has to be defined for each state language individually. Most of the capability descriptions given as examples in this thesis use first-order predicate logic (FOPL) as the state language and a definition of the meta-relation $\models$ for FOPL will follow in section 7.3.3.

We are now in a position to formally define when a restricted capability description subsumes a task description in CDL:

Definition 5.2 (Subsumption for Achievable Objectives (1)) *Let $\mathcal{C}$ be a capability description in* CDL *containing: an input specification $I^{\mathcal{C}}$ containing the variables $v_1, \ldots, v_h$; input constraints $C_I^{\mathcal{C}} = \{C_{I1}^{\mathcal{C}}, \ldots, C_{Ikc}^{\mathcal{C}}\}$; and output constraints $C_O^{\mathcal{C}} = \{C_{O1}^{\mathcal{C}}, \ldots, C_{Olc}^{\mathcal{C}}\}$. Let $\mathcal{T}$ be a task description in* CDL *containing: input constraints $C_I^{\mathcal{T}} = \{C_{I1}^{\mathcal{T}}, \ldots, C_{Ik_{\mathcal{T}}}^{\mathcal{T}}\}$; and output constraints $C_O^{\mathcal{T}} = \{C_{O1}^{\mathcal{T}}, \ldots, C_{Ol_{\mathcal{T}}}^{\mathcal{T}}\}$. We will say that $\mathcal{C}$* **subsumes** *$\mathcal{T}$ if and only if there exists a substitution σ for the variables $v_1, \ldots, v_h$ such that:*

$$C_I^{\mathcal{T}} \models \sigma(C_I^{\mathcal{C}}) \qquad \textit{(input match condition)}$$

and

$$\sigma(C_O^{\mathcal{C}}) \models C_O^{\mathcal{T}} \qquad \textit{(output match condition)}$$

Perhaps a little surprising is the *requirement of the substitution* σ in this definition. The substitution σ accounts for the fact that capability descriptions in CDL are parametrised, i.e. they contain free variables in the input constraints and in the output constraints. These variables must be declared in the input and output specification of the capability, effectively rendering the capability description an action schema. For the actual performance of the capability $\mathcal{C}$ these variables must be instantiated, and this is what the substitution σ allows for. However, the definition does not require the variables to be mapped to ground terms by the substitution. This is to allow for reasoning about partially instantiated capabilities, i.e. this feature can be used to extend the above definition to capabilities subsuming other capabilities rather than tasks.

Perhaps also surprising is the fact that *only variables occurring in the input specification* $(v_1, \ldots, v_h)$ need to be substituted in the above definition. This is because variables occurring in the output specification only play a role in the output match condition. However, one could easily extend the above definition to also require the output specification to unify with all the corresponding outputs in the task description. It is not clear though what the benefit of this would be, since outputs in task and capability description will usually be variables only. As it stands, the output specification allows one to introduce additional free variables into the output constraints of the capability description.

One of the most important features of this definition is the fact that is *does not mention which state language is to be used* in CDL. It only requires certain types of reasoning to be performable in the language $S^{\mathcal{C}}$: firstly, one must be able to build conjunctions of expressions in this language and secondly, the meta-relation $\models$ must be defined in the state language used. Virtually all knowledge representation languages have conjunctions built in since a sequence of assertions is usually interpreted as the conjunction of the asserted expressions. The meta-relation $\models$ should also be defined as part of the semantics of the language as argued above.

```
outKB ← new KnowledgeBase(S^C)
for c ∈ {C^C_O1, ..., C^C_Olc} do
        assert(outKB, c)
σ ← evaluate(outKB, C^T_O1 ∧ ... ∧ C^T_Olτ, {v_1, ..., v_h})
if σ is undefined then
        return false

inKB ← new KnowledgeBase(S^C)
for c ∈ {C^T_I1, ..., C^T_Ilτ} do
        assert(inKB, c)
for c ∈ {C^C_I1, ..., C^C_Ilc} do
        if not evaluate(inKB, σ(c)) then
                return false

return true
```

Figure 5.2: Subsumption algorithm (1)

5.1.2.2 The Basic Algorithm

The basic algorithm used to evaluate capability subsumption is a straightforward implementation of definition 5.2 above. The pseudo-code version of this algorithm is given in figure 5.2.

The algorithm first attempts to create an empty knowledge base, *outKB*, for expressions in the state language $S^{\mathcal{C}}$. This is also the first point where reflective reasoning is necessary. If it cannot be decided at this point which the appropriate knowledge base class for the state language $S^{\mathcal{C}}$ is, or the creation of an empty knowledge base of this type fails for any other reason, the capability will not be considered to subsume the task. Note that this is basically the mechanism for all function calls that involve reflective reasoning. The underlying assumption we are making here is that a certain type of reasoning is needed for the capability subsumption test and if this type of reasoning is not supported by $S^{\mathcal{C}}$ then the test has failed.

The next step in the algorithm asserts all the output constraints $C^{\mathcal{C}}_{O1}, \ldots, C^{\mathcal{C}}_{Olc}$

of capability $\mathcal{C}$ in the knowledge base *outKB*. It is assumed that every knowledge base provides assertion in its functionality and thus, this step does not require reflective reasoning to test for the existence of this functionality. The next step is to evaluate the conjunction of the output constraints $C^{\mathcal{T}}_{O1}, \ldots, C^{\mathcal{T}}_{Ol_{\mathcal{T}}}$ of task $\mathcal{T}$. This step requires reflection again as it is not guaranteed that the required function is defined. If it is defined it will attempt to derive the query (second argument) from the knowledge base (first argument). If this succeeds it will return a substitution for the variables $\{v_1, \ldots, v_h\}$ (third argument) as they need to be instantiated for the derivation. The next step in the algorithm tests whether the returned substitution σ is defined, i.e. whether such a substitution could be found; if not, the test will fail.

The algorithm up to this point implements essentially the output match condition. Assuming that the function *evaluate* used by the algorithm implements the desired behaviour, we know that for the substitution σ: $\sigma(C^{\mathcal{C}}_{O1}) \wedge \ldots \wedge \sigma(C^{\mathcal{C}}_{Ol_{\mathcal{C}}}) \models C^{\mathcal{T}}_{O1} \wedge \ldots \wedge C^{\mathcal{T}}_{Ol_{\mathcal{T}}}$ because this is exactly the substitution we have extracted from the derivation.

The remainder of the algorithm implements the input match condition and is quite similar to the test for the output match condition. First an empty knowledge base, *inKB*, for expressions in $S^{\mathcal{C}}$ is created. Next the input constraints $C^{\mathcal{T}}_{I1}, \ldots, C^{\mathcal{T}}_{Il_{\mathcal{T}}}$ from task $\mathcal{T}$ are asserted in this knowledge base. Finally, the input constraints $C^{\mathcal{C}}_{I1}, \ldots, C^{\mathcal{C}}_{Il_{\mathcal{C}}}$ of capability $\mathcal{C}$ are evaluated against *inKB*. However, as opposed to the output match condition, the constraints are evaluated one by one. The reason for this is that it simplifies the code slightly. Before the evaluation the constraints have to be instantiated with the substitution σ to reflect the input match condition. If there is an instantiated input constraint $\sigma(C^{\mathcal{C}}_{In})$ for $n \in \{1 \ldots l_{\mathcal{C}}\}$ that cannot be derived from *inKB* then the subsumption test has failed.

Otherwise it succeeded and the capability $\mathcal{C}$ can be used to solve the problem described by $\mathcal{T}$.

Soundness and Completeness It is fairly easy to see that this algorithm is sound, assuming the soundness of the evaluation procedure for the knowledge base for expressions in state language $S^{\mathcal{C}}$, i.e. that if the algorithm returns true then the capability $\mathcal{C}$ subsumes task $\mathcal{T}$ as outlined in definition 5.2. However, it is not complete because it does not backtrack over the substitution σ. If the input match condition part of the algorithm fails it might be possible to attempt a different derivation leading to a different substitution for the output match condition, etc. We have chosen not to implement this option for two reasons: firstly, it increases the complexity of the algorithm without benefit in the scenarios we envisage, and secondly, it still requires the completeness of the evaluation function to make this algorithm complete, which is not the case in our implementation. Furthermore, in some state languages the substitution is unique if one exists and thus, backtracking over the substitution would be superfluous.

5.1.2.3 An Example from the Initial Scenario

It is now time to look at an example illustrating the above definitions and the algorithm. The capability and task descriptions in the initial scenario all satisfy the restrictions introduced above, i.e. that capabilities are represented as achievable objectives without input-output constraints and that tasks only consist of input constraints and output constraints. For example, the capability advertised by the engineering company was:

```
(capability
  :state-language fopl
  :input ((BrokenMachine ?machine))
  :input-constraints (
    (elt ?machine Generator)
    (Is ?machine Broken)
    (Has Location ?machine Pacifica))
  :output-constraints (
    (not (Is ?machine Broken))))
```

The engineering part of the **pp**-agent's problem which this capability description must match was described as follows:

```
(task
  :state-language fopl
  :input-constraints (
    (elt generator1 Generator)
    (Is generator1 Broken)
    (Has Location generator1 Pacifica))
  :output-constraints (
    (not (Is generator1 Broken))))
```

To test whether the **ec**-agent's capability subsumes the described task, the algorithm in figure 5.2 will first create a knowledge base (*outKB*) containing the output constraints of the **ec**-agent's capability:

```
(NOT (Is ?machine_3 Broken))
```

In this example *outKB* contains just this one constraint. The index of the variable `?machine_3` is part of the internal representation of the broker again (cf. section 4.5.1). In the next step the conjunction of all the output constraints of the task is evaluated against *outKB* to obtain the substitution σ:

```
(NOT (Is generator1 Broken))
```

As there is just one output constraint, the expression contains just this one constraint. The third parameter in the call to *evaluate* is the set of variables in the input parameter specification of the **ec**-agent's capability:

```
[?machine_3]
```

The variable `?machine_3` is the only variable in the input specification in this example. The call to *evaluate* succeeds and returns the substitution:

```
[generator1->[?machine_3]]
```

The test whether this substitution is defined succeeds and completes the output match condition, i.e. we have now established that under substitution σ:

$$\texttt{(not (Is generator1 Broken))} \models \texttt{(not (Is generator1 Broken))}$$

The next step in our algorithm generates the knowledge base for testing the input match condition (*inKB*) and initialises it with the input constraints from the task:

```
(elt generator1 Generator)
(Is generator1 Broken)
(Has Location generator1 Pacifica)
```

Next the input constraints from the **ec**-agent's capability description are one by one instantiated with the substitution σ and evaluated against *inKB*:

```
(elt generator1 Generator)
(Is generator1 Broken)
(Has Location generator1 Pacifica)
```

Since none of the evaluations fails all the above constraints will be tested and, as a result, the input match condition under σ is established:

$$\left(\begin{array}{c}\texttt{(elt generator1 Generator)}\wedge\\ \texttt{(Is generator1 Broken)}\wedge\\ \texttt{(Has Location generator1 Pacifica)}\end{array}\right)\models\left(\begin{array}{c}\texttt{(elt generator1 Generator)}\wedge\\ \texttt{(Is generator1 Broken)}\wedge\\ \texttt{(Has Location generator1 Pacifica)}\end{array}\right)$$

With output and input match condition successfully verified, we know now that the **ec**-agent's capability subsumes the described task and the algorithm returns `true`.

5.2 Extended Capability Evaluation

In this section we will show how more complex capability descriptions containing input-output constraints, performable actions, and properties can be evaluated against task descriptions in CDL.

5.2.1 Input-Output Constraints

In this section we will consider capabilities with input-output constraints (cf. section 4.2.2), i.e. capabilities which are represented as achievable objectives and task descriptions which shall only contain input constraints and output constraints.

5.2.1.1 Subsumption with Input-Output Constraints

The question the capability evaluation has to answer is still whether a capability represented by CDL description $\mathcal{C}$ can be used to solve a problem represented by CDL description $\mathcal{T}$. In section 5.1.2.1 we have used the input and output match conditions to define what it means for capability $\mathcal{C}$ to subsume task $\mathcal{T}$. Essentially, these conditions require the input constraints to be satisfied in the input situation and the output constraints to be satisfied in the output situation, i.e. we had to evaluate *constraints on situations* to test for subsumption.

Ideally, we could just extend the input and output match conditions to account for the input-output constraints. However, whereas the input constraints and the output constraints are constraints on situations, the *input-output constraints are constraints across situations*, i.e. they are fundamentally different from the constraints discussed in section 5.1.2. There we could define a constraint to be satisfied in a situation if the model corresponding to the situation was a model of the expression representing the constraint. Unfortunately this approach is not applicable here.

To keep the definition of the subsumption relation independent from the state language used within the capability description, we want to retain the approach

of using the model-theoretic semantics of the state language to define the subsumption relation. However, input-output constraints potentially contain variables from the output specification which represent objects that only exist in the output situation. Hence, any model of the input situation will not mention properties or relations involving these objects. If we interpret models in the usual way, i.e. anything not mentioned is false, then the satisfiability of an input-output constraint may not depend on the relations mentioned in it. This is not what we want. To address this problem we would need to be able to *distinguish the parts* of the input-output constraint that refer to the input situation from the parts that refer to the output situation.

To illustrate this problem, let us revisit the *list sorting capability* $\mathcal{L}$ from section 4.2.2. If `?list1` represents the list to be sorted in the input specification $I^{\mathcal{L}}$ and `?list2` represents the sorted list in the output specification $O^{\mathcal{L}}$ then one input-output constraint we need to express in $C^{\mathcal{L}}_{IO}$ is:

```
((forall ?x) (implies (member ?x ?list1) (member ?x ?list2)))
```

This suggests that literals containing variables from $I^{\mathcal{L}}$ refer to the input situation and literals containing variables from $O^{\mathcal{L}}$ refer to the output situation, but this is not the case in general. In fact, if the sorting capability $\mathcal{L}$ sorted by modifying the given list, it becomes obvious that the above constraint cannot be interpreted as intended without further assumptions. One solution would be to annotate parts of the constraint with the situation they refer to, but this is against the spirit of a decoupled action representation with an opaque state language.

The approach we have chosen in CDL assumes that every input-output constraint consists of *two parts which are connected by an implication.* The left hand side or precondition will be interpreted as a constraint on the input situation and the right hand side or conclusion will be interpreted as a constraint on the output situation. Note that this is essentially also the way secondary effects are implemented in UCPOP [Penberthy and Weld, 1992, Barrett *et al.*, 1995]. The advantage of this approach is that it allows us to define capability subsumption in terms

of models again. The disadvantage of this approach is that every input-output constraint has to be expressed as an implication with the two sub-expressions referring to input and output situation respectively. With this approach we can define capability subsumption as follows:

Definition 5.3 (Subsumption for Achievable Objectives (2)) *Let $\mathcal{C}$ be a capability description in* CDL *containing: an input specification $I^{\mathcal{C}}$ containing the variables $v_1, \ldots, v_h$; input constraints $C_I^{\mathcal{C}} = \{C_{I1}^{\mathcal{C}}, \ldots, C_{Ik_{\mathcal{C}}}^{\mathcal{C}}\}$; output constraints $C_O^{\mathcal{C}} = \{C_{O1}^{\mathcal{C}}, \ldots, C_{Ol_{\mathcal{C}}}^{\mathcal{C}}\}$; and input-output constraints $C_{IO}^{\mathcal{C}} = \{C_{IO1}^{\mathcal{C}}, \ldots, C_{IOm_{\mathcal{C}}}^{\mathcal{C}}\}$, each of which having the form $L_n^{\mathcal{C}} \rightarrow R_n^{\mathcal{C}}$ for $n \in \{1 \ldots m_{\mathcal{C}}\}$. Let $\mathcal{T}$ be a task description in* CDL *containing: input constraints $C_I^{\mathcal{T}} = \{C_{I1}^{\mathcal{T}}, \ldots, C_{Ik_{\mathcal{T}}}^{\mathcal{T}}\}$; and output constraints $C_O^{\mathcal{T}} = \{C_{O1}^{\mathcal{T}}, \ldots, C_{Ol_{\mathcal{T}}}^{\mathcal{T}}\}$. We will say that $\mathcal{C}$* **subsumes** *$\mathcal{T}$ if and only if there exists a substitution σ for the variables $v_1, \ldots, v_h$ such that:*

$$C_I^{\mathcal{T}} \models \sigma(C_I^{\mathcal{C}}) \qquad \textit{(input match condition)}$$

and

$$\sigma(C_O^{\mathcal{C}}) \wedge \sigma(R^{\mathcal{C}}) \models C_O^{\mathcal{T}} \qquad \textit{(output match condition)}$$

and

$$\forall n \in \{1 \ldots m_{\mathcal{C}}\} : \textit{if } \sigma(C_O^{\mathcal{C}}) \wedge (\sigma(R^{\mathcal{C}}) - \sigma(R_n^{\mathcal{C}})) \not\models C_O^{\mathcal{T}} \textit{ and } \sigma(C_O^{\mathcal{C}}) \wedge \sigma(R^{\mathcal{C}}) \models C_O^{\mathcal{T}} \textit{ then } C_I^{\mathcal{T}} \models \sigma(L_n^{\mathcal{C}}) \qquad \textit{(input-output match condition)}$$

The input match condition in this definition stays unchanged from definition 5.2: the task's input constraints have to make all of the capability's input constraints true. The output match condition is changed to reflect that there are now additional constraints on the output situation described in the capability: the capability's output constraints in conjunction with the right hand sides of all the input-output constraints have to make the task's output constraints true. The third condition, the input-output match condition, is new here. Essentially it says that, if the right hand side of the nth input-output constraint was necessary to satisfy the output match condition, then the task's input constraints also have to make the left hand side of the nth input-output constraint true.

for $c \in \{C^{\mathcal{C}}_{IO1}, \ldots, C^{\mathcal{C}}_{IOm_{\mathcal{C}}}\}$ **do**
    $assert(outKB,\ conclusionOf(c))$

Figure 5.3: Subsumption algorithm (2)

for $c \in \{C^{\mathcal{C}}_{IO1}, \ldots, C^{\mathcal{C}}_{IOm_{\mathcal{C}}}\}$ **do**
    if $usedInProof(outKB,\ conclusionOf(c))$ **then**
        if not $evaluate(inKB,\ \sigma(premiseOf(c)))$ **then**
            return false

Figure 5.4: Subsumption algorithm (3)

5.2.1.2 The Algorithm

We will now present the modifications to the algorithm described in figure 5.2 that are necessary to test for the extended conditions of the subsumption relation defined above (definition 5.3).[1] As the input match condition does not change, no modification of the algorithm is necessary for this condition. The extended output match condition can be implemented by asserting the right hand sides of all the input-output constraints into *outKB* before the task's output constraints are evaluated. The additional pseudo-code that implements these assertions is given in figure 5.3.

Note that this extension requires reflective reasoning again to extract the right hand side or conclusion from an input-output constraint. Of course, this will only be possible if the state language allows the representation of implications and provides a function to extract the right hand side from the implication. If extraction of the conclusion fails, the capability will be considered not appropriate for the task and return `false`.

[1] The complete pseudo-code for the capability subsumption test including all the extensions presented in this section will be given in section 5.2.4.

The input-output match condition from definition 5.3 can be implemented as described by the pseudo-code in figure 5.4. This code has to be added after the initialisation of *inKB*.

For efficiency reasons this is not a straight forward implementation of the definition of the input-output match condition. The function assumes that the proof that was generated to verify the output match condition is somehow retained in *outKB*. It also assumes there to be a function that can inspect this proof to find out whether a given expression that has been asserted in *outKB* has actually been used in the proof. The function *usedInProof*(*outKB*, *conclusionOf*(*c*)) in figure 5.4 tests for each input-output constraint whether its conclusion has been used in the proof. If so, the left hand side is evaluated against *inKB* to find whether it can be satisfied. If it cannot be satisfied, the capability does not subsume the task.

As before we do not backtrack over the substitution generated from the output match condition. Our implementation also does not extend the set of variables in the substitution to allow for additional free variables in the input-output constraints. Such variables are allowed in UCPOP's action representation in the form of a possible universal quantification over each input-output constraints. However, the domains of these variables must be declared in the representation and these domains must be finite and all elements must be known, effectively reducing the expressiveness to the ground case again.

5.2.1.3 An Example

None of the original scenarios described in chapter 3 requires input-output constraints in the representation. Therefore, to illustrate the extension of the algorithm for this feature, we will describe a slightly modified capability for a hospital from the initial scenario here.

The capability of the new hospital has input constraints identical to the two hospitals in the initial scenario: the given parameter `?person` must be a person;

that person must be injured; and the location of this injured person must be on Pacifica. The sole output constraint, too, is identical to that of the hospitals in the initial scenario: the given person will no longer be injured. The only new condition here is the input-output constraint that states that if the injury is severe then the injured person will have to go to hospital:

```
(capability
  :state-language fopl
  :input ((InjuredPerson ?person))
  :input-constraints (
    (elt ?person Person)
    (Is ?person Injured)
    (Has Location ?person Pacifica))
  :output-constraints (
    (not (Is ?person Injured)))
  :io-constraints (
    (implies
      (Is Injury Severe) (Has Location ?person Hospital))))
```

Note that the left hand side of this implication, `(Is Injury Severe)`, is a constraint on the input situation and the right hand side, `(Has Location ?person Hospital)`, is a constraint on the output situation.

Next we will need a problem that requires the input-output constraint of this capability:

```
(task
  :state-language fopl
  :input-constraints (
    (elt JohnSmith Person)
    (Is JohnSmith Injured)
    (Has Location JohnSmith Pacifica)
    (Is Injury Severe))
  :output-constraints (
    (not (Is JohnSmith Injured))
    (Has Location JohnSmith Hospital)))
```

In this problem description the person John Smith is injured, on Pacifica, and most importantly, the injury is severe. In the state desired by the problem holder John Smith should no longer be injured and he should be in hospital.

To test whether the above capability subsumes the described problem, the algorithm first creates an appropriate knowledge base for expressions in the state language (cf. figure 5.2). Next the output constraints of the capability will be asserted in this knowledge base. Now the right hand sides of the input-output constraints will also be asserted (cf. figure 5.3) before any evaluation takes place. The resulting knowledge base looks as follows:

```
(NOT (Is ?person_3 Injured))
(Has Location ?person_3 Hospital)
```

The call to *evaluate* for the output match condition takes three parameters (cf. figure 5.2). The first parameter is the knowledge base. The second parameter is the query, the conjunction of the output constraints of the task, in this example:

```
(AND (NOT (Is JohnSmith Injured)) (Has Location JohnSmith Hospital))
```

The third and final parameter is the list of variables from the input specification of the capability:

```
[?person_3]
```

In this example, the call to *evaluate* succeeds and returns the following substitution σ:

```
[JohnSmith->[?person_3]]
```

As this substitution is defined, the output match condition has now succeeded. The next step in the algorithm is to test the input match condition. This begins with the construction of a new knowledge base and initialising it with the task's input constraints (cf. figure 5.2). In this example this knowledge base will contain the following statements:

```
(elt JohnSmith Person)
(Is JohnSmith Injured)
(Has Location JohnSmith Pacifica)
(Is Injury Severe)
```

Now the input match condition can be tested by evaluating all of the capability's input constraints against this knowledge base. These constraints are:

```
(elt JohnSmith Person)
(Is JohnSmith Injured)
(Has Location JohnSmith Pacifica)
```

Since none of these evaluations fails, the input match condition succeeds. What remains to be tested is the input-output match condition. In this example there is only one input-output constraint, so the loop in figure 5.4 will only have one iteration. In this, the algorithm will first test whether the conclusion of the input-output constraint has been used in the proof for the output match condition. The conclusion is:

```
(Has Location ?person_3 Hospital)
```

Since this statement was necessary to satisfy the output match condition the algorithm will continue to evaluate the left hand side of the input output constraint against the knowledge base used for the evaluation of the input match condition. The instantiated query for this example is:

```
(Is Injury Severe)
```

This evaluation will succeed and since this was the only input-output constraint to be tested and this condition concludes the capability subsumption test in definition 5.3, the algorithm will return `true`.

5.2.2 Properties

The next extension to the subsumption test is concerned with the *properties* of agents described in section 4.4. The representation of such properties is quite simple in CDL as it only allows for a list of propositional symbols. In a capability description, these propositions are interpreted as true for the capability holding agent, and in a task description they are interpreted as propositions required

if $P^{\mathcal{T}} \not\subseteq P^{\mathcal{C}}$ **then**
return false

Figure 5.5: Subsumption algorithm (4)

to be true for the sought for agent. Thus, in both cases they are interpreted as conjunctions of propositions and the extension of definition 5.3 for capability subsumption is straight forward:

Definition 5.4 (Subsumption for Achievable Objectives (3)) *Let $\mathcal{C}$ be a capability description in* CDL *containing an input specification $I^{\mathcal{C}}$, input constraints $C_I^{\mathcal{C}}$, output constraints $C_O^{\mathcal{C}}$, input-output constraints $C_{IO}^{\mathcal{C}}$, and property specifications $P^{\mathcal{C}}$. Let $\mathcal{T}$ be a task description in* CDL *containing input constraints $C_I^{\mathcal{T}}$,output constraints $C_O^{\mathcal{T}}$, and property specifications $P^{\mathcal{T}}$. We will say that $\mathcal{C}$* **subsumes** *$\mathcal{T}$ if and only if $\mathcal{C}$ subsumes $\mathcal{T}$ (definition 5.3) and $P^{\mathcal{C}} \models P^{\mathcal{T}}$.*

The relation $\models$ in the new condition of this definition is the usual relation for propositional logic. Thus, the *algorithm* that tests for capability subsumption can be extended with the pseudo-code described in figure 5.5 to account for properties of agents. This pseudo-code can be added at the very beginning of the algorithm described this far.

Since property lists in capability and task descriptions represent conjunctions of properties, the test can be reduced to a subset test at this point.

For example, in section 4.4 we have described an additional capability for the **h2**-agent in the initial scenario: the **h2**-agent can move patients to the hospital. As opposed to the capability to treat patients, the **h2**-agent advertised its moving capability as complete by specifying the properties:

```
:properties (complete)
```

The problem description in section 4.4 also contains this property specification in its description, i.e. it can only be addressed by a PSA with a complete problem-solving behaviour. Obviously, the two property specifications match.

5.2.3 Performable Actions

In this section we will look at performable actions as described in section 4.3 and how these can be integrated into the framework.

5.2.3.1 Capabilities as Performable Tasks

Our aim is to design an algorithm that decides whether a capability represented by CDL description $\mathcal{C}$ can be used to solve a problem represented by CDL description $\mathcal{T}$. Up to now we have assumed that the capability as well as the task are described in terms of achievable objectives (cf. section 4.2). Now we also want to allow for capabilities or tasks to be described in terms of performable actions (cf. section 4.3). The most complex and interesting cases here are the ones in which the *representation is mixed*, i.e. where a capability is described in one way, e.g. as a performable action, and the task is described in the other way, e.g. as an objective to be achieved.

In section 4.3.1 we have already mentioned one *example* where this mixing of representations would be useful: suppose that in the initial scenario, the **pp**-agent specified the engineering part of the problem not as an objective to be achieved, i.e. the generator must not be broken, but as an action to be performed, i.e. repairing the generator. In this modified example the broker would need to be able to match the capability described by an achievable objective to a problem described as an action to be performed.

An essential insight here is that even if the problem is specified as an action to be performed, the *underlying problem is normally an objective* to be achieved. For example, given the modified problem above which is described as an action of type repairing by the **pp**-agent, the intended aim of this action is surely not to

find another agent that can perform a repairing action, but to have its generator in a state where it is working as intended. Thus, a problem description that is an action to be performed need not necessarily be taken literally.

Furthermore, if, for example, the generator was not broken but had run out of fuel, and the problem was described as a repairing action to be performed, we would surely expect any reasonable PSA not to perform a repairing action but a refuelling action. Similarly, if the problem was a broken gasket then the action we want the PSA to perform is a repairing of the generator, but it was not specified as a replacing of the gasket. Repairing and replacing are different actions that, in general, do not subsume each other. Thus, even if problem and capability are described as performable actions, it is not necessarily the case that an action addressing the problem is necessarily of the type described in the problem.

Thus, our approach to reasoning about capabilities and tasks represented as performable actions will be to *instantiate* these representations into equivalent specifications of achievable objectives by inheriting the parameter specifications and constraints from a description of the action in an ontology and modifying them, and then test for capability subsumption as described above.[2] As described in section 4.3.2, there are three ways in which an inheriting capability description can modify the capability it inherits from: it can bind parameters to values, it can add new parameters, and it can add new constraints. To treat newly bound parameters we will need the following definition:

Definition 5.5 (Parameter-unifying substitution) *Let $PS^{\mathcal{C}_1}$ and $PS^{\mathcal{C}_2}$ be parameter specifications of capabilities $\mathcal{C}_1$ and $\mathcal{C}_2$, i.e. they can be either input or output specifications. Let each parameter specification have the form $\prec R, F \succ$*[3]

[2] Note that this instantiation may lead to inappropriate behaviour of the PSA, but so may just performing the specified action. The underlying problem here is that communication assumes a shared model but there is no reasonable way to ensure that this is indeed the case. The approach we would suggest is to equip the communicating agent with commonsense knowledge and user modelling facilities to detect misunderstandings, but this is beyond the scope of this thesis.

[3] cf. `<param-spec>` in figure 4.1 and its explanation in section 4.2.4

where R is a role name and F is a term that describes the role filler. A substitution σ is a **parameter-unifying substitution** *for $PS^{\mathcal{C}_1}$ and $PS^{\mathcal{C}_2}$ if and only if:*

$$\forall R : (\prec R, F^{\mathcal{C}_1} \succ \in PS^{\mathcal{C}_1} \wedge \prec R, F^{\mathcal{C}_2} \succ \in PS^{\mathcal{C}_2}) \Rightarrow \sigma(F^{\mathcal{C}_1}) = \sigma(F^{\mathcal{C}_2})$$

Essentially, this defines a parameter-unifying substitution between two parameter specifications as one in which every role unifies with all those terms that are role fillers for this role. Our intention is to use a parameter-unifying substitution between an action's parameter specification and its super-action's parameter specification to instantiate the action. The following definition formalises this notion:

Definition 5.6 (Capability instantiation) *Let $\mathcal{C}$ be a capability description of a performable action in* CDL. *Let σ be a parameter-unifying substitution for $I^{\mathcal{C}}$ and $I^{sup^{\mathcal{C}}}$ as well as for $O^{\mathcal{C}}$ and $O^{sup^{\mathcal{C}}}$. Then the capability description $\mathcal{C}'$ is the* **instantiation of** *$\mathcal{C}$ if and only if $A^{\mathcal{C}'} = A^{\mathcal{C}}$, $S^{\mathcal{C}'} = S^{\mathcal{C}}$, $id^{\mathcal{C}'}$ and $sup^{\mathcal{C}'}$ are undefined, $I^{\mathcal{C}'} = \sigma(I^{\mathcal{C}}) \cup \sigma(I^{sup^{\mathcal{C}}})$, $O^{\mathcal{C}'} = \sigma(O^{\mathcal{C}}) \cup \sigma(O^{sup^{\mathcal{C}}})$, $C_I^{\mathcal{C}'} = \sigma(C_I^{\mathcal{C}}) \cup \sigma(C_I^{sup^{\mathcal{C}}})$, $C_O^{\mathcal{C}'} = \sigma(C_O^{\mathcal{C}}) \cup \sigma(C_O^{sup^{\mathcal{C}}})$, and finally $C_{IO}^{\mathcal{C}'} = \sigma(C_{IO}^{\mathcal{C}}) \cup \sigma(C_{IO}^{sup^{\mathcal{C}}})$.*

In this definition, the identifier of the instantiated capability $id^{\mathcal{C}'}$ is undefined because its CDL description will only be generated for the subsumption test and is not available subsequently. The identifier of the super-action $sup^{\mathcal{C}'}$ must be undefined because the CDL description $\mathcal{C}'$ is no longer a modification description of the action $sup^{\mathcal{C}}$. The input and output parameter specifications of $\mathcal{C}'$ are the unions of the respective parameter specifications of $\mathcal{C}$ and its super-action $sup^{\mathcal{C}}$, instantiated with the parameter-unifying substitution. By the instantiation of a parameter specification $\prec R, F \succ$ under substitution σ we mean the parameter specification $\prec R, \sigma(F) \succ$. Note that, since $I^{\mathcal{C}'}$ and $O^{\mathcal{C}'}$ are sets, and since parameter specifications that occur in $I^{\mathcal{C}}$ and $I^{sup^{\mathcal{C}}}$, or $O^{\mathcal{C}}$ and $O^{sup^{\mathcal{C}}}$, with the same role name will be instantiated to the same parameter specification under the parameter-unifying substitution σ, each role name can occur only once in $I^{\mathcal{C}'}$ and

```
if sup^C is defined then
        return subsumes(instantiate(C), T)
if sup^T is defined then
        return subsumes(C, instantiate(T))
```

Figure 5.6: Subsumption algorithm (5)

$O^{\mathcal{C}'}$. Finally, the various constraints of $\mathcal{C}'$ are simply the union of the respective instantiated constraints of $\mathcal{C}$ and $sup^{\mathcal{C}}$ under the substitution σ.

5.2.3.2 The Instantiation Algorithm

Now we have to incorporate capability instantiation into the existing algorithm. This will be done by instantiating the given capability and task before the usual subsumption test is performed. For this purpose, the pseudo-code in figure 5.6 has to be inserted at the very beginning of the subsumption test, even before the test for the properties.

This pseudo-code calls the function *instantiate* which instantiates the given CDL description as shown above. The pseudo-code for this function is given in figure 5.7.

This algorithm first tests whether the super-action $sup^{\mathcal{C}}$ of the given capability description is defined. If this is not the case, i.e. if the given capability is already described in terms of achievable objectives, a copy of the given capability will be returned. Otherwise a new capability description is initialised with the instantiated super-action of the given capability. This part of the algorithm deals with actions that inherit from actions which are themselves described as a performable action.

Now, at this point $\mathcal{C}'$ is a copy of the capability description of the super-action $sup^{\mathcal{C}}$ described in terms of achievable objectives. The algorithm now modifies $\mathcal{C}'$ to obtain the instantiation of $\mathcal{C}$. First, the capability holder is set to $A^{\mathcal{C}}$, the

function CDL-description *instantiate*($\mathcal{C}$)

if $sup^{\mathcal{C}}$ *is undefined* **then**
return *copy*($\mathcal{C}$)
$\mathcal{C}' \leftarrow$ *instantiate*($sup^{\mathcal{C}}$)

$A^{\mathcal{C}'} \leftarrow A^{\mathcal{C}}$
$sup^{\mathcal{C}'}, id^{\mathcal{C}'} \leftarrow$ *undefined*
$P^{\mathcal{C}'} \leftarrow P^{\mathcal{C}'} \cup P^{\mathcal{C}}$

$\sigma \leftarrow$ *empty substitution*
amend($I^{\mathcal{C}'}, I^{\mathcal{C}}, \sigma$)
amend($O^{\mathcal{C}'}, O^{\mathcal{C}}, \sigma$)
$C_I^{\mathcal{C}'} \leftarrow \sigma(C_I^{\mathcal{C}'}) \cup \sigma(C_I^{\mathcal{C}})$
$C_O^{\mathcal{C}'} \leftarrow \sigma(C_O^{\mathcal{C}'}) \cup \sigma(C_O^{\mathcal{C}})$
$C_{IO}^{\mathcal{C}'} \leftarrow \sigma(C_{IO}^{\mathcal{C}'}) \cup \sigma(C_{IO}^{\mathcal{C}})$

return $\mathcal{C}'$

Figure 5.7: Capability instantiation

```
function amend(PS^C', PS^C, σ)

    for ≺ R, F^C ≻ ∈ PS^C
        if ∃ ≺ R, F^C' ≻ ∈ PS^C' then
            σ ← unify(F^C, F^C', σ)
            PS^C' ← PS^C' − ≺ R, FC' ≻
            PS^C' ← PS^C' + ≺ R, σ(F^C) ≻
        else
            PS^C' ← PS^C' + ≺ R, F^C ≻
```

Figure 5.8: Amending parameter specifications

capability holder of $\mathcal{C}$. Next the action identifier and the super-action of $\mathcal{C}'$ are declared undefined. Then the properties of $\mathcal{C}$ are added to the properties already in $P^{\mathcal{C}'}$. Note how this part of the algorithm almost exactly mirrors the corresponding part of definition 5.6.

The next part of the algorithm amends the parameter specifications and adds the new constraints. First, an empty substitution σ is created. This substitution will be modified to become the parameter-unifying substitution mentioned in definition 5.6. This will be done in the function *amend* which takes two sets of parameter specifications, $PS^{\mathcal{C}'}$ and $PS^{\mathcal{C}}$, and a substitution σ as arguments. The substitution will be modified to become a parameter-unifying substitution for $PS^{\mathcal{C}'}$ and $PS^{\mathcal{C}}$ and the first set of parameter specifications $PS^{\mathcal{C}'}$ will be modified to become $\sigma(PS^{\mathcal{C}'}) \cup \sigma(PS^{\mathcal{C}})$. After calling *amend* for the input and output parameter specifications the substitution σ will be the parameter-unifying substitution. Now this substitution can be used to instantiate and add all the constraints to the respective sets. Finally, the capability description $\mathcal{C}'$ represents the instantiation of $\mathcal{C}$ and can be returned.

What remains to be described is the function *amend*. The pseudo-code for this function is given in figure 5.8.

This function loops over the parameter specifications in the second given set

$PS^{\mathcal{C}}$. For each parameter specification $\prec R, F^{\mathcal{C}} \succ$ with role name R and role filler term $F^{\mathcal{C}}$, the algorithm tests whether there is a parameter specification $\prec R, F^{\mathcal{C}'} \succ$ in the first set $PS^{\mathcal{C}'}$, i.e. whether there is a parameter specification for the same role name. This represents the case where a parameter from the super-action is bound to a value. If so, the algorithm extends the given substitution σ such that the two terms that are the role fillers, $F^{\mathcal{C}}$ and $F^{\mathcal{C}'}$, are unified under σ. Then the old parameter specification $\prec R, F\mathcal{C}' \succ$ is replaced by the instantiated parameter specification $\prec R, \sigma(F^{\mathcal{C}}) \succ$ in $PS^{\mathcal{C}'}$. If there was no parameter specification with the same role name, the parameter is an additional parameter for the inheriting action. In this case the new parameter specification $\prec R, F^{\mathcal{C}} \succ$ just has to be added to $PS^{\mathcal{C}'}$.

This concludes the description of *amend* and the algorithm for the capability subsumption test.

5.2.3.3 An Example

We will now illustrate the capability instantiation algorithm and how it is used within the capability subsumption test with an example introduced in section 4.3.3. In this example the broker knows about an ontology of actions. For our illustration of the algorithm it will be necessary to define this ontology first. For simplicity this ontology contains only one action, a moving action $\mathcal{M}$, represented by the following CDL description:

```
(capability
  :action move
  :state-language fopl
  :input ((Thing ?thing)(From ?p1)(To ?p2))
  :input-constraints (
    (Has Location ?thing ?p1))
  :output-constraints (
    (not (Has Location ?thing ?p1))
    (Has Location ?thing ?p2)))
```

Notice that this capability is specified in terms of achievable objectives, but it provides an action name, `move`, that can be used to inherit from this description.

For example, the second hospital from the initial scenario uses this capability description to describe a capability $\mathcal{C}$, the capability to move injured people to the hospital:

```
(capability
  :properties (complete)
  :isa move
  :state-language fopl
  :input ((To Hospital2)(Ambulance ?a))
  :input-constraints (
    (elt ?thing Person)
    (Is ?thing Injured)))
```

Now suppose the **pp**-agent sends the following problem description $\mathcal{T}$ to the broker, asking for an agent that can deal with this problem:

```
(task
  :state-language fopl
  :input-constraints (
    (elt JohnSmith Person)
    (Is JohnSmith Injured)
    (Has Location JohnSmith Pacifica))
  :output-constraints (
    (Has Location JohnSmith Hospital2)))
```

When the broker receives this problem it will eventually test whether the capability $\mathcal{C}$ advertised by the **h2**-agent subsumes the task $\mathcal{T}$. The algorithm that tests for capability subsumption starts with the pseudo-code in figure 5.6, i.e. it will first test whether the super-action of the capability $sup^{\mathcal{C}}$ is defined. In this example it is, and thus the algorithm will proceed by instantiating the capability description of $\mathcal{C}$.

To instantiate $\mathcal{C}$, the instantiation algorithm described in figure 5.7 will first test whether the given capability description has a super-action, i.e. whether it is indeed described as a performable action. In our example, the super-action is $\mathcal{M}$, i.e. it is defined. Thus the algorithm will proceed by initialising the capability description $\mathcal{C}'$ with the instantiation of $\mathcal{M}$, which is effectively a copy of $\mathcal{M}$ since move does not have a super-action. Next, the agent, super-action, action name, and properties of $\mathcal{C}'$ will be modified and a new, empty substitution is created.

In the next step the input parameter specifications of $\mathcal{C}'$ will be amended. These are:

```
[(Thing ?thing_0), (From ?p1_1), (To ?p2_2)]
```

Note that in this example these are exactly the input parameter specifications from the `move` action $\mathcal{M}$. These will be amended with the input parameter specifications from the **h2**-agent's capability description $\mathcal{C}$, which are:

```
[(To Hospital2), (Ambulance ?a_4)]
```

The first of these parameter specifications, `(To Hospital2)`, is an example of a new binding that is introduced by $\mathcal{C}$. The second parameter specification, `(Ambulance ?a_4)` represents an additional parameter introduced by $\mathcal{C}$. Since there are no output parameter specifications in $\mathcal{C}$ or its super-action $\mathcal{M}$, the second call to amend has no effect. Thus, the parameter-unifying substitution is:

```
[Hospital2->[?p2_2]]
```

This substitution can now be used to instantiate constraints from $\mathcal{C}$ and $\mathcal{M}$ for the capability description $\mathcal{C}'$, as described in figure 5.7. The resulting, instantiated capability description is:

```
(capability
  :properties (complete)
  :state-language fopl
  :input (
    (Thing ?thing_0) (From ?p1_1) (To Hospital2) (Ambulance ?a_4))
  :input-constraints (
    (Has Location ?thing_0 ?p1_1)
    (elt ?thing_0 Person)
    (Is ?thing_0 Injured))
  :output-constraints (
    (NOT (Has Location ?thing_0 ?p1_1))
    (Has Location ?thing_0 Hospital2)))
```

Note that this is a capability description in terms of achievable objectives and this description can now be used for the subsumption test outlined in sections 5.1.2 to 5.2.2.

```
function boolean subsumes(C, T)

    if sup^C is defined then
        return subsumes(instantiate(C), T)
    if sup^T is defined then
        return subsumes(C, instantiate(T))
    if P^T ⊈ P^C then
        return false

    outKB ← new KnowledgeBase(S^C)
    for c ∈ {C^C_O1, ..., C^C_Olc} do
        assert(outKB, c)
    for c ∈ {C^C_IO1, ..., C^C_IOmc} do
        assert(outKB, conclusionOf(c))
    σ ← evaluate(outKB, C^T_O1 ∧ ... ∧ C^T_OlT, {v_1, ..., v_h})
    if σ is undefined then
        return false

    inKB ← new KnowledgeBase(S^C)
    for c ∈ {C^T_I1, ..., C^T_IlT} do
        assert(inKB, c)
    for c ∈ {C^C_I1, ..., C^C_Ilc} do
        if not evaluate(inKB, σ(c)) then
            return false

    for c ∈ {C^C_IO1, ..., C^C_IOmc} do
        if usedInProof(outKB, conclusionOf(c)) then
            if not evaluate(inKB, σ(premiseOf(c))) then
                return false

    return true
```

Figure 5.9: Final version of the subsumption algorithm

5.2.4 The Subsumption Algorithm

The final version of the capability subsumption test that incorporates all the functions described in the previous sections is given in figure 5.9.

5.2.4.1 Capability Evaluation in the Expressiveness Scenario

Before we turn to the integration of the capability subsumption test into a capability retrieval algorithm (cf. section 5.3), we will look at one more example: the expressiveness scenario (example 3.2). The most interesting capability in this scenario was advertised by the **h2**-agent: it can treat injured people from Abyss, Barnacle, or Exodus, but if there is snow or ice on the road, the ambulance needs snow chains for this capability to be applicable. The CDL description for this capability was given in section 4.5.1 as follows:

```
(capability
  :state-language fopl
  :input ((InjuredPerson ?person))
  :input-constraints (
    (elt ?person Person)
    (Is ?person Injured)
    (or
      (Has Location ?person Abyss)
      (Has Location ?person Barnacle)
      (Has Location ?person Exodus))
    (implies (or (on Road Ice) (on Road Snow))
             (have Ambulance SnowChains)))
  :output-constraints (
    (not (Is ?person Injured))))
```

The problem description we want to look at here has also been introduced in section 4.5.1. It is the last problem description there, but the one which will be subsumed by the above capability description. In this problem the injured person is in Exodus, there is snow on the roads, and the ambulance has snow chains. The problem description in CDL was:

```
(task
  :state-language fopl
  :input-constraints (
    (elt JohnSmith Person)
    (Is JohnSmith Injured)
    (Has Location JohnSmith Exodus)
    (on Road Snow)
    (have Ambulance SnowChains))
  :output-constraints (
    (not (Is JohnSmith Injured))))
```

The capability subsumption test (cf. figure 5.9) begins with the instantiation of capability and task, but in this example, both are already described in terms of achievable objectives, and thus, no instantiation needs to take place. Next is the test for properties which succeeds because neither capability nor task mention any properties. The next step then is the output match condition. To test this condition, the algorithm first creates a knowledge base for the capabilities output constraints and the conclusions of any input-output constraints. In this example, this knowledge base will contain only one expression:

```
(NOT (Is ?person_3 Injured))
```

Next the query for this knowledge base is generated as the conjunction of all the task's output constraints, and again there is only one in this example:

```
(NOT (Is JohnSmith Injured))
```

The third argument for the call to *evaluate* is the set of variables we want to know the substitution for:

```
[?person_3]
```

Now the query can be evaluated and the function *evaluate* returns the following substitution with which the query could be derived:

```
[JohnSmith->[?person_3]]
```

The next part of the algorithm is for the input match condition. For this purpose, the algorithm creates another knowledge base and initialises it with the task's input constraints:

```
(elt JohnSmith Person)
(Is JohnSmith Injured)
(Has Location JohnSmith Exodus)
(on Road Snow)
(have Ambulance SnowChains)
```

The capabilities input constraints are now evaluated against this knowledge base. The first two input constraints from the capability description are trivially true again as they are elements of the knowledge base:

```
(elt JohnSmith Person)
(Is JohnSmith Injured)
```

The next input constraint states that the injured person must be either in Abyss, Barnacle, or Exodus:

```
(OR
  (Has Location JohnSmith Abyss)
  (Has Location JohnSmith Barnacle)
  (Has Location JohnSmith Exodus))
```

Again this evaluation succeeds because the injured person is in Exodus in this example. Finally the conditional input constraint of this capability remains to be tested:

```
(IMPLIES
  (OR (on Road Ice) (on Road Snow))
  (have Ambulance SnowChains))
```

Again, it can be seen fairly easily that this follows from the input knowledge base. Thus, as there are no input-output constraints to be evaluated, the capability subsumption test succeeds.

5.3 Capability Retrieval in JAT

In this section we will show how the Java Agent Template was used to implement the agents described in this thesis. We will also describe how the broker reacts to different message types it may receive and how it retrieves capabilities.

5.3.1 The Java Agent Template

The Java Agent Template (JAT)[4] is implemented as a library of classes written in the programming language Java [Eckel, 1997, Campione and Walrath, 1998] that provides the developer of software agents with a number of useful objects. In particular, JAT provides:

- **JavaAgent.agent**: a number of classes for the convenient implementation of software agents. These classes include the agent template, several classes for concurrent message sending, receiving, and buffering, a resource manager, and a special agent called the Agent Name Server (ANS).
- **JavaAgent.resource**: a number of resource classes an agent might need. The most important resource for us is the KQML message class. Other resources include languages and interpreters which can both be managed by the resource manager that comes with an agent.
- **JavaAgent.context**: a number of classes that represent the context in which an agent is embedded. These classes include the low level context interface for communication with other agents and various graphical interface objects for the GUI. These classes are not of much interest to us.

All the agents presented in this thesis have been implemented using JAT. Ideally, we would not have modified the JAT code in order to improve the reusability of the code developed for this thesis. In fact, we have made a few

[4] JAT is available on the WWW at URL: `http://cdr.stanford.edu/ABE/JavaAgent.html`

modifications to JAT, namely ones to remove bugs or to change the interface slightly. Thus, our code should still work, for the most part, with JAT in its standard form. However, while JAT is still available on the WWW, it is not supported anymore and its successor, JATLite[5] is not compatible with JAT and it has not been tested with our agents.

5.3.1.1 JAT Agents

A JAT agent has *two groups of functions* it can perform: it can process KQML messages (cf. section 2.1.2.3) and manage different types of resources.

An agent can process KQML *messages* in two ways. Firstly, it provides a function `sendMessage` which takes a KQML message and transmits it to the receiver named in the message. Secondly, it starts a separate thread that constantly monitors the socket associated with this agent for incoming messages. When the agent receives a message it passes this message to an appropriate interpreter that can deal with it. Both, incoming and outgoing messages are buffered to make sure the agent will not be deadlocked by the communication with other agents. KQML messages are treated as resources in JAT and will be described in more detail in section 5.3.1.2.

Apart from sending and receiving messages an agent also manages a set of *resources*. Any Java class can be managed as a resource by an agent. The resource manager of an agent will just associate the URL where this class can be obtained with the class name. For some special resource classes the resource manager does more than this though. For example, a special type of resource is an interpreter that can deal with messages received by the agent. Other special types of resources are languages and addresses. Resources will be described in more detail below.

When a JAT agent is created it first attempts to *initialise itself*. For this purpose, it is given a file as one of its parameters which contains a number

[5] See URL: `http://java.stanford.edu/` for information on JATLite.

of KQML messages that the agent will load and interpret before it processes any other messages. These messages must provide the agent with all the information it needs to process future messages from other agents. For example, a message from the initialisation file might inform the agent where to find a certain interpreter for processing messages. During the initialisation the agent will also automatically send a message to a special facilitation agent telling it this agent's name and address.

One special agent that is defined as part of JAT is this facilitation agent: the *Agent Name Server* (ANS). The purpose of this agent is to associate agent addresses with names and provide this information to other agents on request.

5.3.1.2 JAT Resources

One of the most important resource types managed by an agent is the *interpreter*. When an agent receives a KQML message it extracts the ontology slot value from this message and asks the resource manager whether an interpreter for this ontology is known. In other words, the ontology of a message determines the interpreter this message will be passed on to in JAT. Note that this is a rather unusual notion of ontology (cf. section 2.3.2). If the resource manager knows of an interpreter class that is associated with the given ontology then a new interpreter of this type will be created in a separate thread, and the message will be given to this interpreter object for processing. For this purpose, every interpreter provides a function `interpretMessage` that takes a KQML message and an agent, the receiver of this message, as arguments. Note that by creating a new interpreter as a separate thread for every message received, no message can deadlock the agent. Unfortunately it also means that processing of messages happens in parallel, i.e. not necessarily in the order in which they are received.

One interpreter defined in JAT which is known to every agent by default is the `AgentInterpreter`. Messages that name the ontology `agent` will be passed to this interpreter. The reason this interpreter must be provided to every agent

is simply due to the fact that all agents must be able to interpret messages from the initialisation file. The messages this interpreter understands are all related to the management of resources. Messages to this interpreter must have the performative `evaluate` and the content language must be KQML. A number of different performatives are allowed for the content KQML message, the most important is `tell-resource` with which the resource manager of the receiving agent can be informed of the name and location of a new resource, for example, an interpreter class.

Other types of resources are *addresses* and *languages*. Addresses are associated with agent names so that KQML messages can just name the receiver of the message and the function `sendMessage` can retrieve the agent's address from the resource manager. KQML messages also allow the naming of the language that is used in the content of a message and languages are also resources managed by an agent. Notice that CDL uses the same mechanism and thus, can also use the resource manager to retrieve unknown languages. Every language provides a function `parseString` which takes a string and parses it into an object that is an instance of this language class.[6] Note that it is up to the interpreter to call this function to parse the content of a given KQML message.

One of the languages provided with JAT is KQML (cf. section 2.1.2.3) and every agent knows about this language by default, just like it knows about the `AgentInterpreter` for much the same reasons. A KQML message in JAT consists essentially of a performative and a number of field-value pairs. The performative can be an arbitrary name, i.e. it is not restricted to the predefined performatives in the KQML definition [Labrou and Finin, 1997]. Field names can also be arbitrary, but every complete KQML message must contain at least the following: `:sender`, `:receiver`, `:ontology`, `:language`, and `:content`. The support for further field names in JAT's implementation of KQML is rather limited.

[6] Those who have used object-oriented programming languages like Java will know that creating a new object in this way is not possible, and the implementation of this functionality is in fact rather messy.

5.3.2 The CDL Interpreter

In our implementation the broker is an ANS that knows about an interpreter that deals with all capability brokering related messages, the CDL interpreter. In JAT incoming messages are passed on to an interpreter according to the ontology named in the message. KQML messages that name the ontology `capabilities` are passed to a new CDL interpreter for processing. A CDL interpreter understands most of the KQML brokering performatives described in table 2.1. The only difference is that the content must be in CDL rather than KQML.

5.3.2.1 Loading an Ontology of Actions

One additional performative provided by the CDL interpreter is `load-ontology` which can be used to tell the broker about an ontology of actions. Descriptions of capabilities and tasks as performable actions can inherit from this ontology of actions once it has been loaded by the broker. Note that this ontology is not an ontology in the JAT sense, i.e. it is not an interpreter. The content of a `load-ontology` message should be the URL pointing to the ontology. On receipt of this message the CDL interpreter will open the URL and read a number of CDL expressions describing capabilities. For each CDL expression read, the broker parses this expression into an object of type CDL description and associates this object representing a capability description with its action name in a hash table for future reference.

5.3.2.2 Capability Advertisements

Once the broker is initialised and has loaded the ontology it is ready to receive the capability advertisements from the PSAs. Messages advertising capabilities must be addressed to the CDL interpreter by naming the ontology `capabilities`, and they must have the performative `advertise`. The content of such a message must be a KQML message again, the message the advertising agent claims it can process. The performative of this content must be either `achieve` or `perform`

```
function recommendOne(A, T)

    for c ∈ V^C
        if subsumes(c, T)
            sendMessage("forward ...")
            return
    sendMessage("sorry ...")
```

Figure 5.10: Essential Capability Retrieval

and its content must be a CDL expression. The performative should be `achieve` if the CDL expression describes a capability as an achievable objective, and it should be `perform` if the CDL expression describes a performable action. Note that `perform` is not a standard KQML performative.

On receipt of such a message the CDL interpreter will extract the CDL expression from the message, parse it into an object of type CDL description, and add it to the list of capabilities known to this broker. Storing capabilities in a flat list is obviously inefficient for retrieval. However, as it was beyond the scope of this thesis to develop a large number of agents, we have chosen this simple representation. Scaling issues are discussed in section 6.3.

5.3.2.3 Brokering

The brokering performatives provided by the CDL interpreter are fairly similar. All of them provide essentially the functionality associated with the performative `recommend-one`. On receipt of a message with this performative the CDL interpreter calls the function *recommendOne* which is given as pseudo code in figure 5.10.

This function takes two arguments: $\mathcal{A}$, the agent seeking the capability; and $\mathcal{T}$, the CDL description of the problem the capability is being sought for. The function *recommendOne* will go through the vector of all the capability descrip-

tions that have been advertised previously, represented in the algorithm as $V^{\mathcal{C}}$, and tests for each capability whether it subsumes the given task $\mathcal{T}$. If so, a message forwarding the capability holding agent's capability description will be sent to $\mathcal{A}$. If no subsuming capability was found the function sends a message to $\mathcal{A}$ indicating that no agent capable of solving the given problem was found.

If the performative of the capability seeking message was `recommend-all` then the function *recommendAll* will be called by the CDL interpreter. This is like *recommendOne*, only that it does not return after the first matching capability has been found. Instead it will keep forwarding all matching capability descriptions to $\mathcal{A}$, followed by a message indicating the end of this stream of messages. If no matching capabilities were found only one message to that effect will be sent.

In analogy to *recommendOne* and *recommendAll* the CDL interpreter has two functions *brokerOne* and *brokerAll* which are called for messages with the performatives `broker-one` and `broker-all` respectively. According to the KQML specification, the only difference between recommending and brokering should be that, instead of forwarding the capability description to the capability seeker, the broker manages the solution of the problem. This is done by sending the content of the matching capability advertisement to the capability advertiser.

Our implementation of *brokerOne* does more than the above though. If the broker fails to find a single agent that has a capability that subsumes the given task, the broker will attempt to find a plan involving the capabilities of several agents that can address the described problem. The planner is a rather simple, SNLP-like, partial-order, causal link planner [McAllester and Rosenblitt, 1991], implemented in Java. The task's input constraints and output constraints are used to specify the problem and capabilities are translated into operator schemata. However, preconditions and effects of operators are limited to lists of literals and any capability description from which these cannot be extracted, e.g. because it uses a state language which is too powerful, it cannot be transformed into an operator schema.

The reason why this planning is only implemented for the `broker-one` performative is simply that recommendation-based performatives expect an agent capability as reply, not a plan. Once a plan has been generated by the CDL interpreter, the broker could proceed by managing the execution of this plan and dealing with various problems that might occur during the execution until the given problem has been solved. However, as this is beyond the brokering problem addressed in this thesis, we have chosen not to implement the plan execution phase.[7]

5.3.3 An Example: The Flexibility Scenario

We will now look at the messages necessary to implement the flexibility scenario (example 3.3) to illustrate what was discussed above. All the messages listed are from the broker's perspective and are taken from its trace. The first set of messages are the ones the broker receives from its initialisation file. The first of these messages is the same for all agents, telling them the fixed address of the broker:

```
(evaluate
  :sender init-file
  :content
  (tell-resource
    :type address
    :name ANS
    :value hera.dai.ed.ac.uk:5001)
  :ontology agent
  :receiver ANS
  :language KQML)
```

This message is somewhat superfluous for this agent, but it is sent all the same in JAT. The next message is more interesting, telling the broker the location of the Java class for CDL interpreters and associating them with the name `capabilities`:

[7] For a brief review of work related to planning agents and plan execution see section 2.3.4.

```
(evaluate
  :sender init-file
  :content
    (tell-resource
      :type interpreter
      :name capabilities
      :value (http://www.dai.ed.ac.uk/students/gw/jat/classes
        JavaAgent.resource.cdl.CDLInterpreter))
  :ontology agent
  :receiver ANS
  :language KQML)
```

Now the broker knows about the CDL interpreter and can process messages that mention the ontology `capabilities`. The next message, however, mentions the ontology agents, i.e. it is for the `AgentInterpreter` again, and tells the broker where to find another resource, namely the state language we have been using in most CDL expressions:

```
(evaluate
  :sender init-file
  :content
    (tell-resource
      :type language
      :name fopl
      :value (http://www.dai.ed.ac.uk/students/gw/jat/classes
        JavaAgent.resource.fopl.FOPLFormula))
  :ontology agent
  :receiver ANS
  :language KQML)
```

The last message from the initialisation file for the broker is actually a message to the CDL interpreter. With this message the broker is told to load an action ontology from the given URL:

```
(load-ontology
  :sender init-file
  :content
    http://www.dai.ed.ac.uk/students/gw/jat/sc/actions.cdl
  :ontology capabilities
  :receiver ANS
  :language URL)
```

The ontology that is being loaded contains only one capability description, the one for the moving action mentioned above. As this capability description

plays no role in the flexibility scenario, we will not repeat it here. However, it is worth noting that this capability uses the state language made known to the broker in the previous message, i.e. this message was actually necessary at this point.

Now the broker is ready to receive capability advertisements from the various PSAs. The first agent to advertise its capability in this scenario is the **h1**-agent, but before the capability advertisement is sent, it has to tell the broker its address:

```
(evaluate
  :sender h1
  :content
    (tell-resource
      :type address
      :name h1
      :value hobby.dai.ed.ac.uk:38197)
  :ontology agent
  :receiver ANS
  :language KQML)
```

This message is followed by the capability advertisement. The rather lengthy appearance of this message is due to the fact that this message contains all wrapper layers omitted in previous descriptions.

```
(advertise
  :sender h1
  :content
    (achieve
      :receiver h1
      :ontology OPlan
      :language CDL
      :content
        (capability
          :state-language lits
          :input ((InjuredPerson ?person))
          :input-constraints (
            (elt ?person Person)
            (Is ?person Injured)
            (Has Location ?person Hospital1))
          :output-constraints (
            (not(Is ?person Injured)))))
  :ontology capabilities
  :receiver ANS
  :language KQML)
```

Note that this capability advertisement names the state language as `lits`. When the CDL interpreter gets this message, extracts the string representing the CDL expression, and attempts to parse it, it will ask the resource manager of the broker for this language. A language named `lits`, however, is unknown to the broker at this point and thus, the broker will send a message to the agent advertising the capability using this language value to inquire where to find this language:

```
(evaluate
  :sender ANS
  :content
    (ask-resource
      :type language
      :name lits)
  :ontology agent
  :receiver h1
  :language KQML)
```

Next the **h1**-agent, which advertised the capability and thus presumably knows where the language called `lits` can be found, sends this language's URL to the broker:

```
(evaluate
  :sender h1
  :content
    (tell-resource
      :type language
      :value (http://www.dai.ed.ac.uk/students/gw/jat/classes
        JavaAgent.resource.fopl.LitLObject):name lits)
  :ontology agent
  :receiver ANS
  :language KQML)
```

This convenient handling of languages as resources is one of the prime features of CDL and contributes to its flexibility.

The next messages are from the **h2**-agent informing the broker of this agent's address and advertising its capability. To keep this example short, we will only repeat the content of the capability advertisement here:

```
(capability
  :state-language fopl
  :input ((InjuredPerson ?person))
  :input-constraints (
    (elt ?person Person)
    (Is ?person Injured)
    (or (Has Location ?person Abyss)
      (Has Location ?person Barnacle)
      (Has Location ?person Exodus)))
  :output-constraints (
    (not(Is ?person Injured))))
```

The final agent to advertise its capability in this scenario is the new agent, the ambulance service. Again, we have omitted the message telling the broker the **as**-agent's address and give only the content of the capability advertisement here:

```
(capability
  :state-language lits
  :input ((InjuredPerson ?person)(From ?p1)(To ?p2))
  :input-constraints (
    (elt ?person Person)
    (Is ?person Injured)
    (Has Location ?person ?p1))
  :output-constraints (
    (not(Has Location ?person ?p1))
    (Has Location ?person ?p2)))
```

Next, the **pp**-agent tells the broker its address, and this message is followed by a request to manage the solution of the described problem:

```
(broker-one
  :sender pp
  :content
    (task
      :state-language lits
      :input-constraints (
        (elt JohnSmith Person)
        (Is JohnSmith Injured)
        (Has Location JohnSmith Delta))
      :output-constraints (
        (not(Is JohnSmith Injured))))
  :ontology capabilities
  :receiver ANS
  :language CDL)
```

Note that the performative here is `broker-one`. When the CDL interpreter gets this message it will first try to find one agent that can solve the described problem. For this purpose it will test for each capability it knows about to determine whether it subsumes the given task. The capability description of the **h1**-agent will fail because of its last input constraint:

```
(Has Location JohnSmith Hospital1)
```

This input constraint cannot be satisfied by the input constraints in the task description. The next capability tested is that of the **h2**-agent. Again, this capability fails on one of its input constraints:

```
(OR (Has Location JohnSmith Abyss)
    (Has Location JohnSmith Barnacle)
    (Has Location JohnSmith Exodus))
```

The problem here is that John Smith, the injured person, is located in Delta. The final capability tested is that of the **as**-agent which fails on the output match condition. Since no capability subsuming the problem could be found, the CDL interpreter will attempt to plan. The first step towards planning is the generation of operator schemata from the capability description. The capability of the **h1**-agent can be transformed into the following operator:

```
(op [?person_3]
:preconds [
  (elt ?person_3 Person),
  (Is ?person_3 Injured),
  (Has Location ?person_3 Hospital1)]
:effects [
  (NOT (Is ?person_3 Injured))])
```

The transformation of the **h2**-agent's capability into an operator fails because of its disjunctive input constraints. Thus, the **h2**-agent's capability cannot be part of the plan. Finally, the transformation of the **as**-agent's capability into an operator succeeds:

```
(op [?person_5, ?p1_6, ?p2_7]
:preconds [
  (elt ?person_5 Person),
  (Is ?person_5 Injured),
  (Has Location ?person_5 ?p1_6)]
:effects [
  (NOT (Has Location ?person_5 ?p1_6)),
  (Has Location ?person_5 ?p2_7)])
```

The next step is the generation of an initial partial plan from the problem description. This will succeed as the problem description is straight-forward. This incomplete plan will then be completed by the SNLP-like planner and again, this will be successful. The resulting plan is the rather trivial action sequence of getting the injured person to the second hospital and then instructing the second hospital to treat the injured person.

5.3.4 Problems with JAT

While we feel that using JAT for the implementation of our agents has saved considerable effort, it also brought with it some problems. This is only to be expected though as JAT is only intended to be a research vehicle. One of the problems with JAT is rooted in the way the class `Language` is implemented. All languages managed by a JAT agent's resource manager must be derived from this class. However, due to the non-abstract implementation of this class in Java, not every class can be derived from the JAT `Language` class. Ultimately this may lead to mismatches during brokering as the reflective reasoning may not find certain functionality required in the subsumption test. We shall return to an example where this problem actually occurred in section 6.2.

Another potential problem with JAT lies in the way received messages are processed. To prevent deadlocks messages are processed in parallel. However, there are situations when this is not desirable. For example during the initialisation of an agent, it receives several messages from the initialisation file (cf. section 5.3.1.1). One of these messages tells it where to find a given interpreter and

a subsequent message is directed to this interpreter. Processing these messages out of order would lead to failure, but with parallel message processing there is no guarantee that these messages will be processed in order. This never caused a problem during our experiments though.

Another potential problem is the way the resource manager loads Java classes as resources from remote hosts. This constitutes a major security risk. Later versions of JAT are addressing this problem, but are not compatible with the version we have used. Finally, as already mentioned, JAT is a research vehicle and the lack of robustness has caused occasional problems.

Summary

In this and the previous chapter we described our capability description language and several algorithms to reason over this language. Basic capabilities are described in terms of achievable objectives in CDL (cf. section 4.2). The algorithm that tests whether such a capability can be used to address a problem is based on the notion of capability subsumption as defined in section 5.1. An alternative to describing capabilities in terms of achievable objectives is to describe them as performable actions and an extension of CDL to deal with this notion is described in section 4.3. Reasoning over such capabilities is performed through instantiation (cf. section 5.2.3). A final feature of the language concerns agent properties. Both, representation and algorithms have been extensively illustrated using the examples from the scenarios presented in chapter 3. Thus, the definition of CDL and description of the algorithms used by our broker to reason about capabilities is now complete.

Chapter 6

Further Experiments and Results

At this point we have defined CDL*, an expressive and flexible action representation that can be used to represent and reason about capabilities of intelligent agents. Our aim in defining this formalism was to address the problem of capability brokering. The next step will be to conduct further experiments with the broker by exploring variations on the scenarios described in section 3.3. The contribution of this chapter will be a summary of the practical results achieved in this thesis.*

Ideally, an evaluation of CDL and our broker would involve the implementation of a large number of different scenarios, with agents that have interestingly different capabilities and which require a number of interestingly different state description languages that the broker does not initially know. Furthermore, the broker should not only be equipped with just the current mechanisms we described in this thesis to address given problems (finding a single agent or finding a plan). Only if CDL and our broker proved fit in such a wide range of scenarios could we claim that we have achieved the goal of providing a generic capability description language and broker suitable for all possible eventualities. However, such an effort is beyond the scope of this thesis. Furthermore, there are limitations to CDL we are aware of, some of which we have pointed out throughout this thesis and some of which we will discuss as possible extensions in section 10.1.

While an exhaustive evaluation of CDL and our broker is not possible in this thesis, we can modify the existing scenarios to provide us with further interesting and relevant examples. Since these variations were not directly considered during the design and implementation of CDL and the broker, they constitute a limited but worthwhile evaluation of the generality and robustness of the language and the brokering mechanisms. The initial scenario was only meant to illustrate the basic communication and thus, we shall only consider variations of the more interesting scenarios in this chapter, i.e. the expressiveness and flexibility scenarios.

6.1 Variations on the Expressiveness Scenario

In this section we will present further interestingly different variations of the expressiveness scenario (example 3.2) to illustrate the generality of our broker as well as some limitations.

To keep the evaluation challenging we will mainly look at minor variations of the problem description in the expressiveness scenario that should result in major changes in the behaviour of the broker, i.e. in failure or success in retrieving an appropriate PSA. The underlying assumption here is that, if the broker can cope well with minor differences in the problems, then it will also be able to distinguish problems with major differences. In fact, the two parts of the problem in the initial scenario, the broken generator and the injured person, represent problems with major differences and our broker coped with these easily.

In the expressiveness scenario (cf. section 3.3.1) the two hospitals have divided the island such that each hospital only deals with emergencies in its half. Barnacle, however, which lies between the two hospitals, is served by both. These conditions are expressed as disjunctive input constraints in the capability descriptions of the two hospitals. Additionally, the second hospital has an input constraint expressing that it needs snow chains if there is ice or snow on the roads. As the capability description for this hospital is essential for this evaluation, it is repeated here from section 4.5.1:

```
(capability
  :state-language fopl
  :input ((InjuredPerson ?person))
  :input-constraints (
    (elt ?person Person)
    (Is ?person Injured)
    (or (Has Location ?person Abyss)
      (Has Location ?person Barnacle)
      (Has Location ?person Exodus))
    (implies (or (on Road Ice)(on Road Snow))
      (have Ambulance SnowChains)))
  :output-constraints (
    (not (Is ?person Injured))))
```

The part of the problem concerning the injured person was communicated to the broker by the **pp**-agent with the following message (cf. section 4.5.1):

```
(recommend-all
  :sender pp
  :content
    (task
      :state-language fopl
      :input-constraints (
        (elt JohnSmith Person)
        (Is JohnSmith Injured)
        (Has Location JohnSmith Delta))
      :output-constraints (
        (not (Is JohnSmith Injured))))
  :ontology capabilities
  :receiver ANS
  :language CDL)
```

This message asks the broker to recommend all agents that can deal with the problem described. In this problem, there is an injured person to be treated, and the location of this person is Delta. In response to this message the broker will first recommend the **h1**-agent, which is the only hospital dealing with patients in Delta, by forwarding its capability description to the **pp**-agent.[1] The second message with which the broker will reply to the above request is a message indicating that it will recommend no further agents:

[1] We will not include messages forwarding capability descriptions here as these are rather lengthy and do not add to the evaluation.

```
(eos
  :sender ANS
  :ontology agent
  :receiver pp)
```

The performative `eos` is short for "end of stream", a standard KQML performative, indicating that this is the last in a stream of messages.

There are two relatively trivial variations on this problem we want to consider next. Firstly, we can ask the broker to recommend one agent that can solve the problem but we omit the input-constraint which requires the person to actually be injured. This results in the following problem description:[2]

```
(task
  :state-language fopl
  :input-constraints (
    (elt JohnSmith Person)
    (Has Location JohnSmith Delta))
  :output-constraints (
    (not (Is JohnSmith Injured))))
```

In reply to this problem the broker only sends one message which indicates that there is no problem-solving agent (PSA) that can solve this problem:

```
(sorry
  :sender ANS
  :ontology agent
  :receiver pp)
```

Like the `eos` performative, the `sorry` performative is a standard KQML performative. It is sent in reply to a request for which no satisfactory answer could be found. Here, it indicates that no PSA with appropriate capabilities is known to the broker. This reply might perhaps be a little surprising as the goal, `JohnSmith` not injured, is easily achieved given that he is not injured in the first place. However, both hospitals state as an input constraint that the given person must be injured and thus, their capabilities are not applicable.

[2] For brevity, we shall omit the outer part of the problem-describing messages in this section. The performative used will be mentioned in the paragraph preceding the problem description.

The second rather trivial variation is one where we ask the broker to recommend one agent that can make the given injured person injured, i.e. we drop the negation in the output constraint:

```
(task
  :state-language fopl
  :input-constraints (
    (elt JohnSmith Person)
    (Is JohnSmith Injured)
    (Has Location JohnSmith Delta))
  :output-constraints (
    (Is JohnSmith Injured)))
```

Again, one might suspect that every agent should be capable of not causing a change, but no agent in this scenario has advertised a capability that has a matching output constraint and thus, the broker will reply with a sorry message again.

Apart from these trivial variations, the most obvious interesting variation on the expressiveness scenario is the location of the injured person. Due to the way the island is split between the two hospitals, the places of interest are Calypso or Delta, which are only served by the **h1**-agent, Abyss or Exodus, which are only served by the **h2**-agent, and Barnacle, which is served by both hospitals. We have already described the behaviour of the broker for the Delta case. Thus, we shall move the injured person to Exodus next and ask the broker to recommend all agents that can deal with this problem. We shall also add input constraints to the problem to account the effect of the conditional input constraint of the **h2**-agent's capability. The resulting problem description is as follows:

```
(task
  :state-language fopl
  :input-constraints (
    (elt JohnSmith Person)
    (Is JohnSmith Injured)
    (Has Location JohnSmith Exodus)
    (on Road Snow)
    (have Ambulance SnowChains))
  :output-constraints (
    (not (Is JohnSmith Injured))))
```

In reply to this message the broker first forwards the capability description of the **h2**-agent to the **pp**-agent, thereby indicating that this agent is capable of solving the given problem. The second message from the broker is an `eos` message indicating that no other PSAs with the desired capabilities could be found. As we would expect, the first hospital is not mentioned in the replies as it does not deal with patients in Exodus.

The third interestingly different option for the location of the injured person in this scenario is Barnacle. Again, we shall include the input constraints which account for the conditional input constraint of the **h2**-agent in the problem description and ask the broker to recommend all agents capable of solving this problem:

```
(task
  :state-language fopl
  :input-constraints (
    (elt JohnSmith Person)
    (Is JohnSmith Injured)
    (Has Location JohnSmith Barnacle)
    (on Road Snow)
    (have Ambulance SnowChains))
  :output-constraints (
    (not (Is JohnSmith Injured))))
```

In reply, the broker first forwards the capability description of the first hospital, then the capability description of the second hospital, and finally, it sends an `eos` message concluding its reply. Notice that the applicability of the first hospital's capability is not affected by the additional input constraints in the problem description.

The next group of interestingly different variations concern the conditional input constraint of the second hospital. To begin with a simple example, let the patient be at Exodus, let the ambulance have snow chains, and let there be both, ice and snow on the road. Now we can ask the broker to recommend one agent for this problem:

```
(task
  :state-language fopl
  :input-constraints (
    (elt JohnSmith Person)
    (Is JohnSmith Injured)
    (Has Location JohnSmith Exodus)
    (on Road Snow)
    (on Road Ice)
    (have Ambulance SnowChains))
  :output-constraints (
    (not (Is JohnSmith Injured))))
```

Our broker does not get confused by the fact that both conditions of the **h2**-agent's conditional input constraint are satisfied and correctly recommends this agent to the **pp**-agent.

The next variation we have considered omits all information about the present road conditions, i.e. we have deleted the respective input constraints from the problem description to the broker. Thus, we have asked the broker to recommend all agents that are capable of addressing the following problem:

```
(task
  :state-language fopl
  :input-constraints (
    (elt JohnSmith Person)
    (Is JohnSmith Injured)
    (Has Location JohnSmith Barnacle)
    (have Ambulance SnowChains))
  :output-constraints (
    (not (Is JohnSmith Injured))))
```

In this case the broker first recommends the **h1**-agent which serves Barnacle and is unaffected by the road conditions anyway. Next the broker recommends the **h2**-agent. The reason for this is simply that the availability of snow chains suffices to show that its conditional input constraint will be satisfied. Finally, the broker sends the `eos` message as before. If we move the patient to Exodus and ask the broker to recommend one PSA, the broker correctly replies by forwarding the **h2**-agent's capability description only.

As an alternative to omitting information about the road conditions, we shall now consider the case where we do not provide snow chains to the ambulance.

The problem description does, however, mention that there is snow on the road. Again, we ask the broker to recommend all agents for the following problem description:

```
(task
  :state-language fopl
  :input-constraints (
    (elt JohnSmith Person)
    (Is JohnSmith Injured)
    (Has Location JohnSmith Barnacle)
    (on Road Snow))
  :output-constraints (
    (not (Is JohnSmith Injured))))
```

For this problem the broker only recommends the **h1**-agent, followed by an `eos` message. The reason for this is that the conditional input constraint of the **h2**-agent causes its capability to be inapplicable for the problem described. Similarly, if we move the patient to Exodus and ask the broker to recommend one agent that can address this problem, the reply by the broker is a `sorry` message. However, if we change the road conditions by providing the information that there is neither ice nor snow on the road, i.e. if we negate the respective input constraints in the problem description, the broker will correctly recommend the **h2**-agent by forwarding its capability description to the **pp**-agent.

Finally, we shall return to the scenario in which we do not provide information about the road conditions in the problem description, and neither do we mention the availability of snow chains. Thus, we will ask the broker to recommend all agents that can deal with the following problem:

```
(task
  :state-language fopl
  :input-constraints (
    (elt JohnSmith Person)
    (Is JohnSmith Injured)
    (Has Location JohnSmith Barnacle))
  :output-constraints (
    (not (Is JohnSmith Injured))))
```

The reply by the broker to this request might seem a little surprising at first: it only forwards the capability description of the **h1**-agent followed by an `eos`

recommend	location	snow	chains	other	reply
all	Delta	no	no	-	h1, eos
one	Delta	no	no	not injured	sorry
one	Delta	no	no	make injured	sorry
all	Exodus	yes	yes	-	h2, eos
all	Barnacle	yes	yes	-	h1, h2, eos
one	Exodus	yes	yes	-	h2
one	Exodus	yes	yes	ice	h2
all	Barnacle	no	yes	-	h1, h2, eos
one	Exodus	no	yes	-	h2
all	Barnacle	yes	no	-	h1, eos
one	Exodus	yes	no	-	sorry
one	Barnacle	no	no	no snow or ice	h2
all	Barnacle	no	no	-	h1, eos
all	Exodus	no	no	-	sorry

Table 6.1: Variations on the expressiveness scenario

message. Similarly, if we move the patient to Exodus, the reply will be a `sorry` message. The reason for the inapplicability of the **h2**-agent's capability is the conditional input constraint: if no snow chains are available, the broker has to prove that there is neither snow nor ice on the road in order to show that the capability is applicable. This cannot be shown here. The underlying reason for this behaviour is that our implementation of the state language used, `fopl`, does not adopt the closed world assumption, i.e. the fact that we have not mentioned that there is snow on the road only means that we do not know whether there is snow or not. Given that the weather conditions on our island are often bad this behaviour is appropriate.

The variations on the expressiveness scenario that have been described in this section are summarised in table 6.1.[3] The first column indicates whether the performative was `recommend-one` or `recommend-all`. The second column gives the location of the injured person. The third and fourth column indicate whether `(on Road Snow)` and `(have Ambulance SnowChains)` were specified as

[3] The order of the table rows reflects the order in which variations have been described in this section.

input constraints in the problem description. The fifth column indicates any other variations that might have been specified, the details of which were described in the above discussion. The final column gives the replies by the broker, where "h1" stands for the forwarding of the **h1**-agents capability description, etc.

In summary, it can be said that we were quite satisfied with all the responses generated by the broker. Furthermore, all of these messages were generated in one session, indicating some robustness of the implementation.

6.2 Variations on the Flexibility Scenario

In this section we will present further interestingly different variations of the flexibility scenario (example 3.3) to illustrate the generality of our broker as well as some limitations.

In the flexibility scenario (cf. section 3.3.2), the capability advertised by the **h1**-agent was changed such that the first hospital could only treat patients at the hospital, i.e. patients cannot be transported to the hospital. The **h2**-agent still only treats patients in Abyss, Barnacle, or Exodus, but its conditional input constraint has been dropped for this scenario. Finally, a new PSA was introduced in this scenario: the ambulance service. The capability advertised by the **as**-agent is that it can transport patients from any place on Pacifica to another, in particular, to a hospital.

One of the most interesting message exchanges in this scenario takes place before the **pp**-agent asks the broker for any help, namely when the **h1**-agent advertises its capability (cf. section 4.5.2):

```
(advertise
  :sender h1
  :content
    (achieve
      :receiver h1
      :ontology OPlan
      :language CDL
      :content
        (capability
          :state-language lits
          :input ((InjuredPerson ?person))
          :input-constraints (
            (elt ?person Person)
            (Is ?person Injured)
            (Has Location ?person Hospital1))
          :output-constraints (
            (not (Is ?person Injured)))))
  :ontology capabilities
  :receiver ANS
  :language KQML)
```

At this point the broker sees the state language named `lits` for the first time. This is a very important point in this scenario, as it illustrates the flexibility of CDL. Since the broker does not know this language it sends a message to the capability advertiser, the **h1**-agent, asking it where to find the language in question; `lits` in this example:

```
(evaluate
  :sender ANS
  :content
    (ask-resource
      :type language
      :name lits)
  :ontology agent
  :receiver h1
  :language KQML)
```

In reply to this request from the broker, the **h1**-agent supplies the Java class that corresponds to the state representation language `lits`, which is managed as a resource of type language by the **h1**-agent's resource manager as explained in section 5.3.1. The actual message supplies the URL of the Java class to the broker:

```
(evaluate
  :sender h1
  :content
    (tell-resource
      :type language
      :value (http://www.dai.ed.ac.uk/students/gw/jat/classes
        JavaAgent.resource.fopl.LitLObject)
      :name lits)
  :ontology agent
  :receiver ANS
  :language KQML)
```

On receipt of this message the broker will attempt to load the class and then continue interpreting the capability advertisement of the **h1**-agent. To create variations of this scenario that would result in a different exchange of messages at this point, it would be necessary to implement further state representation languages, as each resource will only be requested once by the broker. We have

implemented only two different state languages, so no interesting variations can be generated at this point.

We can, however, vary the problem description as we did for the expressiveness scenario above. For example, up to now we have discussed only problem descriptions based on the `broker-one` performative in the flexibility scenario. The recommendation performatives used in the other scenarios also work here. For example, the following message from the **pp**-agent uses the performative `recommend-one`:

```
(recommend-one
  :sender pp
  :content
    (task
      :state-language lits
      :input-constraints (
        (elt JohnSmith Person)
        (Is JohnSmith Injured)
        (Has Location JohnSmith Delta))
      :output-constraints(
        (not (Is JohnSmith Injured))))
  :ontology capabilities
  :receiver ANS
  :language CDL)
```

The broker does not know of any single agent which can deal with this problem. This is mainly due to the location of the patient at Delta and the inability of the broker to plan a solution, because the performative used is `recommend-one` (cf. section 5.3.2.3). Thus, the broker has to reply with a `sorry` message:

```
(sorry
  :sender ANS
  :ontology agent
  :receiver pp)
```

As for the expressiveness scenario, we may generate variations of this problem by moving the patient to another location. We have tested this variation with Barnacle and Exodus, still using the `recommend-one` performative. In both cases the broker correctly responds by forwarding the capability description of the **h2**-agent to the **pp**-agent.

Using the `recommend-one` performative is not very interesting as far as flexibility is concerned. Thus, we shall now return to the `broker-one` performative which opens the possibility for the broker to plan solutions. Only with this performative can we hope to illustrate flexibility. As `broker-one` will be the performative for all remaining messages we shall, as before, drop the outer part of the messages. Again, the first set of variations are concerned with the location of the patient. This was Delta in our original example:

```
(task
  :state-language lits
  :input-constraints (
    (elt JohnSmith Person)
    (Is JohnSmith Injured)
    (Has Location JohnSmith Delta))
  :output-constraints (
    (not (Is JohnSmith Injured))))
```

The reasoning processes which the broker will go through on receipt of this message have been discussed and explained in detail in section 5.3.3. Essentially, the broker fails in its attempt to find a single agent that can deal with the described problem as is and thus attempts to generate a plan involving the capabilities of several agents. This succeeds and the plan consists of first transporting the patient to the first hospital and then treating him there.

Locating the injured person at Barnacle or Exodus results in quite different behaviour for the broker. In both of these cases the broker can find a single agent that can solve the whole problem, the **h2**-agent. Thus, the broker will not attempt to find a plan to solve the given problem. Instead of forwarding the **h2**-agent's capability description to the **pp**-agent, however, the broker now asks the **h2**-agent to solve the given problem. This is due to the performative being `broker-one`, which asks the broker to manage the solution of the problem, even if there is a single agent capable of solving the whole problem. Thus, the broker sends the following message to the **h2**-agent:

```
(achieve
  :sender ANS
  :content
    (task
      :state-language lits
      :input-constraints (
        (elt JohnSmith Person)
        (Is JohnSmith Injured)
        (Has Location JohnSmith Barnacle))
      :output-constraints (
        (NOT (Is JohnSmith Injured))))
  :ontology OPlan
  :receiver h2
  :language CDL)
```

As a final variation, we have considered the case where the location of the injured person is `Hospital1`. The hardly surprising result of this change is that the broker finds the **h1**-agent as the problem solver capable of tackling the given problem and sends it a message virtually identical to the one above, except for the location of the patient and the receiver.

Another interesting variation of this scenario which was not explored in the expressiveness scenario is to change the state language in the problem description. In the original version of the problem only conjunctions of literals are used and the according state language specified was `lits`. Since this is a subset of first-order logic we may change the specification of the state language to `fopl`. For the original problem this results in the following description:

```
(task
  :state-language fopl
  :input-constraints (
    (elt JohnSmith Person)
    (Is JohnSmith Injured)
    (Has Location JohnSmith Delta))
  :output-constraints (
    (not (Is JohnSmith Injured))))
```

The result of this variation is rather interesting: the broker replies to the **pp**-agent with a `sorry` message, i.e. it fails to help with the described problem. Clearly, the broker cannot find a single agent that can solve this problem as

performative	location	state language	response
`recommend-one`	Delta	lits	sorry
`recommend-one`	Barnacle	lits	forward h2
`recommend-one`	Exodus	lits	forward h2
`broker-one`	Delta	lits	plan: as, h1
`broker-one`	Barnacle	lits	h2
`broker-one`	Exodus	lits	h2
`broker-one`	Hospital1	lits	h1
`broker-one`	Delta	fopl	sorry
`broker-one`	Barnacle	fopl	h2
`broker-one`	Exodus	fopl	h2
`broker-one`	Hospital1	fopl	h1

Table 6.2: Variations on the flexibility scenario

it is only the state language that has been changed from the original problem description, not the problem itself. But in this version the broker subsequently fails to find the planned solution. Given that reflection is used to reason about whether certain inferences over the state language expressions need to be made or not, this result is perhaps surprising. Even a closer inspection of the algorithm does not reveal why planning fails for this problem. In fact, the problem here lies in the implementation of JAT, which requires all languages which are managed by a JAT agent's resource manager to inherit from a class provided by JAT. The actual problem is rooted in the way this JAT class is implemented and cannot be changed easily (cf. section 5.3.4).

Less interesting or less surprising are the results of changing the location of the injured person to Barnacle, Exodus, or `Hospital1`. In all of these cases the responses by the broker are the same as those for the case with `lits` as the state description language.

The variations of the flexibility scenario that have been described here are summarised in table 6.2. The first column indicates whether the performative was `recommend-one` or `broker-one`. The second column gives the location of the injured person. The third column names the state language specified. The final

column indicates how the broker responded to the given problem. Details are described and discussed above.

In summary, it can be said that we were again satisfied with all the replies generated by the broker. The only exception was the example involving the state language specified as `fopl` and the patient in Delta, but the underlying problem is rooted in JAT and is not indicative of a limitation in our thesis approach.

6.3 Performance Issues

In this section we will reflect on some performance issues: how expressiveness and flexibility affect broker performance and how our approach might scale up if the number of problem-solving agents was increased drastically.

The first question regarding performance issues we have to answer is, of course, *how fast* the broker actually replies to a request by the problem holder. Unfortunately this question is also rather difficult to answer. We have implemented all the agents and scenarios described in this thesis, but they are rather small in number and thus, a generalisation might be questionable. However, developing a large number of agents and scenarios was beyond the scope of this thesis.

For the scenarios we have tested, the *response times* for the broker to find an appropriate problem solver were virtually instantaneous in all examples.

The broker, the problem-solving agents, and the problem-holding agent are all implemented in Java [Eckel, 1997, Campione and Walrath, 1998] using the Java Agent Template (cf. section 5.3.1). The broker was running on a Sun Sparc station 5 for all scenarios and the various other agents were run on additional Suns on the same LAN. Thus, the response time of the broker includes the time for communication between the different machines, the loading of new Java classes at run time, and the actual retrieval of a capability performed by the broker. Of these, the capability retrieval time is the most interesting for us and we shall analyse this in more detail next.

6.3.1 Complexity of Capability Subsumption

At the heart of capability retrieval is the capability subsumption test and thus, we shall now have a closer look at the complexity of the algorithm performing

```
1:    outKB ← new KnowledgeBase(S^C)
2:    for c ∈ {C^C_O1, ..., C^C_Olc} do
3:        assert(outKB, c)
4:    σ ← evaluate(outKB, C^T_O1 ∧ ... ∧ C^T_Olτ, {v1, ..., vh})
5:    if σ is undefined then
6:        return false

7:    inKB ← new KnowledgeBase(S^C)
8:    for c ∈ {C^T_I1, ..., C^T_Ilτ} do
9:        assert(inKB, c)
10:   for c ∈ {C^C_I1, ..., C^C_Ilc} do
11:       if not evaluate(inKB, σ(c)) then
12:           return false

13:   return true
```

Figure 6.1: Basic capability subsumption algorithm

this test. The most basic form of this algorithm, for capabilities described as achievable objectives, without input-output constraints, inheritance, or properties, was described in figure 5.2 in section 5.1.2.2 and is repeated here for easier reference in figure 6.1 with line numbers.

The algorithm begins with the creation of an empty knowledge base for the capability's output constraints (line 1). We will assume that this can be done in constant time. Similarly, we will assume that the assertion of a single constraint into a knowledge base will take constant time, which may not always be the case but is true for our implementation. Thus, asserting all constraints (lines 2-3) takes linear time in the number of constraints.

In the next step, a sentence representing the conjunction of the task's output constraints is evaluated against the knowledge base (line 4). Obviously, the complexity of this operation depends on the state language used and how the interpreter for the state language performs this test. Normally, one would expect this evaluation to be rather complex, and thus, expensive. For example,

our implementation of first-order logic uses skolemization and resolution theorem proving [Loveland, 1978] to test whether a given sentence follows from the knowledge base. This operation is so complex that it cannot even be analysed in terms of its time complexity: the problem is undecidable. However, our implementation avoids this problem by generating only a limited number of clauses. What this means for the subsumption test is that there might be capabilities that could be used to address a given problem, but due to their complex description the broker is unable to show this. We believe this behaviour to be quite reasonable. For the complexity of the evaluation this means that it can be performed in time exponential in the number of literals in the capability and task description. This time may be required for the transformation of a formula into skolemized conjunctive normal form.

The second part of this algorithm is similar to the first. It consists of the creation of a knowledge base (line 7) followed by the assertion of several sentences (lines 8-9). Then each of the capability's input constraints are evaluated in turn (lines 10-12). Again the evaluation is the crucial step here. This gives us the following complexity result:

> Let c be the number of constraints involved in the evaluation of task $\mathcal{T}$ against capability $\mathcal{C}$. Let the evaluation of a single constraint against a knowledge base be $\mathcal{O}(e(\mathcal{L}))$[4] where $\mathcal{L}$ is the language in which the constraints are expressed. Then the overall complexity of the algorithm for the basic subsumption test described in figure 5.2/6.1 is in $\mathcal{O}(e(\mathcal{L})c)$.

The basic algorithm has been extended in several ways to account for various other features of CDL. For example, the extension for input-output constraints is described in section 5.2.1 and implemented by the algorithm extensions described in figures 5.3 and 5.4. The first part asserts further constraints in the

[4] Note that $e(\mathcal{L})$ may be undefined as it would be for unrestricted resolution.

output knowledge base and the latter performs further evaluations against the input knowledge base. So far, this does not affect the overall complexity of the algorithm. However, the second part of the extension also includes a test of whether a given constraint has been used in the derivation of some sentence. We will make the additional assumption here that this test of whether a sentence has been used in a derivation takes less time than the actual derivation. This means the complexity of the subsumption algorithm remains unchanged by the extension for input-output constraints.

The next extension concerns the properties which are effectively propositions and thus, could be handled as constraints. The extension for the algorithm was described in figure 5.5. The subset test mentioned can be performed in time linear in the number of properties. If we count properties as constraints, this extension again does not affect the complexity of the algorithm.

The final extension deals with capabilities and tasks described as performable actions. The extension to the basic algorithm was described in figure 5.6 in section 5.2.3. What this adds to the algorithm described thus far are two potential capability instantiations. The algorithm for instantiation of CDL descriptions, be they capabilities or tasks, was described in figure 5.7. At the heart of this algorithm is the amending of sets of parameter specifications (cf. figure 5.8). Let there be p parameter specifications, each no longer than f symbols. Adding or deleting parameter specifications from a set may take time linear in p and unifying two role fillers may require time f^2. Thus, the time complexity for amending a set of parameter specifications is in $\mathcal{O}(p(p+f^2))$. For simplicity we shall now assume that f is a constant limiting the length of parameter specifications and that there are more constraints in a CDL description than parameter specifications. Thus, the complexity for the function *amend* is in $\mathcal{O}(c^2)$ where c is the number of constraints involved. This is also the complexity for instantiating a CDL description from a single, other description. However, in a hierarchy of capabilities of depth h a CDL description may inherit from up to h CDL descriptions

due to the single inheritance mechanism.

This gives us the following result for the time complexity of the final version of the capability subsumption test given in figure 5.9 under the assumptions outlined above:

> Let c be the number of constraints involved in the evaluation of task $\mathcal{T}$ against capability $\mathcal{C}$. Let the hierarchy of capabilities on which $\mathcal{T}$ and $\mathcal{C}$ may be based be no deeper than h. Let the evaluation of a single constraint against a knowledge base be in $\mathcal{O}(e(\mathcal{L}))$ where $\mathcal{L}$ is the language in which the constraints are expressed. Then the overall complexity of the algorithm for the basic subsumption test described in figure 5.9 is in $\mathcal{O}(e(\mathcal{L})c + c^2h)$.

6.3.2 Scaling Issues

The problem with the analysis thus far is that it is only for the performance of a single subsumption test. We know, however, that such a test is very fast in practice and that the parameters that affect its complexity are not the ones that will increase in a more realistic scenario. The parameter that will increase, presumably by several orders of magnitude, in a more realistic scenario is the number of problem-solving agents available, and thus the number of capabilities the broker knows about.

Our implementation is very inefficient in this respect. Capabilities are stored in a linear list and when the broker attempts to find an agent capable of solving a particular problem, it goes through this list one by one applying the subsumption test once for every capability in the list. For a capability descriptions this results in $\mathcal{O}(a)$ capability subsumption tests, which is not satisfactory. The reason why we have implemented such a rather trivial algorithm at this point is simply due to the fact that there are only very few capabilities which have been described to the broker in our scenarios and developing large scenarios where the number of capabilities would make a difference were beyond the scope of this thesis.

Furthermore, there are well known techniques for addressing this problem, which is closely related to the problem of memory organisation described in [Charniak and McDermott, 1985, page 396]. The basic approach there is to find necessary criteria that can be easily evaluated and which indicate whether a call to *evaluate* might succeed. One simple but highly efficient technique is to extract all the predicates from the output constraints of a capability description. We could store these in a hash table that associates a predicate with all the capability descriptions using this predicate in one of its output constraints. Now, a task description mentioning some predicates in its output constraints could only be addressed by a capability which mentions at least one of these predicates in its output constraints. Since hash table access is very fast (depending on the hash function), this method would significantly improve the complexity of our capability retrieval algorithm. Of course, there can still be a large number of capabilities mentioning a given predicate, but in this case one can use the arguments to the predicate that are not variables to refine hashing (cf. [Charniak and McDermott, 1985, section 7.2.1]).

Another interesting question one might ask in this context is how the expressiveness used in the state language affects the performance of capability retrieval. It should be clear at this point that this does not depend on the number of capabilities the broker knows but only affects the time for a single capability subsumption test. Specifically, greater expressiveness in the state description language may only increase the function $e(\mathcal{L})$, i.e. the time taken up by the function *evaluate*. The exact amount by which greater expressiveness increases the complexity of this function depends on the state description languages in question. Thus, there is a price to pay for expressiveness here, but this price does not depend on the number of capabilities known by the broker and is no worse than in any other reasoning task involving the more expressive language.

Similarly, we can ask how flexibility affects the performance of capability retrieval. Again, it should be clear that this is not a scaling issue as the number

of capabilities is independent of their flexibility. Flexibility does, however, affect the performance of a single capability subsumption test. This is because our algorithm cannot rely on the fact that certain functionality is actually provided by the state description language, and thus, it uses reflective reasoning to test for the existence of this functionality every time it needs to use it. This takes up a constant amount of time and is done for every call to *evaluate*, i.e. the overall cost is $\mathcal{O}(c)$ and does not affect the complexity of the subsumption test. The only time when flexibility may make a noticeable difference in performance is when the broker receives a new capability description involving a new state language and this language has to be loaded from a remote host. This may take some time, but obviously does not affect the capability retrieval time.

To summarise, expressiveness is a major factor affecting the complexity of the capability subsumption algorithm and thus, the capability retrieval time, but flexibility causes hardly any increase in this complexity. The important parameter we would expect to scale up is the number of capabilities known to the broker. However, the complexity of the subsumption algorithm does not depend on this parameter.

Chapter 7

Expressiveness of CDL

At this point we have described our capability description language CDL *which can be used to represent the content of messages required for capability brokering. We also have shown how* CDL *can be used for capability retrieval. Our aim now is to show that* CDL *has two desirable properties: it is expressive and flexible. The next step towards this goal will be to describe what we mean by expressiveness. The contribution of this chapter will be a description of our concept of expressiveness of action representations and its application to* CDL. *A comparison of the expressiveness of* CDL *with that of other action representations will follow in section 9.2.1.*

7.1 Why more Expressiveness?

In this section we will discuss why expressiveness is one of the properties we want in CDL. *We will use the more complex scenarios (from section 3.3) to illustrate this point along with providing a more theoretical argument.*

Up to this point we have simply *assumed* that expressiveness is a desirable property for our capability description language. But clearly, expressiveness comes at a price: usually the complexity of the algorithms required to reason

about the language increases with the expressiveness of the language. Thus, it is important to make sure that a language provides only as much expressiveness as required for a given class of problems in order to maintain heuristic adequacy. Of course, it is equally important to provide sufficient expressiveness in order to be able to represent problems in this class with epistemological adequacy. We will now outline why we believe that capability descriptions of realistic agents require an expressive action representation language, like CDL.

7.1.1 The Expressiveness Scenario Revisited

One of the scenarios introduced in section 3.3 exactly addresses the question of why we need more expressiveness; the *expressiveness scenario* (example 3.2). The very idea behind this scenario is to illustrate the need for an expressive capability description language. We will now look at this scenario again to see whether we really need this expressiveness.

In the expressiveness scenario, the fact that the first hospital only treats people from Barnacle, Calypso, or Delta was represented as a disjunctive input constraint. Expressing this constraint in a less expressive representation, e.g. one that allows only conjunctions of literals, would be possible by representing the capability as three *separate capabilities*, one for each disjunctive case. A second applicability condition in this scenario was for the ambulance to have snow chains if there is ice or snow on the roads. This condition was represented as an input constraint that is an implication with a disjunctive precondition. But an implication can be rewritten as a disjunction in first-order logic and thus, the same technique as before can be applied.

This technique of replacing one capability description with disjunctive input constraints by several capability descriptions raises *problems* though. Firstly, if one kept splitting capabilities with disjunctive input constraints then a capability description with k disjunctive preconditions, each having n disjuncts, would result in n^k separate capability descriptions. There is a second, more worrying concern

in this example though: we only know that the weather is bad, meaning there is either ice or snow on the roads, but we do not know which. Given this knowledge, the splitting of capability descriptions according to disjunctive input constraints is not epistemologically adequate, as the separate capability descriptions cannot be applied whereas the capability with the disjunctive input constraint is applicable.[1] Thus, the expressiveness scenario does indeed require at least the expressiveness of disjunctive input constraints.

Disjunctions in input constraints are not the only reason why we need an expressive capability description language though. Looking at the capability description of the **h1**-agent in the initial scenario (cf. section 4.2.5), the last input constraint was given as:

```
(Has Location ?person Pacifica)
```

This only states that the injured person has to be on Pacifica. However, to actually apply the described capability the **h1**-agent will need to know where exactly to find the injured person, i.e. it needs to know a more precise location than "on Pacifica." Thus, we could easily argue that *knowledge preconditions* (in this example the knowledge where to find the injured person) and thus the ability to reason about knowledge are required here. Reasoning about knowledge can be performed in modal logics (cf. section 2.2.2). Thus, one could argue that the expressiveness of modal logics is required in this scenario, which is greater than what we have used this far.

Furthermore, while the input constraint which states that the person has to be on Pacifica is too general with respect to the person's location, the input constraint that we need to know where exactly the injured person is, is too specific. In fact the **h1**-agent only needs to know roughly where the injured person is, not exactly where. For example, within a city an address may be sufficient as the patient's location; at the power plant we might need more specific information

[1] Note that, for example, a contingency planner could deal with this situation, but the resulting plan would be rather awkward with two identical branches.

depending on the size of the plant. However, to apply its capability the **h1**-agent will not need to know where exactly in a flat with a given address the patient is. But what does that mean for CDL? It means that the representation will have to be expressive enough to somehow represent *qualitative input constraints on space*, i.e. it would require greater expressiveness again.

It is not difficult to think of further examples that require the representation of further different kinds of knowledge which, in turn, might require more expressiveness. Thus, for a generic capability description language to be able to represent all of the above examples, we *need expressiveness*.

7.1.2 Conciseness of Capability Descriptions

The above examples have illustrated the need for a capability description language to be expressive. A property we expect of descriptions of capabilities in such a language is *conciseness*. In this section we will argue that the need for conciseness in a capability description is a contributing factor for the need for expressiveness in a capability description language. The need for conciseness in a capability description has at least two reasons:

- Conciseness of a capability description implies a relatively short expression representing the capability. Shorter expressions mean less communication overhead and thus, may facilitate more efficient overall problem-solving behaviour.

- Conciseness also means the description should not contain unnecessary detail or redundant information. It is likely that such an expression can be reasoned about more efficiently, resulting in more efficient overall problem-solving behaviour again.

For example, let us look at an agent acting in the *Blocks World* with the four basic capabilities described by the actions stack, unstack, pick-up, and put-down as defined in [Nilsson, 1980, chapter 7]. Another capability of this agent

is to perform any sequence of these actions. While this presents a capability description for this agent, it is arguably not a concise description. It is not short because it contains the full descriptions of all the primitive actions the agent can perform. It also contains the information necessary to derive a sequence of actions necessary to achieve a given goal which is not necessary for brokering, i.e. it contains redundant information.

There are at least *two principal ways* of achieving conciseness in a capability description: abstraction in the description and expressiveness in the language.

Abstraction has been employed in many areas of AI such as planning, machine learning, or natural language understanding. [Russell and Norvig, 1995, page 62] describe abstraction as the process of removing detail from a representation. Note that this is much stronger than what we have described above: a concise representation should not contain unnecessary detail or redundant information. Thus, abstraction is a process that can be used to achieve conciseness in a representation. For example, a more abstract capability description for the Blocks World agent mentioned above would be that it can achieve a state in which any block is to be on top of another.

Expressiveness of a language can be described as the potential to represent certain circumstances in this language that cannot adequately be represented in a less expressive language. But expressiveness not only gives us the possibility to say more, it may also give us the possibility to represent the same facts more concisely. For example, a capability with a disjunctive input constraint may be replaceable by several capability descriptions that are identical except for one of their input constraints. Even if this replacement does not lead to problems as described above, the expression of the capability with a disjunctive input constraint is obviously more concise. Thus, a more expressive language can allow for a more concise representation.

While the above shows that expressiveness of a description language is one possible way of achieving conciseness of representations in this language, it does

not mean that we *necessarily* need more expressiveness. However, if we want our capability description language to be generic, we have to provide for the cases where such expressiveness is required.

7.2 Expressiveness of AR Languages

In this section we will introduce a (fairly lightweight) theoretical framework that defines what we mean by the expressiveness of action representation languages and allows for comparisons of such formalisms.

Up to now we have used the term *expressiveness* in a rather loose fashion. Thus, one of the main claims of this thesis, that CDL is an expressive action representation, is rather ill-defined at this point. Perhaps surprisingly, there does not seem to exist an agreed definition of what it means for an action representation to be expressive. Informally, one can define expressiveness as follows:

Definition 7.1 (Expressiveness) *A knowledge representation language is* **expressive** *if it is possible to represent certain circumstances in this language that cannot be adequately represented in a less expressive language.*

This notion has been used in [Baader, 1996] to formalise the concept of *expressiveness for terminological knowledge representation languages*.[2] Baader also claims that the underlying idea can be used to define the expressiveness of other representation languages. Thus, we shall now have a brief look at his definition of expressiveness.

7.2.1 Expressiveness of KR Languages

The first step towards a formalisation of the above definition of expressiveness is the *definition of the knowledge representation languages* one wants to consider. To this effect Baader defines the class of KR1 languages as [Baader, 1996, page 40]:

Definition 7.2 (KR language based on first-order logic) *Assume that we have countably many variable symbols and countably many predicate symbols of*

[2] By a terminological KR language, Baader means any language based on Brachman's ideas about concept structure such as KL-ONE [Brachman, 1979, Brachman and Schmolze, 1985].

any arity. In addition we assume that we have a binary predicate symbol for equality which has to be interpreted as equality in all models. Let FO denote the set of all first-order formulae that can be built out of these symbols. A **KR1 language** *(KR language based on first-order predicate logic)* **L** *consists of:*

1. *A subset L of the power set of FO, i.e. a set of sets of formulae.*
2. *A model-restriction function Mod_L that maps each set $\Gamma \in L$ to a subclass $Mod_L(\Gamma)$ of all first-order models of Γ.*

According to this definition, a *KR1 language* basically consists of a language in which every expression must be a set of first-order formulae and a model-restriction function that maps expressions in this language into their models. In terminological logics, the languages Baader is interested in, a set of concept descriptions $\Gamma \in L$ is often called a T-Box.

In this definition every expression Γ in the *language* L of a KR1 language **L** must be a set of formulae in first-order predicate logic (FOPL). Allowing the syntax of FOPL only here might seem rather restrictive at a first glance. However, one way of defining the semantics for a new knowledge representation language L' is to define an equivalence semantics [Winston, 1992, page 20]. This defines how a sentence in the new language can be transformed into another language which already has an accepted semantics, thus indirectly adopting the semantics of this other language for the new language L'. Any language that has an equivalence semantics based on FOPL can thus be considered a language of a KR1 language. Languages that cannot be transformed into FOPL are excluded though.

The second part of a KR1 language, the *model-restriction function Mod_L*, defines the models of an expression $\Gamma \in L$. It maps an expression Γ to only a subclass of all first-order models of Γ to allow for T-Boxes that contain cycles and require a fixed-point semantics [McDermott and Doyle, 1980].

Now, given this definition of the languages one wants to consider, the next step towards a formalisation of definition 7.1 is to define when an expression in

L_1 "represents certain circumstances that cannot be adequately represented" by an expression in L_2. The basic idea in Baader's work now is to define that two expressions $\Gamma_1 \in L_1$ and $\Gamma_2 \in L_2$ express the same circumstances, or concepts in the case of terminological logics, if they have the same models. Baader's actual definition is more general than this though, allowing for the renaming of predicate symbols and the presence of auxiliary predicates in L_2 [Baader, 1996, page 41]:

Definition 7.3 (Model equality modulo ψ-embedding) *For an element Γ of FO let $Pred(\Gamma)$ denote the set of all predicate symbols occurring in Γ. Assume that we have a mapping $\psi : Pred(\Gamma_1) \longrightarrow Pred(\Gamma_2)$, and models M_1, M_2 of Γ_1, Γ_2 respectively. The elements of $Pred(\Gamma_2)$ that are outside the range of ψ are auxiliary predicate symbols. We say that M_1 is embedded in M_2 by ψ ($M_1 \subset_\psi M_2$) iff all R in $Pred(\Gamma_1)$ satisfy $R^{M_1} = \psi(R)^{M_2}$.*[3] *Equality of classes of models modulo ψ-embedding is defined by extensionality, i.e. $Mod_{L_1}(\Gamma_1) =_\psi Mod_{L_2}(\Gamma_2)$ iff*

- *for all M_1 in $Mod_{L_1}(\Gamma_1)$ there exists M_2 in $Mod_{L_2}(\Gamma_2)$ such that $M_1 \subset_\psi M_2$, and*
- *for all M_2 in $Mod_{L_2}(\Gamma_2)$ there exists M_1 in $Mod_{L_1}(\Gamma_1)$ such that $M_1 \subset_\psi M_2$.*[4]

This definition tells us what it means for two sets of models to be equivalent. Essentially, it defines two models as equivalent if they are equal subject to a function renaming the symbols and subject to the omission of auxiliary predicates. One can now use the model-restriction function that maps expressions into sets of models to define when two expressions represent the same circumstances, namely if their models as defined by the model-restriction functions are equivalent. This can be formalised as follows [Baader, 1996, page 41]:

[3] Although not defined in Baader's paper, it is fairly obvious that R^M is intended to denote the extension of the predicate R in the model M.

[4] Note that Baader's definition of $=_\psi$ is not symmetric.

Definition 7.4 (Expressive power of KR1 languages) *Let* $\Gamma_1 \in L_1$ *and* $\Gamma_2 \in L_2$ *for KR1 languages* $\mathbf{L}_1$ *and* $\mathbf{L}_2$.

1. Γ_1 *can be expressed by* Γ_2 *iff there exists* $\psi : Pred(\Gamma_1) \longrightarrow Pred(\Gamma_2)$ *such that* $Mod_{L_1}(\Gamma_1) =_\psi Mod_{L_2}(\Gamma_2)$.

2. $\mathbf{L}_1$ *can be expressed by* $\mathbf{L}_2$ *iff for any* $\Gamma_1 \in L_1$ *there exists* $\Gamma_2 \in L_2$ *such that* Γ_1 *can be expressed by* Γ_2*—i.e. iff there is a mapping* $\chi : L_1 \longrightarrow L_2$ *such that* $\Gamma_1 \in L_1$ *can be expressed by* $\chi(\Gamma_1)$.

3. $\mathbf{L}_1$ *and* $\mathbf{L}_2$ **have the same expressive power** *iff* $\mathbf{L}_1$ *can be expressed by* $\mathbf{L}_2$ *and vice versa.*

The first part of this definition formalises when two expressions in different languages express the same circumstances. The second part then generalises this notion for languages: one language can be expressed in another if every expression in the former language can be expressed in the latter. Notice that if $\mathbf{L}_1$ can be expressed by $\mathbf{L}_2$ then $\mathbf{L}_2$ is at least as expressive as $\mathbf{L}_1$. Finally, the third part of this definition defines when two KR1 languages are equally expressive. Notice that this definition does not define an absolute measure of the expressiveness of a KR language. It does, however, group KR1 languages into equivalence classes and defines a partial order on those. The limit is first-order logic itself which is the most expressive language in this framework.

7.2.2 Expressiveness of Action Representations

Baader's definition of expressiveness is only applicable to KR1 languages. In essence, these languages are what we have called state representation languages in section 4.2.3.1. *Action representations* are usually not described as languages that fit into this category (cf. section 2.3.1). Hence, the definition of expressiveness for KR1 languages is not applicable to action representations directly.

In this section we will present our definition of expressiveness for action representations. The first step towards this definition must be a definition of the class of *action representations* we want to consider:

Definition 7.5 (AR1 language) *An* **AR1 language** *(Action Representation based on first-order predicate logic)* **L** *is a quadruple* (A, S, Mod_S, Rel_A) *where:*

- A *is the (decoupled) action description language of* **L***;*
- S *is the state description language of* **L***;*
- *the model-restriction function* Mod_S *maps each state description* $s \in S$ *to a set* $Mod_S(s)$ *of first-order models;*
- *the action definition function* Rel_A *maps each action description* $a \in A$ *to a binary relation* $Rel_A(a)$ *on state descriptions, i.e.* $Rel_A(a) \subseteq S \times S$*.*

This definition reflects our view of *decoupled* action representations presented in section 4.2.3.2, i.e. an action representation consists of a decoupled action description language (A in the above definition) and a state description language (S). Note that the above definition does not define what either language has to look like.

The *state description language* S together with the model-restriction function Mod_S is almost identical to a KR1 language. The only difference is that expressions in S do not have to be sets of formulae in FOPL. However, the function Mod_S has to map every expression in S into a set of first-order models. Thus, while the form of the state description language of an AR1 language is not defined in the above definition, we do require that the semantics of this language has to be definable in terms of first-order models.

The *decoupled action description language* A is the language that allows us to describe actions. Note that in general an expression $a \in A$ is intended to define a

set of actions, not just a single action.[5] The underlying idea here is that actions transform one state into another.[6] Thus, the action description language A can be used to define a binary relation on states, where states are described in the state description language S. The action definition function Rel_A defines how to obtain this relation from a description of a set of actions $a \in A$. Note that this definition does not impose any restrictions on the action description language itself other then the fact that it can be used to define a binary relation on states.

To *illustrate* the above definition we shall now informally describe a propositional version of the STRIPS representation as defined in [Nilsson, 1980, chapter 7] as an AR1 language STRIPS $= (A_s, S_s, Mod_{S_s}, Rel_{A_s})$:

- An expression in the action description language A_s is a set of STRIPS operators. Each operator consists of an action identifier, a set of variables specifying parameters, a precondition expression, an add expression, and a delete expression. Each of these expressions must be an expression in the state description language S_s.

- An expression in the state description language S_s is a set of function-free literals over a given set of predicate, constant, and variable symbols. An expression in the state description language which occurs inside an operator may only contain variables from this operator's parameters.

- The model-restriction function Mod_{S_s} maps an expression in S_s, i.e. a set of literals, into the models of the conjunction of the literals in the set.

- Finally, the action definition function Rel_{A_s} defines two states as related if there is an operator such that the precondition expression is a subset of the first state and the first state modified by the add and delete expressions is a equal to the second state.

[5] Thus, we shall use the term "set of actions" to refer to an action description $a \in A$", i.e. an expression in the action description language A.

[6] We shall use the term "state" to refer to a state description $s \in S$", i.e. an expression in the state description language S.

This gives us a formal definition of an AR1 language. The basic idea now is to say that *two action descriptions in different AR1 languages express the same actions if and only if for every state in one language there is a somehow equivalent state in the other language and the states that can be reached within each representation are again somehow equivalent.*

To formalise this notion we first need to define what we mean by two states in two different state description languages being *somehow equivalent.* As mentioned before, state description languages together with their model-restriction functions are quite similar to KR1 languages and thus, we can use the definition of model equality for sets of models (definition 7.3) to define the equivalence of states which are mapped to sets of models by the model-restriction function.

This leaves us a need to formalise the concept of state *reachability* before we can define the expressiveness of action representations. Intuitively, a state is reachable from another state if there is an action or a sequence of actions that takes us from one to the other.

Definition 7.6 (Reachability in L) *Let* $\mathbf{L} = (A, S, Mod_S, Rel_A)$ *be an AR1 language,* $s \in S$ *a state description, and* $a \in A$ *a set of actions. Then the set* $\mathcal{R}^1(s, a)$ *of all states* **reachable in one step** *from* s *through* a *is the set of all states* s' *for which* (s, s') *is in* $Rel_A(a)$*. The set* $\mathcal{R}(s, a)$ *of all states* **reachable** *from* s *through* a *is the union of* $\mathcal{R}^1(s, a)$ *and all states* s'' *for which there is a state* s' *that is known to be in* $\mathcal{R}(s, a)$ *and* (s', s'') *is in* $Rel_A(a)$*.*

The set $\mathcal{R}(s, a)$ is the set of all states that can be reached from the state s by performing a sequence of actions, where each action in the sequence must be described in a. Thus, $\mathcal{R}(s, a)$ is the possibly infinite state space of all states reachable from s through a. If two sets of actions define the same state space from a given state then these sets of actions can be considered equivalent in this state. If they are equivalent in all states then we can say that they express the same set of actions.

We are now in a position to formally define when two sets of actions represented in two different AR1 languages represent the same actions:

Definition 7.7 (Expressive power of AR1 languages) *Let a_1 and a_2 be two sets of actions described in action representations A_1 and A_2 for AR1 languages $\mathbf{L}_1$ and $\mathbf{L}_2$ respectively. Then:*

1. a_1 **can be expressed by** a_2 *if and only if for all states s_1 in S_1 there exists a state s_2 in S_2 and a function $\psi : Pred(s_1) \longrightarrow Pred(s_2)$ such that:*

   - $Mod_{S_1}(s_1) =_\psi Mod_{S_2}(s_2)$ *and*
   - $\mathcal{R}(s_1, a_1) =_\psi \mathcal{R}(s_2, a_2)$.

2. *AR1 language* $\mathbf{L}_1$ **can be expressed by** *AR1 language* $\mathbf{L}_2$ *if and only if for any set of actions a_1 described in A_1 there exists a set of action a_2 described in A_2 such that a_1 can be expressed by a_2—i.e. if there is a mapping $\chi : A_1 \longrightarrow A_2$ such that a_1 can be expressed by $\chi(a_1)$.*

3. $\mathbf{L}_1$ *and* $\mathbf{L}_2$ **have the same expressive power** *iff* $\mathbf{L}_1$ *can be expressed by* $\mathbf{L}_2$ *and vice versa.*

The first part of this definition just formalises what we have said informally before: that two sets of operator descriptions express the same actions if for every state in one language there is an equivalent state in the other language and the states that can be reached within each representation are also equivalent. The second part of this definition extends this concept to AR1 languages, i.e. an AR1 language can be expressed by another if every set of actions expressible in the first language can also be expressed in the second. Finally, if two AR1 languages can express each other they have the same expressive power.

7.2.3 Polynomial Transformability

An alternative definition of *expressive equivalence of planning formalisms* can be found in [Bäckström, 1995]. We shall now have a brief look at this definition to see how it compares with our concept of expressiveness as described above.

Bäckström defines expressive equivalence based on the *general planning problem* and its solutions [Bäckström, 1995, page 24]:

Definition 7.8 (General planning problem) *Given a planning formalism X, the* **(general) planning problem** *in X (***X-GPP***) consists of a set of instances, each instance* Π *having an associated set* $Sol(\Pi)$ *of solutions.*

A solution to a given planning problem in a given formalism needs to be defined for every formalism individually. Based on this concept of a general planning problem *expressive equivalence* is defined as follows [Bäckström, 1995, page 25]:

Definition 7.9 (Expressive equivalence) *Given two planning formalisms X and Y, we say that X is at least as expressive as Y with respect to plan existence if* **Y-GPP** $\leq_p$ **X-GPP**, *i.e.* **Y-GPP** *polynomially transforms*[7] *into* **X-GPP**. *Further, X and Y are equally expressive with respect to plan existence iff both* **X-GPP** $\leq_p$ **Y-GPP** *and* **Y-GPP** $\leq_p$ **X-GPP**.

Essentially, this definition states that two planning formalisms are equally expressive if every instance of a planning problem expressed in X polynomially transforms into an instance of a planning problem expressed in Y and vice versa. Furthermore, every problem in X must have a solution if and only if the corresponding problem in Y also has a solution.

An interesting aspect of this definition is that it is based solely on *plan existence*, i.e. it does not require the plans in $Sol(\Pi)$ to be somehow equivalent. It

[7] See [Garey and Johnson, 1979, section 2.5] for a definition of polynomial transformability in general.

only matters whether the set $Sol(\Pi)$ is empty or not for expressive equivalence. This is consistent with our definition of expressive equivalence (cf. definition 7.7). The classical planning problem [Tate *et al.*, 1990, page 28] is given as an initial state description, a goal state description, and a set of operator schemata. A solution is a sequence of operator instances that transforms the initial state into the goal state. It is easy to see that such a solution exists (as required for definition 7.9) if and only if the goal state is reachable (cf. definition 7.6) from the initial state. Thus, both definitions of expressiveness do not place any constraints on the sequence of operators required to reach the goal state.

One difference between the two definitions of expressiveness lies in the requirement of a *formal semantics* for an AR1 language in our definition. In contrast, Bäckström only requires the set of solutions $Sol(\Pi)$ for a given planning problem to be defined. While this broadens the applicability of his definition[8] it also provides little insight as to what exactly an action representation is. For example, the concept of a decoupled action representation cannot be incorporated into definition 7.9.

The most interesting difference between Bäckström's definition of expressiveness and our own is that he requires instances of planning problems to polynomially transform into each other. The transformation that is implicit in this requirement corresponds to the mapping $\chi : A_1 \longrightarrow A_2$ of definition 7.7, only that we did not require this mapping to be computable in polynomial time. The intension behind our (and presumably Baader's) definition of expressive equivalence is that two languages are equally expressive if one can say the same things in both languages, however complicated the translation process might be. Bäckström's argument for this additional requirement is that polynomial transformability implies that transformaion does not change the complexity class of the underlying problem. This seems sensible, and it is questionable whether an expression that grows exponentially in the translation process should still be considered as ex-

[8] Bäckström applied his definition of expressiveness to compare the expressiveness of various variants of propositional STRIPS.

pressing the same content.

Therefore, we shall retain our definition of expressiveness as given in definition 7.7, but add the following extension:

Definition 7.10 (Poly-expressive equivalence) *Let a_1 and a_2 be two sets of actions described in action representations A_1 and A_2 for AR1 languages $\mathbf{L}_1$ and $\mathbf{L}_2$ respectively. Then:*

1. *AR1 language $\mathbf{L}_1$* **can be poly-expressed by** *AR1 language $\mathbf{L}_2$ if and only if for any set of actions a_1 described in A_1 there exists a set of action a_2 described in A_2 such that a_1 can be expressed by a_2—i.e. if there is a mapping $\chi : A_1 \longrightarrow A_2$ such that a_1 can be expressed by $\chi(a_1)$ and χ is computable in polynomial time.*

2. *$\mathbf{L}_1$ and $\mathbf{L}_2$ are* **equally poly-expressive** *iff $\mathbf{L}_1$ can be poly-expressed by $\mathbf{L}_2$ and vice versa.*

7.3 CDL: An AR1 Language

In this section we will present a formal semantics for CDL *that can be used to compare its expressiveness to that of other action representation languages. Such a comparison will follow in section 9.2.1.*

7.3.1 The State Description Language

One of the components of an AR1 language as defined in definition 7.5 is the *state representation language* S. We have described CDL in chapter 4 as a decoupled action representation language, i.e. as an action representation language into which arbitrary state representations can be plugged to form a complete action representation. To define CDL as an AR1 language it is necessary to describe at least one state language that can be used within the decoupled action representation and, as before, we shall use first-order logic for this purpose. This choice also allows us to view the state representation together with the model-restriction function as a KR1 language.

A *syntax* of FOPL has already been defined in figure 4.2. This formalism will be the basis of the first-order language that we will use as the state representation S here. The only change we need to make in the syntax concerns the predicate, function, constant, and variable symbols of the language: these need to be defined in a KR1 language (cf. definition 7.2). Since our definition of expressiveness allows for the embedding of models with a function ψ that effectively renames these symbols, the names of these symbols in the definition of the syntax do not matter. The revised syntax of FOPL which will be used as state representation for CDL in this chapter is given in figure 7.1.

Note that in this definition of the state representation we have defined the syntactical category of *terms* as part of the state description language, whereas it was defined as part of the action description language in chapter 4. This is necessary to define the state language independent from the action representation.

```
<formula>    ::= ( <quant> <c-form> ) | <c-form>

<quant>      ::= ( <quantifier> <varspec>+ )
<quantifier> ::= forall | exists
<varspec>    ::= <variable> |
                 ( <variable> <constant> )

<c-form>     ::= <literal> |
                 ( not <formula> ) |
                 ( and <formula> <formula>+ ) |
                 ( or <formula> <formula>+ ) |
                 ( implies <formula> <formula> ) |
                 ( iff <formula> <formula> ) |
                 ( xor <formula> <formula> ) |

<literal>    ::= <predicate> |
                 ( = <term> <term> )
                 ( <predicate> <term>+ )

<term>       ::= <constant> | <variable> |
                 ( <function> <term>+ ) |

<predicate>  ::= P1 | P2 | P3 | ...
<function>   ::= f1 | f2 | f3 | ...
<constant>   ::= c1 | c2 | c3 | ...
<variable>   ::= ?v1 | ?v2 | ?v3 | ...
```

Figure 7.1: Syntax of the state representation S

```
<cdl-descr> ::= ( <ctype>
                    :state-language fopl
                    :action <name>
                    :isa <name>
                    :properties ( <name>+ )
                    :input ( <param-spec>+ )
                    :output ( <param-spec>+ )
                    :input-constraints ( <constraint>+ )
                    :output-constraints ( <constraint>+ )
                    :io-constraints ( <constraint>+ )

<ctype> ::= capability | task

<param-spec> ::= ( <name> <term> )
<term>       ::= <constant> | <variable> |
                 ( <function> <term>+ ) |
<function>   ::= f1 | f2 | f3 | ...
<constant>   ::= c1 | c2 | c3 | ...
<variable>   ::= ?v1 | ?v2 | ?v3 | ...

<constraint> ::= <formula>
```

Figure 7.2: Syntax of the action representation A

7.3.2 The Action Description Language

The next step towards the definition of CDL as an AR1 language is the definition of the *action description language* A. Again, the syntax of the A can be based on the syntax of CDL described in chapter 4. The syntax is repeated in figure 7.2 here for convenience. Note that it contains the same modification as the state language: the function, constant, and variable symbols are now defined in the syntax.

Also, the syntactical category of *terms* is now defined as part of the state and as part of the action representation. Both definitions are identical, based on the same set of function, constant, and variable symbols. This is not actually necessary as terms in the action representation may, and usually will, only use

a subset of the symbols listed in the state representation. However, to keep the definition simple we have defined them with the same sets of symbols.

A further difference between this definition of the action representation and the syntax of CDL described in figure 4.4 is that the state language `fopl` is now *fixed* in the language: the value following the keyword `:state-language` is `fopl` and the syntactical category `<constraint>` is defined as being a `<formula>`, i.e. an expression in the state description language FOPL as described above.

7.3.3 The Model-Restriction Function

The *model-restriction function* Mod_S maps each state description $s \in S$ to a set $Mod_S(s)$ of first-order models, thus effectively defining the semantics of the state description language. The choice of FOPL as our state representation in CDL makes the definition of Mod_S relatively straight forward. We shall only describe the semantics briefly and informally here based on a more formal definition in [Shanahan, 1997, sections 2.2 and 2.3].

The semantics of FOPL is based on the concept of an *interpretation*. An interpretation consists of a non-empty set D of objects that is the domain for this interpretation. An interpretation also consists of two functions F and P. F maps all constant and variable symbols into elements in D and all n-ary function symbols into functions from D^n to D. P maps all n-ary predicate symbols to the extension of this predicate which is a subset of D^n. For an interpretation M one can then define the following abbreviations:

- if x is a predicate symbol, constant symbol, or function symbol, let M[x] be P(x), F(x), or F(x) respectively;
- if f is a function symbol and $t_1, \ldots, t_n$ are terms, then let M[$(ft_1 \ldots t_n)$] be M[f](M[t_1], ..., M[t_n]).

In other words, M maps every term into the domain object it stands for and

predicates to their extension. Based on the above, one can define the *satisfaction relation* between an interpretation M and a well-formed formula in S as follows:

- M satisfies $(P\ t_1 \ldots t_n)$ if $\langle \mathrm{M}[t_1], \ldots, \mathrm{M}[t_n] \rangle \in \mathrm{M}[P]$ for predicate symbol P and terms $t_1, \ldots, t_n$;
- M satisfies $(=\ t_1\ t_2)$ if $\mathrm{M}[t_1] = \mathrm{M}[t_2]$ for terms t_1 and t_2;
- M satisfies (not F) if M does not satisfy F for formula F;
- M satisfies (and $F_1 \ldots F_n$) if for all $i \in \{1 \ldots n\}$ M satisfies F_i for formulae $F_1 \ldots F_n$;
- M satisfies (or $F_1 \ldots F_n$) if there exists $i \in \{1 \ldots n\}$ such that M satisfies F_i for formulae $F_1 \ldots F_n$;
- M satisfies (implies $F_1\ F_2$) if M satisfies F_1 implies that M satisfies F_2 for formulae F_1 and F_2;
- M satisfies (iff $F_1\ F_2$) if M satisfies F_1 if and only if M satisfies F_2 for formulae F_1 and F_2;
- M satisfies (xor $F_1\ F_2$) if either M satisfies F_1 or M satisfies F_2, but not both, for formulae F_1 and F_2;
- M satisfies ((forall $vs_1 \ldots vs_n$) F) where vs_i is either v_i or $(v_i\ c_i)$, if for all interpretations M' that agree with M except possibly in the interpretation of $v_1 \ldots v_n$, M' satisfies F and maps all variables v_i specified as $(v_i\ c_i)$ to domain objects of type c_i;
- M satisfies ((exists $vs_1 \ldots vs_n$) F) where vs_i is either v_i or $(v_i\ c_i)$, if there is some interpretations M' that differs from M only in the interpretation of $v_1 \ldots v_n$, M' satisfies F and maps all variables v_i specified as $(v_i\ c_i)$ to domain objects of type c_i;

This allows us to finally define the model-restriction function Mod_S that maps each state description $s \in S$ to the set $Mod_S(s)$ of interpretations M that satisfy s. These interpretations are also called the *models* of s.

7.3.4 The Action Definition Function

The last component of an AR1 language that remains to be defined for CDL is the *action definition function* which maps a set of capability descriptions a in CDL to a binary relation on states $Rel_A(a) \subseteq S \times S$. To define this relation we will effectively use the match conditions as described in definition 5.3: capability $\mathcal{C}$ **subsumes** task $\mathcal{T}$ if and only if there exists a substitution σ such that:

$$C_I^{\mathcal{T}} \models \sigma(C_I^{\mathcal{C}}) \qquad \text{(input match condition)}$$

and

$$\sigma(C_O^{\mathcal{C}}) \wedge \sigma(R^{\mathcal{C}}) \models C_O^{\mathcal{T}} \qquad \text{(output match condition)}$$

and

$$\forall n \in \{1 \ldots m_{\mathcal{C}}\} : \text{if } \sigma(C_O^{\mathcal{C}}) \wedge (\sigma(R^{\mathcal{C}}) - \sigma(R_n^{\mathcal{C}})) \not\models C_O^{\mathcal{T}} \text{ and } \sigma(C_O^{\mathcal{C}}) \wedge \sigma(R^{\mathcal{C}}) \models C_O^{\mathcal{T}}$$
$$\text{then } C_I^{\mathcal{T}} \models \sigma(L_n^{\mathcal{C}}) \qquad \text{(input-output match condition)}$$

We will say that a pair of state descriptions is related by a set of actions, i.e. $(s_1, s_2) \in Rel_A(a)$, if there is a capability description in a that can be instantiated to $\mathcal{C}$ (cf. definition 5.6) such that the capability $\mathcal{C}$ subsumes the task $\mathcal{T}$ that corresponds to the state transition (s_1, s_2).

The *task* $\mathcal{T}$ that corresponds to the state transition (s_1, s_2) can be defined as a CDL description that contains exactly one input constraint, namely s_1, i.e. $C_I^{\mathcal{T}} = \{s_1\}$, and exactly one output constraint, namely s_2, i.e. $C_O^{\mathcal{T}} = \{s_2\}$. Thus, $\mathcal{T}$ describes the task of transforming the state described by s_1 into a state in which s_2 is true.

Finally, the match conditions outlined above are based on the relation $\models$ between states rather than the function Mod_S that we have defined above. However, we have already pointed out in section 5.1.2.1 that the relation $\models$ can be

defined as the subset relation between models, i.e. $e_1 \models e_2$ for state descriptions $e_1, e_2 \in S$ if and only if $Mod_S(e_1) \subseteq Mod_S(e_2)$. Thus, the definition for capability subsumption can be used to define the action definition function Rel_A as outlined above.

This concludes our definition of expressiveness and the application of this framework to CDL. A comparison of the expressiveness of CDL with that of other action representations will follow in section 9.2.1.

Chapter 8

Flexibility of CDL

At this point we have described our capability description language CDL *which can be used to represent the content of messages required for capability brokering. We have also shown how* CDL *can be used for capability retrieval. Our aim is to show that* CDL *has two desirable properties: it is expressive and flexible. The next step towards this goal will be to define what we mean by flexibility in action representations. The contribution of this chapter will be a discussion of how flexibility has been achieved through decoupled action representations. In particular, we will be highlighting the issues involved in implementing this approach in* CDL. *A comparison of the flexibility of* CDL *with that of other representations will follow in section 9.2.2.*

8.1 Why Flexible Action Representations?

In this section we will argue why we need a flexible action representation for capability descriptions. This argument will be based on a scenario presented earlier in this thesis (in section 3.3).

As for expressiveness, we have simply *assumed* up to this point that flexibility is a useful property of our capability description language. However, while expressiveness can be extremely costly, flexibility has less of an impact on the

potential decline in performance (cf. section 6.3.2). General performance issues have already been discussed in section 6.3. In this section we will discuss why flexibility is a desirable property.

8.1.1 The Flexibility Scenario Revisited

One of the scenarios introduced in section 3.3 addresses the question of why we need more flexibility; the *flexibility scenario* (example 3.3). The very idea behind this scenario was to illustrate the need for a flexible capability description language. We will now look at this scenario again to re-evaluate this need for flexibility.

8.1.1.1 Restating the Scenario

In the flexibility scenario, our focus was on three problem-solving agents: two hospitals and an ambulance service. The *first hospital* does not have an ambulance in this scenario and thus, can only treat patients that are at the hospital. This hospital advertises the following capability description to the broker (cf. section 4.5.2):

```
(capability
  :state-language lits
  :input ((InjuredPerson ?person))
  :input-constraints (
    (elt ?person Person)
    (Is ?person Injured)
    (Has Location ?person Hospital1))
  :output-constraints (
    (not(Is ?person Injured))))
```

Note that the *state language* plugged into CDL is given as `lits`, which only allows literals as its expressions. Since lists of constraints are interpreted as conjunctions in CDL, this language effectively corresponds to the STRIPS representation. Describing the capability of the **h1**-agent in this rather simple representation does not present a problem in the scenario.

The *second hospital* has an ambulance, but does not want to spare it for too long. Driving the ambulance to Calypso or Delta is considered too far and thus, this hospital effectively only treats patients who are in Abyss, Barnacle, or Exodus. The capability description it advertises to the broker is represented as (cf. section 4.5.2):

```
(capability
  :state-language fopl
  :input ((InjuredPerson ?person))
  :input-constraints (
    (elt ?person Person)
    (Is ?person Injured)
    (or (Has Location ?person Abyss)
      (Has Location ?person Barnacle)
      (Has Location ?person Exodus)))
  :output-constraints (
    (not(Is ?person Injured))))
```

The fact that patients can only be treated if they are at certain locations on Pacifica is expressed as a disjunction. Thus, the *state language* required here needs to be more powerful than `lits`. The only state language we have implemented that can express such constraints is `fopl` and this is the state language used in this capability description.

Finally, the *ambulance service* cannot treat patients at all. It only transports them from any place on Pacifica to a hospital, i.e. no restrictions are imposed on the location of the injured person to be transported. The ambulance service advertises the following capability description to the broker (cf. section 4.5.2):

```
(capability
  :state-language lits
  :input ((InjuredPerson ?person)(From ?p1)(To ?p2))
  :input-constraints (
    (elt ?person Person)
    (Is ?person Injured)
    (Has Location ?person ?p1))
  :output-constraints (
    (not(Has Location ?person ?p1))
    (Has Location ?person ?p2)))
```

As for the first hospital, the capability description is sufficiently simple to only require conjunctions of literals in its *state language*.

The *problem* for this scenario was to heal an injured person at the power plant which is located in Delta. The task description in CDL is represented by (cf. section 4.5.2):

```
(broker-one
  :sender pp
  :content
    (task
      :state-language lits
      :input-constraints (
        (elt JohnSmith Person)
        (Is JohnSmith Injured)
        (Has Location JohnSmith Delta))
      :output-constraints (
        (not(Is JohnSmith Injured))))
  :ontology capabilities
  :receiver ANS
  :language CDL)
```

The capability description of the second hospital will not match this problem as the patient needs to be at Abyss, Barnacle, or Exodus for this capability to be applicable. Similarly, the capability of the first hospital is not applicable because the injured person is not at this hospital. Finally, the ambulance service is not capable of achieving the right kind of objective at all. However, a combination of the ambulance service and the first hospital will solve the given problem and this is the *planned solution* the broker finds.

8.1.1.2 Analysis of the Required Flexibility

What is interesting in this scenario is that different agents use different state representation languages in their capability descriptions that allow for different types of reasoning, i.e. they exploit the flexibility offered by CDL. Our aim now is to *show that this flexibility in the capability description language was necessary to adequately deal with this scenario*. To this end we will argue that the alternatives are not suitable, leaving only the flexible representation as adequate.

The first alternative would be to simply *choose the most expressive state representation* language used and make all agents describe their capabilities using this state language. In the above example this would be first-order logic (`fopl`) and the re-expression of the capabilities of the first hospital and the ambulance service using `fopl` would not pose a problem. Note that in general there might not exist one language in a given scenario that is more expressive than all languages used in this scenario.

In the flexibility scenario, reasoning over a language as powerful as first-order logic for all agents causes a problem. Since none of the single agents can solve the problem at hand the broker will attempt to create a plan involving the capabilities of several agents. However, our planner cannot deal with capabilities described using first-order logic. This limitation is not artificial but fairly common amongst current AI planners. Thus, if we allowed only `fopl` as a possible state description language here, the broker would not be able to come up with a plan involving the ambulance service and the first hospital, i.e. the broker would not be able to find a solution to the problem described.

The second alternative would be to *only allow conjunctions of literals as state descriptions* within CDL. The problem then is how to describe the capability of the second hospital, specifically, how to represent its disjunctive input constraint. The approach of splitting the single capability description into several descriptions has already been discussed in section 7.1.1. The problem with this approach is that it may lead to a large number of separate capabilities.

Another option that requires only conjunctions of literals in the capability description of the second hospital would be to sacrifice correctness and to drop the disjunctive constraint altogether. However, assuming there are equally many injuries in all five cities on Pacifica, dropping the disjunctive input constraint would lead to 40% false matches during brokering. This may not be acceptable.

To summarise, maximising the expressiveness of the state language to avoid the need for flexibility also means minimising the potential inferences that can

be drawn. Minimising the expressiveness on the other hand leads to problems in re-expressing capabilities that previously took advantage of a more expressive representation. Any compromise between these extremes is bound to lead to some of these problems, too, and for the flexibility scenario which is based on only two content languages there is no such compromise. Thus, what is required to appropriately deal with the above scenario is the *flexibility* we have designed into CDL.

8.1.2 Further State Representations

The flexibility scenario illustrates the need for flexibility in a capability description language by requiring two different state description languages: first-order logic and conjunctions of literals. The need for only two different content languages in this scenario might be considered insufficient to allow us to conclude that, in general, different scenarios may require a whole spectrum of languages and thus, flexibility. To address this concern, we will now look at some *further possible circumstances* that may need to be represented in different scenarios and which require different state state languages.

- **Agent Knowledge**: Many actions not only change the physical world but also the knowledge states of agents. For example, to transport the patient to a hospital the ambulance service needs to know where the patient is. This is a knowledge precondition not represented in the current flexibility scenario. Similarly, there could be knowledge goals. Representations that can express such circumstances are presented e.g. in [Moore, 1985, Morgenstern, 1987, Lesp'erance, 1989]. However, reasoning about knowledge in general is still an open issue due the problem of logical omniscience (cf. [Fagin *et al.*, 1995, chapter 9]).

- **Existence**: Many actions create and/or destroy objects. CDL allows for the representation of newly created objects in its output parameters, but the

general problem is that current state description languages do not provide an adequate representation for existence. For a detailed discussion of the problems associated with the representation of and reasoning about existence see [Hirst, 1991].

- **Uncertainty**: Many actions are intrinsically uncertain. For example, performing an operation always contains the risk that the patient might die. This is an example of uncertainty in the relation between states, i.e. uncertainty in the outcome of an action. A second type of uncertainty is uncertainty in single states. For example, if the information we have is only of qualitative nature, like "the patient is heavy", we are dealing with uncertainty in states.

The state representations we have actually implemented to support our chosen scenarios cannot adequately handle any of these circumstances. Thus, further, more expressive state representations would be necessary in scenarios which required the representation of such circumstances. Therefore, the flexibility of CDL is required for a generic capability description language that does not preclude the representation of knowledge that may be important for certain capabilities.

8.2 Defining and Implementing Flexibility

In this section we will define what we mean by flexibility and discuss some of the problems involved in implementing flexible representations such as CDL *through decoupled languages.*

8.2.1 A Definition of Flexibility

We will now define what we mean by a flexible knowledge representation language. Unlike for expressiveness, *this will not lead to a formal framework* that could be used to develop a flexibility hierarchy of different representations. One reason for this difference is the fact that we are not aware of any previous work on formalising what we have called flexibility. Another reason is that the issues involved are not so much problems of formalisation, but problems with the implementation of flexible representations to which we will turn later in this section.

8.2.1.1 Flexibility and Trade-Offs

In the flexibility scenario the problem arises because the problem-solving agents require different state representation languages within CDL: first-order logic and conjunctions of literals. First-order logic is required because it provides the expressiveness needed by the second hospital to represent its disjunctive input constraint. Conjunctions of literals are required for the first hospital and the ambulance service because expressions of this type allow sufficiently efficient reasoning to facilitate planning. Thus, the underlying problem in the flexibility scenario is a *classic trade-off* that can be found in knowledge representation and reasoning: expressiveness versus efficiency.

Most conventional knowledge representation languages are designed as a *compromise* with respect to this trade-off, i.e. they offer some degree of expressiveness which allows for some degree of efficiency. For such languages, a specific compromise has been chosen when the language was designed. CDL is different in

that it does not prescribe a fixed compromise. This is exactly what we mean by flexibility in a knowledge representation language:

Definition 8.1 (Flexibility) *A knowledge representation language is* **flexible** *if it allows the knowledge engineer to choose a compromise regarding a certain trade-off at the time of knowledge representation rather than having to adopt a fixed compromise prescribed by and designed into the representation.*

Note that this definition is not specific to action representations but can be applied to knowledge representation formalisms in general. Note also that a compromise regarding a given trade-off has to be chosen in conventional as well as flexible knowledge representation languages. The *difference lies in when* this compromise has to be chosen. While this is at language design time in conventional knowledge representation languages, flexible knowledge representations allow one to make this choice later, i.e. at knowledge representation time, e.g. when a capability needs to be represented. Thus, it is possible to choose different compromises for every statement (e.g. a capability in CDL) in a flexible knowledge representation, rather than having one fixed compromise prescribed by the language. In this sense, flexible knowledge representations to some degree can be regarded as a least commitment approach to knowledge representation.

8.2.1.2 Language Properties

The trade-off mentioned in the definition of flexibility is a trade-off *between some properties* of a knowledge representation language, e.g. expressiveness and efficiency. During knowledge representation, a compromise between such properties has to be chosen. Up to now we have only looked at the compromise between two such properties: expressiveness and efficiency. The former has already been discussed in chapter 7.

Now we will briefly look at some *further properties* that might cause trade-offs during knowledge representation and thus might require flexibility. By effi-

ciency we essentially mean the potential to perform fast reasoning over a formalism; what has been called heuristic adequacy in [Wilkins, 1988, page 8] or [McCarthy and Hayes, 1969]. Another way to view efficiency is as usage of the resource time, which is to be minimised for time efficiency. Similarly, other resources, like memory, can be minimised to result in different kinds of efficiency in a language, e.g. memory efficiency. Another language property we have considered in the design of CDL is the formality of a language (cf. section 4.1.1). Yet other properties that might cause trade-offs include generality, i.e. the ability to support generic rather than task specific reasoning, richness [Polyak and Tate, 1998], explainability [Swartout, 1983], and declarativeness [Ginsberg, 1993, page 9].

To summarise, flexibility in a knowledge representation language, like CDL, allows one to choose a trade-off between several properties at the time of knowledge representation. In a dynamic world of agents this flexibility is required in a capability description language to allow each agent to choose an appropriate compromise in its capability description, as illustrated in the flexibility scenario.

8.2.2 Flexibility through Decoupling

Flexibility is achieved in CDL through its implementation as a *decoupled action representation*. Thus, we shall now briefly discuss decoupled action representations and how they provide flexibility.

8.2.2.1 Integral Action Representations

As we pointed out in section 4.2.3.1, many knowledge representation languages are *state representation languages* at heart, i.e. they assume the world to be in exactly one state. That is, unless otherwise stated, a set of sentences in such a language is assumed to refer to the same state. The most commonly used knowledge representation language that makes the above assumptions is first order logic. It is possible to represent and reason about actions in first order logic as demonstrated by the situation calculus (cf. section 2.2.1), but this leads to a number of problems;

most prominently the frame problem. Hence the development of specific action representation languages (cf. section 2.3.1).

In conventional action representation languages the state representation language is an integral part of the overall representation. We have called such languages *integral action representations* in section 4.2.3.1. For example, the first (integral) action representation language was the STRIPS representation [Nilsson, 1980, chapter 7]. In STRIPS-like languages an action is represented as a statement that contains several sub-expressions in what can be considered the state representation language. This state representation is an integral part of the action representation.

Thus, integral action representations are *not flexible* because the integrated state representation language prescribes a fixed compromise that has been chosen when the language was designed.

8.2.2.2 Decoupling Action Representations

To allow the arbitrary combination of action and state representation one needs to define the action representation language independently from the state representation language. We have called this a *decoupled action representation* in section 4.2.3.2. Thus, a full action representation consists of the combination of a decoupled action representation with a state representation language.

The most important difference between a decoupled action representation and its conventional, integral counterpart is that it allows one to plug different state representation languages into the same decoupled action representation language. This feature of the language results in the *flexibility* of the action representation as described in definition 8.1. This flexibility is what is needed to address the problem in the flexibility scenario: we can combine an appropriate decoupled action representation (CDL) with an appropriate state representation (e.g. `fopl` or `lits`) for each agent's individual representational needs.

Decoupled action representation languages are flexible because plugging in

different state languages changes, for example, the expressiveness of the overall language as well as the efficiency with which we can reason over this language. In this way, decoupled languages allow compromises between these properties to be chosen at knowledge representation time and thus, *they provide flexibility.*

8.2.3 Implementing Decoupled Languages

As opposed to expressiveness, the implementation of a flexible knowledge representation turns out to be quite challenging. We will now discuss some problems encountered during the implementation of CDL as a decoupled action representation. Some of these problems are specific to decoupled action representations, but most problems need to be addressed in any decoupled knowledge representation. We will return to these problems in section 9.2.2 where they will form the basis for our evaluation of the flexibility of CDL as compared to other decoupled languages.

8.2.3.1 Problems with Decoupling the Languages

The first group of problems we have encountered during the implementation of CDL as a decoupled action representation is related to the implementation of the internal representation of a statement in the language itself.

How to Allow for Arbitrary Content Languages Our implementation of CDL as a decoupled language follows the example of KQML (cf. section 2.1.2.3). KQML allows content expressions to be in some arbitrary content language by having a field that names this language and one that holds exactly one expression in this language as a sub-expression of the KQML message. CDL, too, *allows for arbitrary content languages* by having a field that names the content language to be used, namely the state-language field. Requiring the content language to be explicitly named permits the plugging in of arbitrary content languages.

There are some *minor differences* between KQML and CDL though. As opposed to KQML, there are several fields that contain expressions in the content language

in CDL: the input constraints, the output constraints, and the input-output constraints. Furthermore, each of these fields contains a list of expressions in the named language rather than just one expression as in KQML. Another difference between KQML and CDL lies in the meaning of the expressions. The outer part of a KQML message represents the speech act that is performed with this message and the inner part conveys the content of the message. An expression in CDL represents an action; either a capability that can be performed or a task that needs to be performed. Content expressions within a KQML message have to be interpreted with respect to the performative in which they are embedded whereas content expressions in CDL always represent constraints on states.

Where to Decouple the Languages One of the issues arising in the design of a decoupled representation is *where to decouple the language*, i.e. where to make the cut between inner and outer language. Looking at the syntax of an integral knowledge representation, one can, in principle, create a decoupled representation by replacing any non-terminal symbol with a named language. However, such arbitrary decoupling can hardly be expected to result in useful decoupled languages. As there currently exist only a handful of decoupled representations, it is difficult to generalise where such languages should be decoupled. In our limited experience the cut should be made such that the different languages represent fundamentally different entities. For example, in KQML the outer expression represents a speech act and the inner expression is a statement of some kind.

In CDL, the outer expression represents a binary relation between states and the inner expressions represent constraints on states, i.e. the *cut is between actions and states.* This cut is meaningful only for action representations though. As suggested in section 4.2.4, a second cut that could conceivably be made in CDL would be to allow a separate language for terms. Again, these represent a fundamentally different collection of entities, namely objects in the domain. Thus, such a cut could be useful but has not been made in CDL in order to simplify this implementation.

How to Parse Decoupled Languages Another problem with the implementation of decoupled languages is the *parsing problem*. There are basically two approaches to parsing a sentence in a decoupled language. Firstly, the parser can read and parse the sentence according to the syntax of the outer language up to the point where it expects a sub-expression in the content language. Then it can extract this expression as a separate string and continue parsing after the end of this sub-expression, according to the syntax of the outer language. After the complete outer part of the sentence has been parsed the inner expression can be dealt with. The problem with this approach is that the parser for the outer language needs to be able to decide where the expression in the inner language ends. CDL as well as most implementations of KQML assume that this is possible, usually by requiring the inner language to be enclosed in parentheses and to only contain balanced pairs of parentheses.

A second approach to parsing a decoupled language is to start parsing the expression according to the syntax of the outer language up to the point where it expects a sub-expression in the content language. At this point the parser switches to the syntax of the inner language, parses the content, and returns to the outer language afterwards. However, in general the parser will need to read at least one more token at the end of the inner language to decide whether this expression is actually complete. If the inner expression was complete, this token will not be defined in the syntax of the inner language and the behaviour of the parser is undefined at this point. Even worse, the token following the inner expression might have meaning in the syntax of the inner and outer language. Again, this can considerably complicate parsing.

8.2.3.2 Problems with Reasoning over Decoupled Languages

Defining the internal representation of a decoupled action representation language is not the only group of implementation problems though. The second group of problems we have encountered during the implementation of CDL as a

decoupled action representation is related to the implementation of the reasoning mechanisms for decoupled languages.

How to Determine What Inferences can be Drawn To reason over a decoupled language, virtually any reasoner will need to make explicit inferences over the inner language. In CDL, for example, we have evaluated whether a capability subsumes a task by performing certain inferences over expressions in the state description language (cf. chapter 5). That is, we have reduced inferences over the outer language to inferences over the inner language. But we have not only drawn inferences within the inner language, we have also drawn inferences about the inner language. Since the inner language could be an arbitrary knowledge representation, we have to work out which inferences are supported by this language. The inferences supported by the inner language then determine which inferences can be performed over the outer language. Reasoning about the inner language to determine which inferences are supported is a kind of *reflective reasoning* (cf. section 2.2.3.1).

We shall illustrate this using an imaginary decoupled version of STRIPS. In the STRIPS representation state expressions are used to describe preconditions, an add-list, and a delete-list. Initial and goal states are also represented as expressions in the state language. The STRIPS planner basically works by decomposing goals, testing whether an expression is true, and by regressing goals through operators. The latter is mainly a combination of retraction and assertion of state expressions to generate new states. Thus, the STRIPS planner could, in principle, work with any state representation language that defines these operations (decomposition, expression evaluation, assertion, retraction) in its interface. However, a decoupled STRIPS planner would not only need to perform these inferences, it would also need to reflect on whether they are defined in the actual state language used. Similarly, a decoupled plan-space planner requires a state representation language that supports decomposition, test for entailment (can the action bring about some goal state), and test for inconsistency (test for clobbering).

How to Define available Inference Mechanisms The problem with reflective reasoning as required for the implementation of decoupled languages is that it is hardly supported in Java and other programming languages. Reflection in Java primarily allows one to find a function that has a given name and takes certain parameters. If there is a function that performs the right kind of inference over the state language, but this function has a different name from the one the reflective algorithm is looking for, it will not be found. We have addressed this problem in CDL by introducing what we call *optional functions*. These functions are defined in the API of the class `Language` from which every state language must inherit. However, rather than enforcing the implementation of these functions through the normal inheritance mechanism, these functions are optional, i.e. they may or may not be implemented in a class inheriting from the class `Language`. The idea behind optional functions is that they constitute the definition of an interface for certain functionality in case this functionality is provided. With such an interface the interpreter for the outer language can easily reflect on whether some functionality of the inner language is available.

When a reasoner like our broker attempts to perform certain inferences over expressions in the outer language it has two options for *testing whether the functionality required in the inner language is available*. Firstly, the reasoner could reflect on whether all the required functionality in the inner language exists before attempting to reason over the outer language. If this is not the case, the reasoning attempt over the outer language is immediately abandoned. Secondly, the reasoner could only test for functionality when it is required. This is how we have implemented brokering for CDL. The advantage of this approach is that functionality that may be used in an algorithm but is not necessarily used will only be tested if it is actually used. The disadvantage is that the test will be performed every time the functionality is used, leading to a slight inefficiency.

How to Control the Reasoning Process Another, potentially more severe problem is the fact that the reasoner over the outer language has to pass *control* to

a reasoner over the inner language when it uses the functionality offered there. For example, to test for capability subsumption our broker needs to evaluate whether a set of sentences in the state language entails another. We have used first-order logic as the state language in most of our examples and entailment is tested via resolution theorem proving. This process is not guaranteed to terminate. Thus, when the broker passes control to the theorem prover, it might never re-gain control. This behaviour is highly undesirable. In our implementation we have addressed this problem by limiting the number of clauses that will be generated, but in general the problem remains.

How to React to an Unknown Language Finally, a problem arises when the reasoner over the outer language attempts to perform an inference and discovers that it does not know the named inner language. In this case Java and JAT provide the support needed to address the problem (cf. section 5.3.3). Effectively what happens in this case is that the reasoner automatically contacts the sender of the message containing the unknown inner language. It is reasonable to assume that the sender knows the language which it is using to communicate. Thus, this agent is asked where the Java class corresponding to this language can be found. When this information is made available to the reasoner, it will attempt to load this class from the specified location and then perform the reflective reasoning over this language as outlined above. With this mechanism, the outer language can be completely decoupled from the inner language.

Summary

In this section we have defined what we mean by a flexible knowledge representation language and how this flexibility can be achieved in an action representation through its implementation as a decoupled language. We have also discussed a number of problems that arise in the implementation of decoupled languages. We shall return to these problems when we evaluate the flexibility of CDL by

comparing the solutions to these problems with those adopted in other flexible representations (cf. section 9.2.2).

Chapter 9

Related Work and Evaluation

At this point we have defined CDL*, an expressive and flexible action representation that can be used to represent and reason about capabilities of intelligent agents. Our aim in defining this formalism was to address the problem of capability brokering. The next step will be to compare* CDL *to the more closely related work described in chapter 2. The contribution of this chapter is an evaluation of* CDL*, specifically its expressiveness and flexibility, through a comparison with related work. It will also use a range of examples from other domains to demonstrate the generality of our approach.*

9.1 Comparison with other Brokers

In this section we will present a comparison of our CDL *broker with several other brokers described in section 2.1. Our focus will be on the capability description languages and matching algorithms used by the different systems.*

9.1.1 Capability Description Languages

The first aspect of the different systems we shall be looking at is the *capability description language used*. The languages supported by the different systems limit the capabilities that can be represented. Furthermore, the two properties

```
(<brokering-performative>
  ...
  :language KQML
  :content
    (<performative>
      ...
      :language <language>
      :content (<capability-description>)))
```

Figure 9.1: General format of brokering messages in KQML

we are most concerned with, expressiveness and flexibility, are properties of the capability description language. Thus, these languages are our initial focus. Later (section 9.1.2) we shall be looking at the inference mechanisms utilised by the different brokers to put the languages in perspective.

9.1.1.1 Languages Used

According to the KQML specification [Labrou and Finin, 1997], the content of most messages related to brokering should be another KQML message, i.e. the *general format* of these messages is as given in figure 9.1.

The brokering performative in the *outer part* of this message describes the brokering action to be performed with this message, e.g. `advertise` or `recommend-one`. The most important KQML performatives related to brokering were summarised in table 2.1 in section 2.1.2.

The content of the outer message is again a KQML message, the *inner* KQML *message*, which contains a performative and some content. The meaning of the inner message depends on the brokering performative in the outer message. In our scenarios all capabilities are physical actions on the environment of the agents. KQML provides only one performative that could be used as the performative of the inner message to represent such capabilities: `achieve`. All other performatives only deal with reasoning actions. Note that all brokers using KQML for

inter-agent communication should adhere to the message format described this far, i.e. they should be indistinguishable at this level.

The *content of the inner message* represents the capability in the message. KQML does not specify or suggest a language to be used for the content of the inner message, i.e. the language for describing capabilities is undefined. This is where the capability descriptions supported by the various brokers differ. Thus, our comparison of capability description languages offered by the different brokers concentrates on the content languages of the inner message. One possible content language is, of course, CDL and this is exactly the way we have used KQML in this thesis. In the remainder of this section we shall briefly review the different content languages used for capability descriptions before turning to their evaluation.

The ABSI facilitator (cf. section 2.1.3.1 or [Singh, 1993a, Singh, 1993b]) is based on an early KQML specification [Finin *et al.*, 1993] and supports only KIF [Genesereth *et al.*, 1992] as the content language to describe capabilities.

The SHADE and COINS matchmakers previously described in section 2.1.3.2 (or cf. [Kuokka and Harada, 1995a, Kuokka and Harada, 1995b]) support free text descriptions in the case of COINS, and KIF and MAX in the case of SHADE. MAX (Meta-reasoning Architecture for "X") [Kuokka, 1990] is a structured logic representation. In MAX all knowledge is declaratively stored in logic frames (or lframes). Each lframe denotes a possible "state" by representing the conjunction of a set of predicate logic literals. Lframes may be composed of other lframes, and may have local variables.

Like the SHADE matchmaker, the InfoSleuth broker (cf. section 2.1.3.3 or [Bayardo *et al.*, 1997, Nodine and Unruh, 1997, Nodine *et al.*, 1998]) supports two content languages for capability descriptions and again, the first supported language is KIF. The second supported language is the deductive database language LDL++ [Zaniolo, 1991] which has a semantics similar to Prolog, but which supports transparent access to external databases as well as its own fact base.

Brokers for problem-solving methods (PSMs), e.g. the Intelligent Broker (IB)

[Fensel, 1997, Decker *et al.*, 1998] and the IBROW^3 broker [Benjamins *et al.*, 1998, Armengol *et al.*, 1998] can be described as being in their early stages which means that they are not yet implemented and important design decisions remain to be taken. For example, the language in which PSMs are to be described is not yet defined. There is, however, a draft proposal for a PSM description language that is mostly based on KADS models of expertise, the conceptual modelling language CML [Wielinga (ed) *et al.*, 1994, chapter 3], and ML^2 (cf. section 2.4.1.2). This new language will be called the Unified Problem-solving Method description Language (UPML) [Fensel *et al.*, 1998a, Fensel *et al.*, 1998b].

The final broker we will have a look at here is the Object Request Broker of the Common Object Request Broker Architecture (CORBA) [Orfali *et al.*, 1997, Baker *et al.*, 1997, CORBA V2.2, 1998]. This broker is not based on KQML and was intended for the brokering of objects rather than agent capabilities. The language in which objects and their interfaces have to be described to the broker is called the *Interface Definition Language* (IDL) [CORBA V2.2, 1998, chapter 3]. IDL allows the specification of classes of objects in terms of their ingredients and interface, i.e. the functionality an instance of this class will offer to other objects.

To summarise, we can distinguish *three types of languages* for capability descriptions supported by the different brokers:

- **Free text** is supported by the COINS matchmaker and is used in most PSM description languages, e.g. UPML.

- KIF, a **logical language** based on first-order predicate logic, is supported by the ABSI facilitator, the SHADE matchmaker, and the InfoSleuth broker.

- **Object description languages** (MAX, LDL++, and IDL) are supported by the SHADE matchmaker, the InfoSleuth broker, and the Object Request Broker.

9.1.1.2 Evaluation

The most important question for this evaluation now is *how these languages compare to* CDL. The first group of languages mentioned above, languages based on free text, are, of course, very powerful languages. Every capability described in CDL can also be described in natural language, but presumably not vice versa. Similarly, since KIF is based on first-order predicate logic, it provides a highly expressive language. In fact, as we will argue in section 9.2.1.2, FOPL is a more expressive action representation than CDL. Finally, object description languages, or frame languages as they are sometimes called, also provide a quite powerful formalism. Thus, it appears that CDL, the language supported by our broker, compares rather unfavourably to the capability description languages supported by other brokers.

While expressiveness is an important issue, it is not our only concern. Another important issue is the support offered by a framework for *knowledge engineering*, e.g. the task of describing capabilities in a given formalism. Free text or KIF do not provide any support for this task and the knowledge engineer has to make a large number of choices without any guidance. CDL presents a significant advance in this respect: capabilities have to be represented as a collection of different objects manipulated by the capability and different types of constraints. Furthermore, our language facilitates the implementation of an ontology of performable actions from which capability descriptions can be derived. We believe that such an ontology is a very effective means for the facilitation of the knowledge engineering task, and although CDL does not include an ontology, it does provide the representational basis for it. Thus, CDL provides substantially more support for the knowledge engineering task than any of the languages used by other brokers.

9.1.2 Reasoning Facilities to Support Brokering

Comparing just the languages results in an incorrect picture. It is equally important to *compare the inference mechanisms* employed by the different brokers

to reason over the languages they support. If a language has certain features that are not supported by the broker's reasoner then these features should not be considered part of the representation. We will now review the reasoning mechanisms implemented by the different brokers. In the next section we will re-evaluate the different capability description languages, showing that they are not nearly as expressive as they initially appeared.

Brokers reason about capabilities on *two levels*. Firstly, they need to test whether a given capability can be used to solve a given problem. This inference can be seen as the essence of brokering. Secondly, brokers maintain a database of capability descriptions on which they can perform certain operations, e.g. retrieving a capability for a given problem. Since the interface to a broker is defined in KQML, all brokers adhering to the KQML specification should support the same inferences at this level. Note that this interface corresponds to the outer part of the general format of brokering messages described in figure 9.1. We shall look at matching of capabilities and problems in section 9.1.2.2. But first we review the interface provided by the different brokers for maintaining their database of capability descriptions.

9.1.2.1 Supported Performatives

KQML only defines the behaviour a broker should exhibit on receipt of the various brokering performatives, i.e. it defines what the result of the reasoning that has to take place in the broker should look like (cf. table 2.1 in section 2.1.2). KQML does not specify how this is to be implemented though.

The only brokering performative explicitly provided by the ABSI facilitator is `handles`. Note that no such performative is defined in the KQML specification. The `handles` performative implements exactly the behaviour specified for the `advertise` performative in KQML, i.e. the ABSI facilitator effectively provides this performative. The functionality of another KQML brokering performative, namely `broker-one`, is only supported implicitly by the ABSI facilitator. The

basic mechanism is that problems are sent to the facilitator as if this agent was the one to solve the problem. The facilitator then manages the solution of the problem. The resulting behaviour corresponds to the `broker-one` performative in KQML. Thus, `advertise` and `broker-one` are the only brokering performatives which are (implicitly) supported by the ABSI facilitator.

The SHADE and COINS matchmakers as well as the InfoSleuth broker support all the KQML brokering performatives described in the specification and thus, they completely adhere to the standard.

The PSM brokers and CORBA on the other hand cannot be compared to the other brokers in this respect as they do not provide a KQML interface. The PSM brokers are still in an early phase of their development and such an interface might well follow. For CORBA there are no plans to provide a KQML interface, although such an extension has been attempted [Benech and Desprats, 1997].

Finally, of the brokering performatives defined in KQML, the only one our broker does not support is `subscribe`. All other brokering performatives are supported by our broker and conform to the KQML specification.

To summarise, amongst the KQML-based brokers there are *several that support all the brokering performatives* defined in the KQML specification. The only performative not provided by our broker is `subscribe`. The broker supporting the smallest number of performatives is the ABSI facilitator which only supports `advertise` and `broker-one` implicitly.

9.1.2.2 Matching of Capabilities and Problems

As mentioned above, brokers not only maintain a database of capability descriptions, they also have to test whether a capability can be used to address a given problem. How this inference is performed in the different brokers is the focus of this section.

KQML does not specify a content language for capability descriptions. It does, however, specify that matching between capabilities and tasks is to be performed

by comparing the respective performatives and contents of the two inner messages (cf. figure 9.1), and these match if they are equal [Labrou and Finin, 1997, page 19]. Note that this form of matching is rather trivial.

The matching between capabilities and problems which is performed by the ABSI facilitator is based on the matching algorithm between KIF expressions. For this matching, the KIF expression representing the generalised message content and the KIF expression representing the actual message content are treated as Prolog terms, and matching is performed like a unification with the Prolog equality predicate. If this unification succeeds, the additional constraints will be evaluated using the variable bindings obtained in the unification. If all the constraints can be satisfied, the capability subsumes the problem.

The matching that is performed by the COINS matchmaker is based on a concept vector extracted from text employing an inverse document frequency scheme, a technique often used in search engines [Witten *et al.*, 1994, Howe and Dreilinger, 1997]. The matching performed by the SHADE matchmaker is similar to the matching of the ABSI facilitator and based on the matching of KIF expressions. The MAX representation is based on frames and slots and provides little more than the KIF matcher by providing a Prolog-like unifier. Furthermore, advertisement and request must match solely based on their content and no additional predicates are allowed as for the ABSI facilitator. Limited inference for future versions is envisaged though.

Although the first language supported by the InfoSleuth broker is KIF, the standard matching method for KIF used by the other brokers is not used here. Capability descriptions using KIF are translated into LDL++ and the matcher operates on this language only. Effectively, the advertisement of a capability results in an assertion to a database. This is quite similar to the way the ABSI facilitator treated capability advertisements. Requests seeking capabilities are then treated as normal database queries.

The brokers for PSMs are, as mentioned before, still in an early phase of their

development. Not even the language they will use to describe PSMs is finalised yet, and no description of the matching algorithm they will use is available.

Finally, the matching algorithm used in CORBA is fairly straightforward considering that this is an object broker. It will be roughly the same as the one found in any compiler for unifying parameter specifications.

To summarise, most of the brokers employ a KIF-based matching algorithm that is very *similar to a simple unification algorithm* as described e.g. in [Robinson, 1965]. Specifically, matching between KIF expressions which is used by several brokers is defined in such a way. The only significant extension is provided by the ABSI facilitator, where in addition to this unification-like matching there may also be a number of Prolog predicates that must be evaluated. Note, however, that these predicates cannot be user defined. Other forms of matching are the equality test specified in KQML and the keyword-based algorithm for COINS.

9.1.2.3 Evaluation

Matching capabilities and problems is the essence of the reasoning the broker has to perform. All the brokers we have looked at including our own provides at least one brokering performative that is based on this matching. Thus, *we do not see a significant difference in the brokering performatives supported by the different brokers.* The reason for the omission of `subscribe` in our broker is that the type of scenario we were aiming to address is based on isolated problems which a problem-holding agent (PHA) experiences and seeks help with, i.e. capabilities are to be evaluated at run-time (cf. section 1.2.2). These problems largely tend to be non-recurrent. The `subscribe` performative on the other hand is intended for recurrent problems and the posting of persistent requests. Thus, the implementation of the `subscribe` performative did not appear necessary for our broker and we do not consider its omission a deficiency of our system.

While there is no significant difference in supported brokering performatives, *the matching algorithm implemented in* CDL *and based on the notion of capability*

subsumption (cf. definition 5.4) is significantly more powerful than the matching provided by other brokers. It is fairly easy to see that the notion of capability subsumption is more powerful than the equality test specified in KQML; if capability and problem description are equal then the capability also subsumes the problem. Thus, our matching algorithm (cf. figure 5.9) includes, and in fact, surpasses the KQML specification.

The same is true for all matching algorithms based on unification; CDL's subsumption test can be used to emulate this behaviour. One simple way to achieve this is to define a result variable as an output parameter to a CDL capability and have just one output constraint stating that the result variable must be equal to the expression describing the capability. The same procedure can be applied to the problem description. In this case our capability subsumption test would attempt to unify the two expressions to test the output match condition, and the result of this test would determine whether the capability matches the problem, since there are no input constraints. Thus, the result of our capability subsumption test depends solely on the result of the unification, i.e. it emulates the other brokers' matching behaviour.

A slight extension of this procedure can be used to emulate the matching performed by the ABSI facilitator, which allows for additional constraints on the variables bound during the unification. These constraints must be specified in terms of predicates defined in the language, e.g. Prolog. Assuming our broker also knows this language and the predicates defined in it, these constraints can simply be specified as input constraints of the CDL capability and they will be evaluated as part of the input match condition. Thus, our capability subsumption test can also emulate the matching behaviour of the ABSI facilitator. An example, of this procedure for a capability used by the ABSI facilitator is given in section 9.3.1.

This shows that the subsumption test defined in CDL can indeed be used to emulate the matching performed by the other brokers. However, the rather simple, unification-based matching used by these brokers cannot emulate our

subsumption test. The reason for this is that KIF-like unification is essentially a syntactic matching whereas the subsumption test for CDL is based on the content language's semantics. For example, a KIF sentence consisting of the conjunction of two propositions only matches (unifies with) an identical sentence, but both conjuncts as well as the sentence in which the two conjuncts have swapped positions logically follow from the original sentence. The subsumption test used in CDL for matching easily copes with this example, but matching based on unification must fail for it. Thus, CDL's subsumption test is indeed more powerful than the matching used by other brokers.

A final concern worth mentioning here is that of *efficiency*. It is true that the subsumption algorithm used in CDL will usually be less efficient than a unification-based algorithm. This is only to be expected though as the problem it solves is far more complex than that solved by unification. A detailed complexity analysis of the capability subsumption algorithm can be found in section 6.3.1. However, as pointed out in section 6.3.2, the complexity of the capability subsumption algorithm does not depend on the parameter we would expect to scale up, namely the number of capabilities known to the broker. Thus, the complexity of this test does not present a problem for our broker.

9.1.3 Evaluating Expressiveness and Flexibility

In this section we will evaluate CDL by comparing the *usable* expressiveness and flexibility of the various capability description languages supported by the different brokers to that of CDL. We will show that CDL's usable expressiveness and flexibility are higher than those offered by other languages and that these features are the right ones to consider in the context of brokering.

9.1.3.1 Usable Expressiveness

Our *aim* now is to show that, while the expressiveness of the capability description languages offered by the other brokers appears high (cf. section 9.1.1.2), their

usable expressiveness is in fact comparatively low.

Characterising Usable Expressiveness By the *usable expressiveness* of a capability description language we mean the expressiveness of the language that can be utilised in a capability description and which can be adequately handled by the broker based on this language. In other words, the *usable expressiveness* of a language is the expressiveness of the subset of the language that can be adequately handled by the broker. Note that this characterisation of usable expressiveness of a language depends on the existence of a broker that handles capability descriptions in this language, i.e. usable expressiveness is only defined in the context of brokering.

The key requirement here is that the broker needs to be able to *adequately handle* capability descriptions. A broker handles capability descriptions by reasoning about them. What capability descriptions can be adequately handled by a broker depends on the reasoning mechanisms employed by the broker to draw inferences over the capabilities. Thus, the usable expressiveness of a language depends on the adequacy of the reasoning mechanisms employed by the respective broker.

Thus, our *aim* here is to show that the expressiveness which can be adequately reasoned about by other brokers is lower than the expressiveness that can be adequately reasoned about by our broker. We have reviewed the reasoning mechanisms implemented for the various brokers in section 9.1.2.

What remains to be specified at this point is when the reasoning mechanisms employed by a broker should be considered adequate. The essence of the reasoning performed by brokers is the matching between capabilities and problems, and this is also where the brokers we reviewed differ most from our CDL broker. We will say that a *broker's matching is adequate* if it minimises the number of false matches. By a false match we mean a situation in which the broker believes a capability description to subsume a problem, but in fact, the capability cannot be

used to address the problem. Furthermore, an adequate matcher will maximise the probability that it finds a capability to address a given problem if such a capability is available.

Comparing Usable Expressiveness In section 9.1.1.1 we showed that there are three types of languages for capability descriptions used by the brokers we reviewed. The first group is based on *free text*. The matching performed by the COINS matchmaker is based on a concept vector extracted from text employing an inverse document frequency scheme. Such keyword-based techniques can never be guaranteed not to result in false matches or to find a capability if there is one available. The reason for this potential inadequacy is that these techniques rely on words in the textual capability description which are taken out of context and are usually ambiguous. Thus, this form of matching will often be inadequate and the usable expressiveness of free text is in general undefined.

The other languages used by the different brokers for capability descriptions are KIF *and frame languages* for object descriptions. The limitation for the usable expressiveness here is again the matching performed by the different brokers. This matching is mostly based on unification. Only the ABSI facilitator has a slightly more powerful matcher that allows for additional constraints. As we have argued in section 9.1.2.3, the behaviour of such algorithms can be emulated by the subsumption algorithm implemented for CDL. Thus, the usable expressiveness of CDL is at least as high as that offered by the languages used by the other brokers.

A closer inspection of the KIF-like unification-based matching algorithms even reveals that CDL's usable expressiveness is higher. For example, unification-based matching cannot handle cases in which the capability description contains commutative operators or in which the capability cannot be instantiated to the problem description solely by the binding of variables. In such cases a unification-based matcher may fail to find the capability that can address the problem. Thus, the matching is inadequate in these cases and the usable expressiveness of the

language does not include them. CDL, however, can handle these cases, and thus, its *usable expressiveness is higher* than that of the languages used by the other brokers.

Expressiveness or Usable Expressiveness? Evaluating CDL by comparing its usable expressiveness to that offered by languages supported by other brokers allows for a more meaningful comparison than comparing the expressiveness of the various languages at face value as we did in section 9.1.1. In considering the usable expressiveness of a capability description language we disregard capability descriptions that can be written in a language, but which cannot be adequately reasoned about by the broker. Disregarding these capability descriptions does not change the desired or intended behaviour of the broker. But the desired and intended behaviour is exactly what we are interested in and thus, in the context of brokering comparing the usable expressiveness of capability descriptions is more meaningful.

9.1.3.2 Usable Flexibility

The second property we are most concerned with in this evaluation of CDL and our broker is *flexibility*. Again, we should look at the flexibility offered by the different systems in the context of the reasoning mechanisms supporting this flexibility, i.e. their usable flexibility. This turns out to be unnecessary though.

Flexibility as we have defined it (cf. definition 8.1) allows the knowledge engineer to *choose* a compromise regarding a certain trade-off at the time of knowledge representation rather than having to adopt a fixed compromise prescribed by and designed into the representation.

All the brokers we have reviewed provide either one fixed language for capability descriptions or at most two alternative languages. Providing just one language clearly provides no flexibility as there is no choice to be made. The brokers offering two different languages and thus some *limited degree of flexibility* are SHADE and the InfoSleuth broker.

	content language	reasoning performatives	reasoning matching	usable expressiveness	usable flexibility
KQML	undefined	all defined	equality	undefined	undefined
ABSI	KIF	`handles/` `broker-one`	unification/ constraints	low	none
SHADE/ COINS	KIF/MAX free text	all KQML performatives	unification keywords	low	low
InfoSleuth	KIF/ LDL++	all KQML performatives	unification	low	low
PSMs	UMPL	—	undefined	undefined	low
CORBA	IDL	—	unification	low	none
CDL	CDL	all except `subscribe`	subsumption test	high	high

Table 9.1: Comparison of different brokers

For our broker, capabilities have to be described in CDL. CDL is not just one language though. Through its plug-in mechanism for state languages it provides a framework for a whole set of languages, all based on the same top-level syntax. Each language in this set may provide a different compromise regarding a certain trade-off and by choosing the state language the knowledge engineer can choose the required compromise. The only caveat here is that we have only implemented two content languages that can be plugged into CDL, but our implementation of CDL as a decoupled language allows for arbitrary content languages.

Thus, CDL in principle offers a far *greater degree of flexibility* than the capability description languages offered by the other brokers. Furthermore, since this flexibility is fully supported by the reflective reasoning mechanisms, this high flexibility also means a *high usable flexibility* of CDL.

9.1.4 Summary

The result of the comparison between our CDL broker and other brokers is summarised in table 9.1.

In this section we have evaluated our broker and CDL by comparing it to other brokers reviewed earlier in this thesis. For this purpose, we have first compared

the capability description languages supported by the different brokers. At this point it appeared that the expressiveness offered by CDL was inferior to that offered by other languages. The only advantage of CDL revealed at this stage of the comparison was its better support for knowledge engineering.

In the next stage of this evaluation we reviewed the reasoning mechanisms employed by the different brokers. The essential difference between other brokers and our CDL broker found at this point was in the matching algorithm used to test whether a capability and problem description match: CDL's subsumption algorithm can be used to emulate the matching behaviour of the other brokers and, in fact, surpasses them.

The fact that other brokers only have limited matching algorithms also limits the usable expressiveness of the capability description languages they provide. Thus, CDL provides a higher usable expressiveness and we have shown that this is a more meaningful criterion than plain expressiveness in the context of brokering. Similarly, the fact that CDL is implemented as a decoupled language gives it a higher flexibility than offered by the other broker's languages. Thus, CDL and our broker can be used to represent and reason about all capabilities that can be adequately handled by other brokers and more.

A question that remains at this point is whether we have compared CDL with the right languages. CDL is an action representation and the languages used by the other brokers are far more general. Thus, we shall now continue this evaluation by comparing the expressiveness of CDL to that of other languages for the representation of similar entities: action representations. Note that we cannot compare the usable expressiveness here as there are no brokers for these languages. Similarly, we want to compare the flexibility of CDL to other languages that are meant to offer this property. This is what we will do in the following section.

9.2 CDL: Expressiveness and Flexibility

In this section we will look at the two important properties of CDL *again and show what has been achieved and where there are still open issues.*

9.2.1 Expressiveness of CDL

Expressiveness of action representation languages as we have defined it in definition 7.7 is a *relative measure*, i.e. we can only formally show that one language is more expressive than another, but we cannot show that a language is expressive in an absolute sense. However, using a less formal notion of expressiveness, one could say that a language is expressive if it is at least as expressive as most other languages that are intended for the representation of similar entities and relations. This is what we mean by our claim that CDL *is an expressive action representation.* Thus, we shall now compare CDL with other languages that are intended for the representation of similar entities and relations: action representations.

A formal comparison of CDL with other action representation languages would require these languages to be defined as AR1 languages, too, so that definition 7.7 of the expressiveness of AR1 languages could be applied. Needless to say, none of the action representation languages mentioned in section 2.3.1, which are the representations we intend to compare CDL with here, are defined as AR1 languages. This means that the comparisons that will follow may only be *informal.*

A final remark before we begin to compare the expressiveness of CDL with that of other action representations concerns the fact that AR1 languages are defined as decoupled action representations. Looking at the first part of definition 7.7 again, there are *two conditions* listed for a set of actions in one language to be expressed by a set of actions in another language. Firstly, for every state description in the first language there must be a corresponding state description in second language. Secondly, the set of states reachable through the two sets

of actions in their respective representations must also correspond to each other. Thus, one action representation can be more expressive than another because it can represent more complex states, i.e. the re-expression would fail on the first condition, or it can be more expressive than another because it can represent more complex relations between states in its decoupled actions, i.e. the re-expression would fail on the second condition. Note how these two reasons for differences in expressiveness reflect the distinction between states and actions in a decoupled action representation.

9.2.1.1 Comparing State Descriptions

In this section we will look at the *state description languages* used within the more interesting action representations described in section 2.3.1 and we will compare them with FOPL which we have implemented as one of the state languages in CDL. For this comparison we shall treat all representations as if they were defined as decoupled action representations and mostly ignore the part of the representation that expresses relations between states. This part of the representation will be reviewed in the following section. Here, we shall concentrate on the world states expressible in the different representations.

The first action representation to be reviewed here is the *situation calculus* which uses FOPL as the underlying representation for actions and states [McCarthy and Hayes, 1969, Shanahan, 1997]. Essentially, the predicate `Holds` is used to represent that a fact is true in a given situation. The fluent representing the fact is technically a function term but it describes factual knowledge in the form of a positive literal. It is fairly easy to see that `Holds` commutes with various connectives and thus allows the full expressiveness of FOPL for state descriptions.

The next group of action representations to be reviewed here are the classical non-hierarchical representations (cf. section 2.3.1.2). The original STRIPS planner [Fikes and Nilsson, 1971, Fikes *et al.*, 1972] was described as an extension of a

resolution theorem prover and a state was represented as a set of clauses, thus incorporating the power of FOPL for the state description language. However, the theorem prover was later dropped and only conjunctions of literals were allowed in the state representation, resulting in a far less expressive state representation, and a far more efficient planner [Nilsson, 1980, chapter 7]. Many planners are based on this restricted version of STRIPS.

The middle ground between the STRIPS representation with conjunctions of literals and the situation calculus was explored by [Pednault, 1989] and resulted in a new action representation called ADL. This language successfully combined the expressiveness of the situation calculus with the STRIPS assumption. Thus, states could be represented using FOPL. Interestingly, the language is described in a way that clearly distinguishes state and action representation. No attempt was made to define ADL with a state description other than FOPL though. The UCPOP planner was the first planner that was based on a restricted version of ADL [Penberthy and Weld, 1992, Barrett *et al.*, 1995]. However, once again states in this version were restricted to conjunctions of literals and limited universal quantification was only permitted over finite domains so that they could be replaced by a long conjunction. The latest version of UCPOP also allows disjunctions in places, thus allowing more expressiveness in states, but work on such extensions is still in progress.

The representations used in *contingency planners* like CNLP or Cassandra [Peot and Smith, 1992, Pryor and Collins, 1996] are essentially based on the STRIPS representation and world states are represented as conjunctions of literals only. However, looking at the action representation, alternative sets of effects can be specified for an action under different contingencies. This effectively represents a disjunction between two possible outcomes of an action and thus, provides more expressiveness than conjunctions of literals. On the other hand, it does not provide the full power FOPL by providing a limited form of disjunction in states.

Real world planners such as O-Plan [Currie and Tate, 1991, Tate *et al.*, 1994,

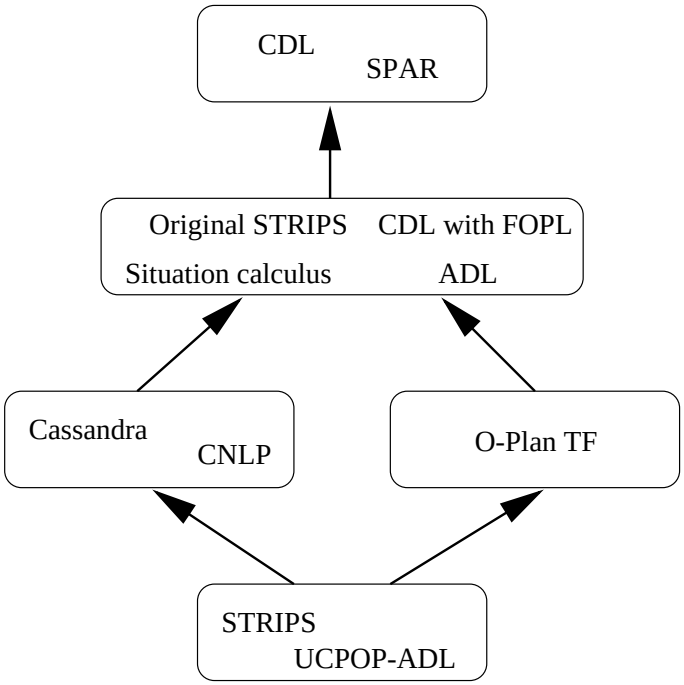

Figure 9.2: Expressiveness of state representations

Tate, 1995] must be efficient and thus essentially only allow conjunctions of literals in world states. However, these planners allow for a number of extensions such as reasoning about resources that, were they represented in FOPL, would require more than just conjunctions of literals to represent. Thus, the effective expressiveness of the state description language is higher than that of conjunctions of literals, but it does not provide the power of FOPL.

Finally, *shared action representations* such as SPAR [SPAR, 1997, Tate, 1998] are designed as decoupled action representations, although they are not defined as AR1 languages. Thus, states in SPAR could be represented in any language one chooses to plug in, including FOPL. In this sense SPAR can be considered to be the language most similar to CDL, allowing for the most expressive state description language.

The result of this comparison of the expressiveness of the various state representations used in action representations is summarised in figure 9.2. Action representations allowing the most expressive state description languages are at

the top of the figure.

9.2.1.2 Comparing Action Descriptions

In this section we will look at the *decoupled action description languages* used within the more interesting action representations described in section 2.3.1 and compare them with CDL. Note that the decoupled action representation just describes the relation between states, not how states are to be represented. As in the previous section, we shall treat all representations as if they were defined as decoupled action representations and we will mostly ignore the part of the representation that expresses states. This part of the representation was reviewed in the previous section.

As in the previous section, we shall begin our review with the *situation calculus* [McCarthy and Hayes, 1969, Shanahan, 1997]. However, a problem here is that the situation calculus does not provide a structure for the representation of an action. Any first-order sentence that mentions a specific action contributes to the definition of this action. Depending on the kinds of axioms one considers a situation calculus representation, actions can be highly expressive. [Shanahan, 1997, section 2.7] illustrates this point nicely with a formalisation of a toggle action. He uses the usual effect axioms to define the state after the action has been performed in a given state, but first-order logic also permits an axiom that expresses that the state remains unchanged after toggling a switch twice. Shanahan does not consider such axioms part of the situation calculus. Obviously the expressiveness of the decoupled action representation of the situation calculus very much depends on what exactly is permitted in the representation.

In the STRIPS action representation [Fikes and Nilsson, 1971, Nilsson, 1980] an action description consists mainly of three components: the preconditions, the add list, and the delete list. These define the relation between states as outlined in section 7.2.2. The difference between the early and later version of STRIPS lies only in the state representation.

In ADL [Pednault, 1989] actions are essentially defined as a number of situation calculus formulae. However, the syntax of ADL resembles more that of STRIPS and there is a precise definition of how to generate situation-calculus-like formulae from an action represented in ADL. This defines the semantics of ADL as an equivalence semantics based on the situation calculus, but it avoids the problem with the situation calculus mentioned above: that it is not clear which formulae should be considered situation calculus representations of an action and which should not. As for the expressiveness of the decoupled actions in ADL, the representation allows for preconditions, add list, and delete list, like the STRIPS representation. An essential extension is that one can also specify conditional add and delete lists in ADL. The same extension is implemented in the UCPOP planner [Penberthy and Weld, 1992, Barrett *et al.*, 1995]. The restrictions imposed by this planner are solely on the state language and thus, UCPOP's version of ADL is as expressive as ADL as far as decoupled actions are concerned. In CDL, conditional effects are represented as input-output constraints.

Contingency planners differ from conventional planners by allowing the specification of various sets of effects in an action to represent the various contingencies that may occur. However, as we have argued above, these contingencies can be seen as one disjunctive set of effects and thus, this feature does increase the expressiveness of the decoupled action representation. Thus, the decoupled action representation of a CNLP action [Peot and Smith, 1992] is not more expressive than STRIPS, allowing only preconditions, add list, and delete list. Cassandra [Pryor and Collins, 1996] on the other hand implements contingencies as conditional effects and, as a by-product, it can deal with ordinary conditional effects too.

Real world planners like O-Plan [Currie and Tate, 1991, Tate *et al.*, 1994, Tate, 1995] need to be efficient, as pointed out before, and thus, do not implement conditional effects but provide many other features important for planning. And finally, the SPAR action representation provides more like an ontological definition

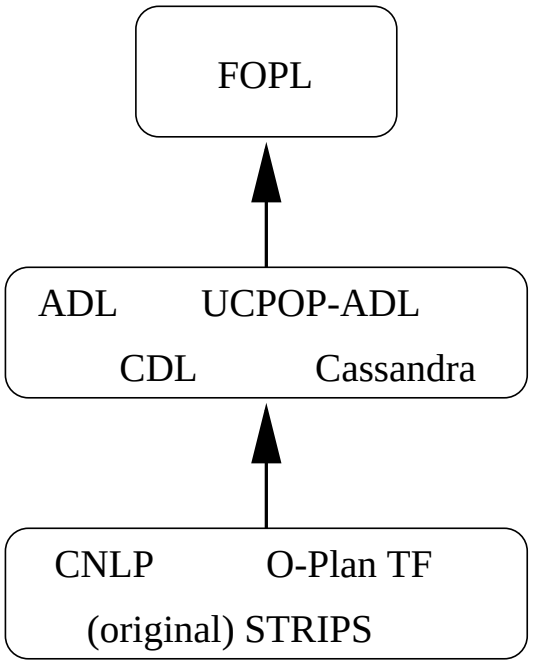

Figure 9.3: Expressiveness of action representations

of an action, rather than a formalism. While this leads to a very open language, it also means there is nothing that cannot be added to an action by defining a new class of actions that inherits from SPAR. Thus, it is not clear how one could compare the expressiveness of SPAR's decoupled action representation to that of CDL.

The results of this comparison are again summarised in figure 9.3. Only three levels of expressiveness for decoupled action representation are shown: basic STRIPS-like languages, languages supporting ADL-like conditional effects, and the unstructured representation of FOPL. CDL falls into the middle category.

9.2.2 Flexibility of CDL

Unlike expressiveness, flexibility has been defined in section 8.2.1 as a property that a knowledge representation either has or does not have. It may also be possible to distinguish different degrees of flexibility, but we have not done so. As for expressiveness, our evaluation of the flexibility of CDL will be done through a detailed comparison with other representations. However, the result will not

	UnLang	Ctl	OpFn	Refl	Pars	a/s	Content
KQML	ok	local	no	no	undef	no	arbitrary
Modal L.	fail	global	no	no	fixed	no	limited
Meta-K.	fail	local	no	yes	fixed	no	arbitrary
ADL	fail	global	no	no	fixed	yes	limited
O-Plan TF	defer	local	no	yes	undef	yes	arbitrary
SPAR	defer	local	no	yes	undef	yes	arbitrary
PDDL	fail	global	no	no	–	no	fixed
CML	ok	global	no	yes	–	no	fixed
CDL	load	local	yes	yes	( )	yes	arbitrary

Key: **UnLang**: behaviour on encountering undefined content language; **Ctl**: control over reasoning; **OpFn**: interface uses optional functions; **Refl**: interpreter uses reflective reasoning; **Pars**: approach to parsing problem; **a/s**: language distinguishes actions and states; **Content**: allowed content languages

Table 9.2: Comparison of flexibility

be a flexibility hierarchy, but a close inspection of the solutions offered by other languages to the implementation problems described in section 8.2.3.

The results of this comparison are summarised in table 9.2. In this table each row is labelled with a representation that can be considered flexible and each column is labelled with a problem that has to be addressed in a flexible language.[1] The first four columns are concerned with reasoning aspects.

The first column (UnLang) looks at the behaviour of an interpreter for the representation when it encounters an unknown language (cf. section 8.2.3.2). In many languages the parsing of an expression with an unknown content language would simply fail. This is because, although the definitions of these languages can be seen as decoupled, in practise they are integral knowledge representation languages. There are some notable exceptions though. KQML (cf. section 2.1.2.3), for example, is meant to perform no reasoning over the content of a message and thus, most KQML interpreters will cope with an unknown content language. However, a problem might well occur at a later stage in the reasoning process, as it would, for example, in the JAT implementation of KQML. A different approach

[1] By a language we mean a class in the object-oriented sense here, i.e. it includes the functionality provided for reasoning over this language.

is taken in O-Plan TF (cf. section 2.3.1.5) and SPAR (cf. section 2.3.1.6). Both languages allow for the specification of constraints in new languages. SPAR even provides a mechanism that allows one to specify the syntax of a new language. However, no example of this extension mechanism is actually implemented as far as we are aware today. The basic mechanism for reasoning over constraints in an unknown language in O-Plan is to postpone evaluation of these constraints until an appropriate constraint manager becomes available. CML (cf. section 2.4.1.2) is a special case as it allows free text as content which includes any language, i.e. there cannot be an unknown language.

Our language, CDL, distinguishes itself from all other languages here by facilitating the retrieval of an interpreter for an unknown content language which may be incorporated into a reasoning process (cf. section 5.3.1.2), resulting in the flexible behaviour of the broker.

The second column of table 9.2 (Ctl) analyses whether an interpreter for the representation passes control to an external interpreter for the content language (cf. section 8.2.3.2) which corresponds to local control; otherwise control is considered global. Passing control to an interpreter for the content language can be seen as a greater degree of decoupling in the language. For example, KQML does not specify how the reasoning over the content language is to be performed, but the implementation in JAT, for example, maintains a number of external interpreters that perform this reasoning, i.e. control is local to each interpreter. Similarly, most systems using explicit meta-level knowledge (cf. section 2.2.3), e.g. PRODIGY's search controller, have independent modules to reason over meta- and object-level knowledge. O-Plan also has a controller and constraint-associator to handle its meta-level reasoning. In CDL control is passed to an external interpreter when capabilities are evaluated (cf. section 5.1.2).

The third column of table 9.2 (OpFn) studies whether the procedural semantics of the language is defined in terms of optional functions, i.e. functionality that is defined for all languages but only provided for some (cf. section 8.2.3.2).

Although optional functions do not directly indicate flexibility in a language, we have found them to be a very useful implementational mechanism for a decoupled language. Of the languages listed in the table, CDL appears to be the only language that supports this mechanism. An example of such a function is *evaluate* in figure 5.2.

The fourth column of table 9.2 (Refl) examines whether an interpreter for the representation uses explicit reflective reasoning to determine how to reason over the content expressions (cf. section 8.2.3.2). Interpreters for representations using meta-level knowledge almost by definition fall into this category. O-Plan can also be described as employing reflective reasoning when it performs explicit reasoning over which knowledge source to fire next based on the open issues in a partial plan. CML is again a special case because of its free text base. If there is any reasoning to be performed it almost has to be reflective to work out which part of the free text can be used for reasoning. CDL uses reflective reasoning when it performs a subsumption test as described in section 5.1.2.2.

The last three columns of table 9.2 are concerned with solutions to implementation problems of the individual representations. The fifth column (Pars) looks at the approach to the parsing problem (cf. section 8.2.3.1) in the different representations. The only languages we are interested in here are, of course, those that do not have a fixed parser for the content language as this means the language is not really flexible. KQML is a decoupled language, but the approach to the parsing problem is undefined in the language. The most generic approach to the parsing problem is probably that in SPAR and the latest version of O-Plan TF, which allows the specification of a syntax as part of the language. How ambiguities are to be resolved is not clear though. PDDL (cf. section 2.3.1.6) allows for various extensions in the language, but these are fixed and thus the parser is fixed, too. CML is a special case again, because no attempt is made to parse the free text that is its content. CDL is somewhat similar to some KQML implementations in that it requires expressions in the content language to be in

brackets, and the content languages we have implemented adopt the Lisp syntax (cf. figure 4.4).

The sixth column of table 9.2 (a/s) takes a look at the cut that is made in the representation, specifically, whether it distinguishes between actions and states (cf. section 8.2.3.1). Not all of the representations in the table are action representations and thus, it is not surprising that the only representations other than CDL which decouple actions from their underlying state representation are ADL (cf. section 2.3.1.2), O-Plan TF, and SPAR. The decoupling of CDL to distinguish actions and states has been discussed in section 4.2.3.

Finally, the last column of table 9.2 (Content) examines whether the outer language allows the plugging in of arbitrary content languages (cf. section 8.2.3.1). This is obviously one of the most fundamental features of a flexible action representation. KQML, meta-level knowledge representations, O-Plan TF, and SPAR all allow for arbitrary content languages. Modal logics (cf. section 2.2.2) also allow some degree of freedom but the range of language they have been used with is rather limited. Similarly, the definition of ADL can be interpreted as the definition of a decoupled language, but the actual language was only specified for one content language. CDL allows one to plug in arbitrary content languages (cf. section 4.2.3.2).

Summary

A language can be considered expressive if it is at least as expressive as other representations for similar entities and relations. Figures 9.2 and 9.3 illustrate the result of our comparison of CDL with other action representations. CDL together with SPAR provide the most expressive state description languages and only FOPL provides a more expressive decoupled action representation than CDL. The latter is due to the generality of FOPL which essentially leads to the lack of methodology problem with other brokers mentioned in section 9.1.

Table 9.2 shows how the flexibility of CDL compares to that of other languages.

CDL allows for plugging in of arbitrary content languages and uses reflective reasoning to test for capability subsumption. While this much is true for the other flexible languages reviewed here, CDL is different in the way it handles unknown content languages and specifies a procedural interface to the content languages through optional functions.

Thus, CDL is an expressive and flexible action representation.

9.3 Other Domains

In this section we will briefly discuss and reflect on the adequacy of CDL *if it was applied to common domains used in related systems and approaches. It will demonstrate the generality of our approach.*

9.3.1 Brokers

We have already compared our broker to some other generic brokers in section 9.1. In this section we will consider some of the domains these brokers have been applied to and we will demonstrate that our broker, too, could handle these domains. The only limitation is the fact that our broker does not support the `subscribe` performative. As explained in section 9.1.2.3, the reason for the omission of this performative in our broker is not a crucial one, but simply that this performative was not required in our type of scenario. We believe that it would not be too difficult to extend our broker to support this performative though.

Another remark concerns the brokers we have compared our CDL broker to in section 9.1. These brokers were all developed as part of a larger agent system. The descriptions of the brokers cited in this thesis are in fact mostly descriptions of these systems of which the broker is just one agent. Thus, descriptions are brief and sometimes not illustrated with any example domains. As the most elaborate description is that of the ABSI facilitator [Singh, 1993a, Singh, 1993b] we have concentrated on the examples used to describe the behaviour of this broker. As will be seen this is still a rather trivial domain.

The domain chosen to illustrate the ABSI facilitator is that of information provision, e.g. the computation of factorials. To make this domain interesting they have implemented two PSAs for this task, one that computes the factorials of even numbers, and one that computes the factorials of odd numbers. The message with which the first of these agents advertises its capability to the facilitator is given as follows [Singh, 1993a, page 54]:

```
(package
  :content
    '(stash (<= (handles efact-agent (list-of 'factorial ?x))
                (= (denotation ?x) ?y)
                (even-integer ?y)))
  :sender 'efact-agent
  :reply-with nil)
```

The content of this capability-advertising message is fairly easy to understand: `(list-of 'factorial ?x)` represents the format of the message the `efact-agent` advertises it can process. This format expression is followed by two constraints defined in terms of functions known to the broker. The capability advertisement of the `ofact-agent` which computes factorials for odd numbers is almost identical to the above.

Now, the way the ABSI facilitator is implemented requires PHAs to send their problems to the broker directly, rather than having the facilitator recommend a PSA. For example, the facilitator might receive the following message:

```
(package
  :content
    (factorial 2)
  ... )
```

On receipt of this message the facilitator will attempt to match the content expression to an expression in a previously received capability advertisement and evaluate the according constraints. On success, the matching PSA will be asked to solve the problem, i.e. to perform the computation, and the result will be forwarded to the PHA.

Capabilities of this type can be expressed quite easily in CDL. Remember that a capability advertisement in CDL must have the following format:

```
(advertise ...  (<performative> ...  :content (<cdl-expression>)))
```

The `performative` must be the performative of the message the agent advertises it can process and the `cdl-expression` that is the content of this message must represent the capability. Returning to the example from the ABSI facilitator, the performative to be used here is `achieve` as the capability description

that is the content of this message will describe what can be achieved with this capability.

The CDL expression representing the above capability can be described as follows: The only input object is the argument of the function to be computed and there is one new output object that is the result of this computation or the the value of the expression to be evaluated.

The format of the expression is expressed in an output constraint by requiring it to be equal to the output parameter. Thus, the only input constraint that needs to be expressed is the fact that the given integer must be even. This could be done by specifying this predicate in first-order logic and using FOPL as the state language or by assuming a content language in which such a predicate is implemented directly. Assuming we have an according content language called `absiL`, the above capability can be described in CDL as follows:

```
(capability
  :state-language absiL
  :input ((Argument ?x)))
  :output ((Value ?y))
  :input-constraints ((even-integer ?x))
  :output-constraints ((== ?y (factorial ?x))))
```

This capability description is sufficient to emulate the behaviour of the ABSI facilitator. Thus, our CDL broker can be used to implement the scenarios used to illustrate the behaviour of the ABSI facilitator. However, the domain of the ABSI facilitator is concerned with information brokering rather than capability brokering. This is not our primary interest. Therefore, we shall now turn to another source for relevant example domains.

9.3.2 Planning Domains

As we have argued in section 4.2.1, capabilities and actions bear a close similarity. Domains involving the representation of actions have been investigated in AI planning, and these are the domains we shall consider next.

As we shall argue, CDL is perfectly adequate to represent many of the domains developed for AI planning scenarios. This is not a coincidence but true because we have designed CDL is such a way that these domains can be represented in CDL. When we designed CDL we first defined a set of properties we want this language to have (cf. section 4.1.1). We then performed a preliminary evaluation of representations reviewed in chapter 2 against these properties to determine which formalisms possess the properties we desire (cf. section 4.1.2). One result of this evaluation was that the knowledge we need to represent most closely resembles the knowledge represented in action representations as used in AI planning. Thus, we decided to base the structure of CDL on that of these action representations. This explains why most of the domains that can be represented in these formalisms can also be represented in CDL.

There are two groups of domains that cannot be represented in CDL though. Firstly, there are the rich domains developed for real world planners such as O-Plan or SIPE (cf. section 2.3.1.5). These include rich representations for time and resource constraints, for example. CDL does not provide for such rich domains as it is. We believe, however, that this richness is mostly found in the state language and thus, this limitation might be addressed by implementing a similarly rich state representation language and plugging it into CDL. Secondly, CDL does not support hierarchical action representations. The reason for this limitation is that the representation of how to refine an action is not the kind of knowledge we wanted to include in a capability description. Capabilities in CDL are meant to represent an exterior view of a performable action and not how this action may be performed or broken down into more primitive actions. Apart from these restrictions CDL should be suitable for the representation of any action from a planning domain.

To further substantiate this claim we have looked at the set of domains that come with the UCPOP planner.[2] There are eleven different domains ranging from

[2] The UCPOP planner (version 4.1) and the domains are available on the Internet at `http://www.cs.washington.edu/research/projects/ai/www/ucpop.html`.

rather simple domains like two different formalisations of the Blocks World to more complex domains such as the so-called flat-tyre domain with 14 different operators. Although we have not attempted to represent every operator defined in these domains in CDL, it is fairly obvious that such a translation would be quite straightforward.

For example, the operator for removing a wheel is represented as follows in the flat-tyre domain:

```
(define (operator remove-wheel)
    :parameters ((wheel ?x) (hub ?y))
    :precondition (:and (:neq ?x ?y) (:not (on-ground ?y))
                        (on ?x ?y) (unfastened ?y))
    :effects ((:effect (:and (have ?x) (free ?y)
                        (:not (on ?x ?y))))))
```

The parameters correspond to inputs and outputs in CDL. In this example both parameters represent objects that exist in the input situation. Preconditions and effects correspond to input and output constraints and only a change of syntax is required to translate them into the state language `lits` where only literals were allowed. The resulting capability description of the above operator in CDL can be given as follows:

```
(capability
  :state-language lits
  :input ((wheel ?x) (hub ?y))
  :input-constraints (
    (not (== ?x ?y))
    (not (on-ground ?y))
    (on ?x ?y)
    (unfastened ?y))
  :output-constraints (
    (have ?x)
    (free ?y)
    (not (on ?x ?y))))
```

Summary

To summarise, CDL can not only be used to represent capabilities in the Pacifica domain described in chapter 3, but it is also fairly straightforward to translate

the operators in domains of classical, non-hierarchical planners into capability descriptions in CDL. Furthermore, our broker can also emulate the information brokering behaviour of other brokers reviewed in this thesis. Thus, CDL can be considered a generic capability description language.

Chapter 10

Conclusions

At this point we have described and addressed the problem of capability brokering. We have defined a new capability description language that can be used for this purpose. We also have demonstrated and discussed two new and desirable properties of this language: its expressiveness and flexibility. The final step will be to summarise the argument presented in this thesis and reflect on it.

10.1 Possible Extensions

In this section we will indicate how the scenarios described in chapter 3 could be extended and how these extensions could be realized in the framework described in this thesis. This will show the extensibility as well as current limitations of our approach.

The development of extensions to our capability description language should always be driven by the problems, agents, and capabilities the broker needs to distinguish between. Of course, generality of the description language is one aim, but all features of the capability description language should be demonstrable in example scenarios like the ones described in chapter 3 in this thesis. There are two principal ways in which these scenarios could be extended.

Firstly, we could increase the number of PSAs in the scenarios. The broker would then have more PSAs and capabilities to choose from, and different agents with different capabilities may provide new challenges. We would not expect such an extension to lead to more communication between agents, apart from the advertisement messages of the additional agents. Thus, the basic framework described in this thesis would remain. New PSAs and capabilities are most likely to require an extended capability description language. An extended language in turn will require extended algorithms to reason over it. We believe that increasing the number of PSAs and capabilities in the scenarios will be inevitable in developing extensions to CDL.

Secondly, our scenarios could be further complicated by adding another problem-holding agent, e.g. a doctor who has an ill patient in one of the towns that requires hospital treatment. Depending on the time when this second problem arises the exact capabilities of the PSAs may have changed. For example, the **h1**-agent's sole ambulance might be at the power plant and thus not available. Having several PHAs is a very realistic extension to the scenarios and the broker should be able to cope with it. However, often tasks do not interact, which is essentially why the STRIPS assumption is reasonable. In these cases the CDL and the broker described are sufficient and no extension is required. If tasks interact through changing capabilities, one way of addressing the problem would be to send messages to the broker updating capabilities. Another way would be to leave this detail to the PSA and fail there.

10.1.1 Extensions based on more PSAs

We believe that an increased number of interestingly different PSAs leads to more interesting problems. For example, the capability descriptions of the different agents may be insufficient to decide on which one to recommend. With an increasing number of PSAs this is bound to happen at some point, e.g. if two agents advertise identical capability descriptions. In fact, the initial scenario already

illustrates this problem. The currently implemented solution is to recommend the first agent found capable of solving the given problem. While this solution works fine in our scenarios, it may be inappropriate elsewhere.

Another option would be to forward the problem to all the PSAs found capable of addressing it and ask them to perform an assessment of their own capabilities, i.e. to ask them how likely they think they are to find a solution to the given problem. This option would not require an extension to CDL and only a minor change in the brokering algorithms. Since we expect a capability description to be an abstraction of what can actually be done by a PSA, the capability-possessing PSAs may well have further detailed knowledge about their capabilities which they can use in a self-assessment. If the PSAs use search to solve the problem then using techniques based on inspecting a partial search space might help in choosing a PSA [Wickler and Pryor, 1996]. However, such a self-assessment cannot be comparative as the PSAs will, in general, only be aware of their own capabilities. Thus, it may not help the broker to decide which PSA to recommend if only one PSA is to be recommended.

Part of the problem here is that capabilities either subsume or do not subsume a given task in our framework. If capability evaluation was based on the notion of how well a capability can be used to address a given task, the broker could always recommend the best agent, which is far less likely to lead to ambiguity than the subsumption concept defined in this thesis. For example, the hospital closer to an emergency is more likely to be reached faster and thus, its capability better addresses the given task. In general, deciding how well a capability could address a given problem would require the broker to obtain knowledge about the solutions offered by the different PSAs and the utility these solutions have for the PHA. The crucial problem here is the representation of a utility function for the PHA. Unfortunately, the representation of generic utility functions is not well understood at present and some of the problems are discussed in [Russell and Norvig, 1995, pages 473–484].

A final problem that may arise in scenarios involving a large number of agents is related to the representation of capabilities and tasks. While the framework provides different types of parameters and constraints for the representation of capabilities, there is still a number of open representational choices. Current approaches to knowledge sharing consistently suggest the use of ontologies to narrow these choices. CDL already provides a framework for the representation of ontologies of actions and their incorporation into the brokering mechanism, but for the practical use of the broker the ontology will need to be filled in. This not only allows for the convenient expression of capabilities as performable actions, as described in this thesis, but also provides the PSA with some vocabulary to represent its capabilities. Without this vocabulary different agents may use different terminology to represent the same problems and capabilities, making it impossible for the broker to perform appropriate matching.

10.1.2 Other Extensions

Apart from extensions arising through an increased number of agents in the scenarios, there are also some limitations to our broker we are aware of which are related to the fact that it is a "proof of concept" rather than a complete product. For example, as we have pointed out in section 4.5.2, our broker does not attempt to manage the solution of problems, and neither do the PHAs we have implemented. However, we have also argued that capability descriptions usually cannot be guaranteed to be complete or even sound. This may lead to problems during the application of a capability or during the execution of a plan involving several agents' capabilities. For example, if one PSA fails to solve a problem, there is currently no way the PHA could ask the broker to recommend another PSA. If one or more agents fail in the performance of their capabilities while the broker manages a plan to solve a given problem, the broker needs to re-plan. Performance problems could provide a very interesting set of extensions to the current scenarios.

Another limitation of the current implementation is the way capabilities and problems based on different languages are handled. For example, a capability described in CDL using `fopl` as the state language and a problem described in CDL using `lits` which has to be evaluated against this capability will work because the two languages are based on the same abstract Java classes. A cleaner solution would be to have the broker offer an explicit translation service to other agents in which they could ask the broker to translate a given expression from one representation into another, if this is possible. The Enterprise Toolkit [Stader, 1997] implements an approach to such a service. The broker itself could use this explicit service to translate problems into an appropriate representation before performing the subsumption test. Currently the implementation of this translation is rather ad-hoc. We believe that the introduction of an explicit translation service by the broker would add to the flexibility, although such a service cannot be considered to operationalise part of the brokering process.

In conclusion, while the broker and language presented in this thesis present a comprehensive framework for the representation of and reasoning about capabilities of intelligent agents, there are also some practical issues that remain to be resolved before the work can be embedded in a large, realistic scenario. On the other hand, the framework provides a promising vehicle for basic research into capabilities.

10.2 Summary

In this section we will summarise the results of the work presented in this thesis. This will include what has been achieved as well as problems encountered.

10.2.1 Introduction of the Problem

In chapter 1 we introduced and described the problem of capability brokering, the main problem addressed in this thesis.

In this thesis we have addressed the problem of capability brokering which arises when intelligent agents communicate and cooperate. Finding an agent that can help one solve a given problem is at the heart of capability brokering. There are a variety of contexts in which capability brokering can take place and we have chosen to assume that: capabilities will be evaluated at run-time; problem-solving agents have domain knowledge; and problem-holding agents are interested in finding other agents that can solve the whole problem. To successfully address the problem of capability brokering we were aiming for a reasonably robust implementation of a broker and a capability description language that could be used to operationalise several scenarios. Furthermore, we expected the capability description language to be expressive and highly flexible.

10.2.2 Relevant Work in the Literature

In chapter 2 we reviewed work relevant to the problem of capability brokering in order to have an established foundation for our own work.

The first step towards understanding the problem of capability brokering consisted of a literature survey. As the problem arises when intelligent agents communicate and cooperate, we have looked at approaches from this area first. In fact, the problem of capability brokering had been addressed as the connection

problem in Distributed AI. The most interesting contributions for our work found in research into intelligent software agents were the generic agent communication languages, specifically KQML, which is highly flexible, and which a number of brokers were designed to utilise in recent years.

The second area we looked at are logics as capability representation formalisms. The best known logic, first-order predicate logic, has been used in the situation calculus to represent actions, but first-order logic is more naturally seen as a state representation language. We also reviewed some more advanced logics but none of these offered itself for the representation of agent capabilities. More promising was the approach to re-use meta-level knowledge, but ultimately this turned out to be inauspicious due to the utility problem. Terminological logics were interesting not so much as capability representations but because they have been used for the representation of ontologies and there is an interesting theory of expressiveness defined for these logics.

Representations geared more towards capabilities are action representations as used in AI planning, and this is the area we looked at next. A number of action representations have grown out of this area and one that stands out as the most influential is the STRIPS representation based on the STRIPS assumption. A more recent contribution we have taken up in our work is the development of ontologies of actions, although our work only provides a framework rather than an actual ontology. Processes can be seen as refined models of activity, but most of the work in this area attempts to model interactions between processes which is not a problem that needs to be addressed for capability brokering. Similarly, the problems addressed in work on agents that plan with capabilities, e.g. execution failure and re-planning, were not our concern although they are relevant.

A final area which inspired us rather than provided results is concerned with the modelling of problem-solving methods. These reasoning actions have been analysed mostly from a knowledge acquisition perspective in two major efforts: the KADS and the PROTÉGÉ project. However, their models and guidelines for

modelling are mostly based on informal representations. Currently the HPKB program is also aiming for models of problem-solving to greatly speed up the knowledge engineering process, but few results are available yet.

10.2.3 The Scenarios

> In chapter 3 we introduced a number of scenarios which defined the target behaviour we wanted our broker to exhibit.

Having looked at various areas that have represented knowledge similar to capability knowledge, we defined a number of scenarios which we wanted our broker and representation to handle. The first scenario was rather simple and it was aimed at introducing our domain, the island Pacifica with its agents, along with illustrating the basic exchange of messages we envisaged. Messages were described in KQML and the content was only given informally to illustrate what needed to be represented. The initial scenario was followed by two more complex scenarios that were meant to motivate and exemplify the two properties we desired for our capability description language: expressiveness and flexibility.

10.2.4 The Capability Description Language

> In chapter 4 we defined our capability description language, CDL, and illustrated this language with a number of examples from our scenarios.

Given the work we reviewed on capability brokering and representations along with the scenarios illustrating our aims, we were now in a position to evaluate previous work and see which ideas we could utilise for our new capability description language. For this purpose, we described several desirable characteristics for our language. The main result of this preliminary evaluation was that we wanted: to preserve the structure found in action representations; to benefit from the expressiveness of powerful logics; and to retain the flexibility of KQML.

This gave us a sufficient foundation to design our new capability description language based on the concept of achievable objectives. We first argued that capabilities are essentially actions and discussed the knowledge they contain: input and output parameters as well as several types of constraints on the situations before and after the capability has been applied. To implement a representation formalism for this knowledge we introduced the concept of a decoupled action representation language that separates states from actions. After the definition of the syntax in BNF we used this language to complete the messages from the initial scenario, thereby illustrating the language itself.

Based on this core language centred around the concept of achievable objectives we developed and presented two extensions. The first of these was based on the idea of performable actions. The idea here was to describe an action as a modified description of another action, thus allowing one to build a complex ontology of actions. The description of the syntax extension was again followed by examples from the initial scenario. The second extension was concerned with the representation of properties of the problem-solving agents which was achieved through a set of propositions added to the representation. Finally, since most of the examples used this far stemmed from the relatively simple initial scenario, we showed how CDL could be used to complete the messages required for the more complex scenarios, the expressiveness and flexibility scenario. Remember that these messages were described with an informal content initially.

10.2.5 Reasoning over CDL

In chapter 5 we defined and described the reasoning mechanisms and algorithms we implemented for capability brokering with CDL.

By defining the language we also defined the communication that was to take place in the various scenarios. However, equally important is the implementation of the language and the mechanisms that can be used to reason about it. For this purpose we have first formalised the internal representation in order to have

a precise definition of what a CDL capability description is. Next we introduced the most basic version of an algorithm which tests whether a given capability subsumes a given task. The concept of capability subsumption has been defined in terms of the logical entailment relation through the input and output match condition, and this definition provided the basis for the basic algorithm that evaluated capability subsumption. The algorithm was given as pseudo-code and applied to an example.

The basic algorithm suffered from a number of limitations that we imposed on the capability and the task for which subsumption was to be evaluated. The next step in our work was to relax these restrictions and extend the algorithm to deal with the resulting capability and task descriptions. The first step was to allow input-output constraints in the capability description which could be handled with an extension which was very similar to the original algorithm. Next we dealt with capabilities and tasks described as performable actions. The approach here was to translate a capability or task described as a performable action into one described in terms of achievable objectives before applying the algorithm described previously. This translation is effectively an instantiation of the capability or task. The last extension deals with agent properties. For all extensions we have also extended our definition of capability subsumption and illustrated the extensions with examples.

The final part of the description of our implementation of CDL and the broker concerned the embedding of our work into the Java Agent Template. After a brief description of JAT we described the CDL Interpreter which is effectively the broker attached to an agent name server. An interpreter is just one type of resource managed by a JAT agent, and our implementation also made use of another type of resource: JAT languages. The mechanisms provided by JAT for managing resources allowed an elegant implementation of CDL as a decoupled language. This was exemplified by tracing the reasoning and messages generated in the flexibility scenario.

10.2.6 Evaluation of the Broker

> In chapter 6 we presented the results of applying our broker to our scenarios and some variations on these which constitutes a practical evaluation of our work.

Having described the language and its implementation, it was time to evaluate what has been achieved in practise this far. The first question we had to address was how generic and robust the broker is. Extensive testing was beyond the scope of this thesis, but we used a number of variations of the expressiveness and flexibility scenario in order to evaluate broker performance. The result was that our broker performed well in virtually all cases. The next question then was how efficient the broker is. Actual response times were virtually instant, but this might have been due to the small number of PSAs in our scenarios. A detailed complexity analysis that followed essentially revealed that the complexity of evaluating capability subsumption mainly depends on the underlying state language used, but that this does not affect scaling issues. Thus, we showed that our broker is reasonably generic and robust, and that it exhibits adequate performance, i.e. we showed that our practical criteria for success have been achieved.

Next we turned to the more theoretical issues of expressiveness and flexibility.

10.2.7 Expressiveness of CDL

> In chapter 7 we defined and discussed a formal notion of expressiveness for action representation languages which could be used to compare such languages.

Having met the practical criteria for success outlined in the introduction, all that remained to be shown was that CDL possessed the two properties we claimed it has. We first looked at expressiveness, a property claimed for many representations but hardly ever formally defined. In fact, the first question we had to

answer was why we need expressiveness in our capability description language. The argument is mostly based on the expressiveness scenario and the potential for conciseness offered by expressiveness. To define what we meant by the expressiveness of an action representation we have looked at a very general framework for terminological KR languages. Based on the ideas found there we defined what it means for an action representation to be more expressive than another. While we were happy with this definition, an open question remains, namely whether one should impose an additional condition of polynomial transformability onto the definition.

The type of language for which we have defined the concept of expressiveness is what we called AR1 languages. The next step in our work was to define CDL as such an AR1 language. This required the definition of a state description language for which we used first-order logic, although CDL allows different state languages to be plugged in. Next we had to define the decoupled action representation language which expresses relations between states. The third component of an AR1 language is the model-restriction function which defines the semantics of the state language. The final component of an AR1 language, the action definition function, then defines the semantics of the actions based on the semantics of the state language. By defining CDL as an AR1 language we effectively also defined the semantics of this language.

10.2.8 Flexibility of CDL

> In chapter 8 we introduced our notion of flexibility and discussed a number of problems that arise during the implementation of flexible languages.

Flexibility turned out to be an entirely different matter. To begin with, we again had to answer the question of why we need flexibility in our capability description language. The argument was based on the flexibility scenario and a number of examples of further state representation languages one might want to

plug into CDL in different scenarios. However, while expressiveness was a relatively well-understood concept, flexibility is new. Thus, we did not even attempt to formalise it. One issue we discussed is the trade-offs that flexibility allows one to resolve at a later time, and this is how we informally defined flexibility. The most interesting issues arose out of the question of how a decoupled knowledge representation language can be implemented and we described how we addressed various problems in the implementation of CDL, e.g. reflective reasoning or parsing. Finally, we have also argued that decoupled action representations provide us with a deeper understanding of actions and their representations.

10.2.9 Evaluation of CDL

> In chapter 9 we evaluated CDL by comparing it to languages used by other brokers and by comparing its expressiveness and flexibility to that of other relevant formalisms.

Having defined expressiveness and flexibility, the next step was an evaluation of CDL in this respect. This has been achieved though a comparison of CDL and our broker with other brokers. There we showed that, while there is no significant difference in supported brokering performatives, the matching algorithm implemented in CDL and based on the notion of capability subsumption (cf. definition 5.4) is significantly more powerful than the matching provided by other brokers. To evaluate expressiveness, we have compared CDL with various action representations which are the formalisms for representing the same type of entity. For this comparison other action representation were treated as if they were also decoupled languages. In summary, CDL can be described as an expressive action representation. For flexibility, we have compared CDL with languages that offer at least some degree of this property. Again, the result of this comparison reveals that CDL is a highly flexible language. Finally, we have demonstrated the generality of our broker by applying to different domains from AI planning and other brokers.

10.2.10 Conclusions

To summarise, we have described and addressed the problem of capability brokering. To address this problem we have presented a new capability description language that possesses two desirable properties: it is expressive and highly flexible. These were two of the criteria for success we set out, others being that our broker be reasonably robust and efficient. As we have shown in chapters 6 and 9, our broker and CDL do indeed meet all the criteria for success set out in the introduction. We showed this in a number of representative scenarios and compared our work with other related work to show what we have achieved and that CDL is indeed the generic capability description language that can be used for capability brokering we set out to create.

Bibliography

[Aamodt *et al.*, 1993] Agnar Aamodt, Bart Benus, Cuno Duursma, Christine Tomlinson, Ronald Schrooten, and Walter Van de Velde. Task features and their use in CommonKADS. Deliverable D 1.5, Free University of Brussels, Brussels, Belgium, January 1993.

[Aben, 1995] Manfred Aben. *Formal Methods in Knowledge Engineering.* PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, February 1995.

[Aitken *et al.*, 1998] Stuart Aitken, Ian Filby, John Kingston, and Austin Tate. Capability descriptions for problem-solving methods. AIAI, University of Edinburgh, Edinburgh, Scotland (HPKB Deliverable), January 1998.

[Allen *et al.*, 1990] James Allen, James Hendler, and Austin Tate, editors. *Readings in Planning.* Morgan Kaufmann, San Mateo, CA, 1990.

[Ambros-Ingerson and Steel, 1988] José A. Ambros-Ingerson and Sam Steel. Integrating planning, execution and monitoring. In *Proc. 7th AAAI*, pages 83–88, Saint Paul, MN, August 1988. Morgan Kaufmann. Also in [Allen *et al.*, 1990, pages 735–740].

[Anderson, 1981] John R. Anderson. Tuning of search of the problem space for geometry proofs. In *Proc. 7th IJCAI*, pages 165–170, Vancouver, Canada, August 1981. University of British Columbia, William Kaufmann.

[Armengol *et al.*, 1998] Eva Armengol, Richard Benjamins, Stefan Decker, Dieter Fensel, Enrico Motta, Rudi Studer, and Bob Wielinga. State of the art deliverable. Deliverable D1.4, University of Amsterdam, Amsterdam, The Netherlands, May 1998.

[Attardi and Simi, 1984] Giuseppe Attardi and Maria Simi. Metalanguage and reasoning across viewpoints. In Tim O'Shea, editor, *Proc. 6th ECAI*, pages 413–422, Pisa, Italy, September 1984. North-Holland.

[Baader, 1996] Franz Baader. A formal definition for the expressive power of terminological knowledge representation languages. *Journal of Logic and Computation*, 6(1):33–54, February 1996.

[Bäckström, 1995] Christer Bäckström. Expressive equivalence of planning formalisms. *Artificial Intelligence*, 76:17–34, 1995.

[Baker *et al.*, 1997] Sean Baker, Vinny Cahill, and Paddy Nixon. Bridging boundaries: CORBA in perspective. *IEEE Internet Computing*, 1(5):52–57, 1997.

[Barr, 1979] Avron Barr. Meta-knowledge and cognition. In *Proc. 6th IJCAI*, pages 31–33, Tokyo, Japan, August 1979. William Kaufmann.

[Barrett and Weld, 1994] Anthony Barrett and Daniel S. Weld. Partial-order planning: Evaluating possible efficiency gains. *Artificial Intelligence*, 67:71–112, 1994.

[Barrett *et al.*, 1995] Anthony Barrett, Dave Christianson, Marc Friedman, Chung Kwok, Keith Golden, Scott Penberthy, Ying Sun, and Daniel Weld. UCPOP user's manual (version 4.0). Technical Report 93-09-06d, University of Washington, Seattle, WA, November 1995.

[Barros *et al.*, 1996] Leliane Barros, André Valente, and Richard Benjamins. Modeling planning tasks. In Brian Drabble, editor, *Proc. 3rd International Conference on Artificial Intelligence Planning Sytems*, pages 11–18, Edinburgh, Scotland, May 1996. AAAI Press.

[Bayardo *et al.*, 1997] R. Bayardo, W. Bohrer, R. Brice, A. Cichocki, G. Fowler, A. Helal, V. Kashyap, T. Ksiezyk, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishnan, A. Unruh, and D. Woelk. Semantic integration of information in open and dynamic environments. In Joan M. Peckman, editor, *Proc. ACM SIGMOD International Conference on Management of Data*, Tucson, AZ, May 1997. ACM Press.

[Benech and Desprats, 1997] D. Benech and T. Desprats. A KQML-CORBA based architecture for intelligent agents communication in cooperative service and network management. In *Proc. IFIP/IEEE InternationalConference on Management of Multimedia Networks and Services*, Montréal, Canada, July 1997.

[Benjamins *et al.*, 1997] Richard Benjamins, Dieter Fensel, and B. Chandrasekaran. PSMs do IT! In *Proc. IJCAI Workshop on Problem-Solving Methods for Knowledge-Based Systems*, Nagoya, Japan, August 1997.

[Benjamins *et al.*, 1998] Richard Benjamins, Enric Plaza, Enrico Motta, Dieter Fensel, Rudi Studer, Bob Wielinga, Guus Schreiber, and Zdenek Zdrahal. IBROW^3 — an intelligent brokering service for knowledge-component reuse on the world-wide web. In *Proc. 11th Workshop on Knowledge Acquisition, Modeling and Management*, Banff, Canada, April 1998.

[Blum and Furst, 1995] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. In *Proc. 14th IJCAI*, pages 1636–1642, Montréal, Canada, August 1995. Morgan Kaufmann.

[Bond and Gasser, 1988] Alan H. Bond and Les Gasser, editors. *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann, San Mateo, CA, 1988.

[Brachman and Levesque, 1985] Ronald J. Brachman and Hector J. Levesque, editors. *Readings in Knowledge Representation*. Morgan Kaufmann, Los Altos, CA, 1985.

[Brachman and Schmolze, 1985] Ronald J. Brachman and James G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, April 1985.

[Brachman, 1979] Ronald J. Brachman. On the epistemological status of semantic networks. In Nicholas V. Findler, editor, *Associative Networks*, pages 3–50. Academic Press, New York, NY, 1979.

[Bradshaw, 1997] Jeffrey M. Bradshaw, editor. *Software Agents*. AAAI Press/ The MIT Press, Menlo Park, CA/Cambridge MA, 1997.

[Brazier *et al.*, 1995] Frances M. T. Brazier, Jan Treur, and Niek J. E. Wijngaards. Modelling interaction with experts: The role of a shared task model. Technical Report IR-382, Free University of Amsterdam, Amsterdam, The Netherlands, 1995.

[Breuker and Van de Velde, 1994] Joost A. Breuker and Walter Van de Velde, editors. *CommonKADS Library for Expertise Modelling*. IOS Press, Amsterdam, The Netherlands, 1994.

[Breuker and Wielinga, 1989] Joost Breuker and Bob Wielinga. Models of expertise in knowledge acquisition. In Giovanni Guida and Carlo Tasso, editors, *Topics in Expert System Design*, chapter 5, pages 265–295. Elsevier Science Publishers, Amsterdam, The Netherlands, 1989.

[Breuker *et al.*, 1987] Joost Breuker, Bob Wielinga, Maarten van Someren, Robert de Hoog, Guus Schreiber, Paul de Greef, Bert Bredeweg, Jan Wielemaker, Jean-Paul Billault, Massoud Davoodi, and Simon Hayward. Model driven knowledge acquisition: Interpretation models. KADS Deliverable Task A1, University of Amsterdam, Amsterdam, The Netherlands, 1987.

[Breuker, 1997] Joost Breuker. Problems in indexing problem-solving methods. In *Proc. IJCAI Workshop on Problem-Solving Methods for Knowledge-Based Systems*, Nagoya, Japan, August 1997.

[Brewka, 1991] Gerhard Brewka. *Nonmonotonic Reasoning: Logical Foundations of Commonsense*. Cambridge University Press, Cambridge, UK, 1991.

[Brooks, 1986] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.

[Brooks, 1991] R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.

[Bundy and Welham, 1981] Alan Bundy and Bob Welham. Using meta-level inference for selective application of multiple rewrite rule sets in algebraic manipulation. *Artificial Intelligence*, 16(2):189–212, 1981.

[Bundy *et al.*, 1979] Alan Bundy, Lawrence Byrd, George Luger, Chris Mellish, and Martha Palmer. Solving mechanics problems using meta-level inference. In *Proc. 6th IJCAI*, pages 1017–1027, Tokyo, Japan, August 1979. William Kaufmann.

[Bylander, 1994] Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1–2):165–204, September 1994.

[Campione and Walrath, 1998] Mary Campione and Kathy Walrath. *The Java Tutorial: Object-Oriented Programming for the Internet.* The Java Series. Addison-Wesley Longman, 2nd edition, 1998.

[Carbonell *et al.*, 1992] Jaime G. Carbonell, Jim Blythe, Oren Etzioni, Yolanda Gil, Robert Joseph, Dan Kahn, Craig Knoblock, Steven Minton, Alicia Pérez, Scott Reilly, Manuela Veloso, and Xuemei Wang. PRODIGY 4.0: The manual and tutorial. Technical Report CMU-CS-92-150, Carnegie Mellon University, Pittsburgh, PA, June 1992.

[Chagrov and Zakharyaschev, 1997] Alexander Chagrov and Michael Zakharyaschev. *Modal Logic.* Clarendon Press, Oxford, UK, 1997.

[Chaib-Draa *et al.*, 1992] B. Chaib-Draa, B. Moulin, R. Mandiau, and P. Millot. Trends in distributed artificial intelligence. *Artificial Intelligence Review*, 6(1):35–66, 1992.

[Chang and Lee, 1973] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving.* Computer Science and Applied Mathematics Series. Academic Press, New York, NY, 1973.

[Charniak and McDermott, 1985] Eugene Charniak and Drew McDermott. *Introduction to Artificial Intelligence.* Computer Science Series. Addison-Wesley, Reading, MA, 1985.

[Chase *et al.*, 1989] Melissa P. Chase, Monte Zweben, Richard L. Piazza, John D. Burger, Paul P. Maglio, and Haym Hirsh. Approximating learned search control knowledge. In Alberto Maria Segre, editor, *Proc. 6th International Workshop on Machine Learning*, pages 218–220, Ithaca, NY, June 1989. Cornell University, Morgan Kaufmann.

[Chellas, 1980] Brian F. Chellas. *Modal Logic: An Introduction.* Cambridge University Press, Cambridge, UK, 1980.

[Chi *et al.*, 1981] Michelene T. H. Chi, Paul J. Feltovich, and Robert Glaser. Categorization and representation of physics problems by experts and novices. *Cognitive Science*, 5(2):121–152, April 1981.

[Cohen and Levesque, 1990] Paul R. Cohen and Hector J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.

[Cohen and Levesque, 1995] Philip R. Cohen and Hector J. Levesque. Communicative actions for artificial agents. In Victor Lesser, editor, *Proc. 1st International Conference on Multi-Agent Systems*, pages 65–72, San Francisco, CA, June 1995. AAAI Press/The MIT Press.

[Cohen *et al.*, 1989] Paul R. Cohen, M. L. Greenberg, D. M. Hart, and A. E. Howe. Trial by fire: Understanding the design requirements for agents in complex environments. *AI Magazine*, 10(3):32–48, Autumn 1989.

[Cohen *et al.*, 1998] Paul Cohen, Robert Schrag, Eric Jones, Adam Pease, Albert Lin, Barbara Starr, David Gunning, and Murray Bruke. The DARPA high-performance knowledge bases project. *AI Magazine*, 19(4):25–49, Winter 1998.

[CORBA V2.2, 1998] The Object Management Group. *The Common Object Request Broker: Architecture and Specification*, February 1998.

[Croft, 1985] David Croft. Choice making in planning systems. In Martin Merry, editor, *Proc. 5th Expert Systems Conference*, pages 125–141, Warwick, UK, December 1985. University of Warwick, Cambridge University Press.

[Currie and Tate, 1991] Ken Currie and Austin Tate. O-Plan: The open planning architecture. *Artificial Intelligence*, 52(1):49–86, 1991.

[Davis and Buchanan, 1977] Randall Davis and Bruce G. Buchanan. Meta-level knowledge: Overview and applications. In *Proc. 5th IJCAI*, pages 920–927, Cambridge, MA, August 1977. MIT, William Kaufmann. Also in: [Brachman and Levesque, 1985, pages 389–396].

[Davis and Smith, 1983] Randall Davis and Reid G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20(1):63–109, 1983. Also in: [Bond and Gasser, 1988, pages 333–356].

[Davis, 1980] Randall Davis. Meta-rules: Reasoning about control. *Artificial Intelligence*, 15(3):179–222, 1980.

[Davis, 1990] Ernest Davis. *Representations of Commonsense Knowledge.* Morgan Kaufmann, San Mateo, CA, 1990.

[Decker *et al.*, 1997] Keith Decker, Katia Sycara, and Mike Williamson. Middle-agents for the internet. In *Proc. 15th IJCAI*, pages 578–583, Nagoya, Japan, August 1997. Morgan Kaufmann.

[Decker *et al.*, 1998] Setfan Decker, Michael Erdmann, Dieter Fensel, and Rudi Studer. Reasoning with metadata: Ontobroker. University of Karlsruhe, Karlsruhe, Germany, 1998.

[Doyle, 1997] Jon Doyle. Problem-solving method language proposal. MIT, Cambridge, MA, October 1997.

[Eckel, 1997] Bruce Eckel. *Thinking in Java.* Prentice Hall, 1997.

[Eriksson *et al.*, 1995] Henrik Eriksson, Yuval Shahar, Samson W. Tu, Angel R. Puerta, and Mark A. Musen. Task modeling with reusable problem-solving methods. *Artificial Intelligence*, 79(2):293–326, December 1995.

[Eskey and Zweben, 1990] Megan Eskey and Monte Zweben. Learning search control for constraint-based scheduling. In *Proc. 8th AAAI*, pages 908–915, Boston, MA, August 1990. AAAI Press/The MIT Press.

[Etzioni and Minton, 1992] Oren Etzioni and Steven Minton. Why EBL produces overly-specific knowledge: A critique of the PRODIGY approaches. In Derek Sleeman and Peter Edwards, editors, *Proc. 9th International Workshop on Machine Learning*, pages 137–143, Aberdeen, Scotland, July 1992. Morgan Kaufmann.

[Etzioni *et al.*, 1992] Oren Etzioni, Steve Hanks, Daniel Weld, Denise Draper, Neal Lesh, and Mike Williamson. An approach to planning with incomplete information. In Bernhard Nebel, Charles Rich, and William Swartout, editors, *Proc. 3rd KR*, pages 115–125, Cambridge, MA, October 1992. Morgan Kaufmann.

[Etzioni *et al.*, 1993] Oren Etzioni, Henry M. Levy, Richard B. Segal, and Chandramohan A. Thekkath. OS agents: Using AI techniques in the operating system environment. Technical Report 93-04-04, University of Washington, Seattle, WA, April 1993.

[Etzioni, 1997] Oren Etzioni. Moving up the information food chain. *AI Magazine*, 18(2):11–18, Summer 1997.

[Fagin *et al.*, 1995] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning about Knowledge.* The MIT Press, Cambridge, MA, 1995.

[Farquahar *et al.*, 1996] A. Farquahar, R. Fikes, and J. Rice. The Ontolingua server: A tool for collaborative ontology construction. Technical Report KSL 96-26, Stanford University, Stanford, CA, September 1996.

[Fensel *et al.*, 1998a] Dieter Fensel, Richard Benjamins, Stefan Decker, Mauro Gaspari, Rix Groenboom, Enrico Motta, Enric Plaza, Guus Schreiber, Rudi Studer, and Bob Wielinga. UPML: The very high idea. University of Karlsruhe, Karlsruhe, Germany, 1998.

[Fensel *et al.*, 1998b] Dieter Fensel, Richard Benjamins, Stefan Decker, Mauro Gaspari, Rix Groenboom, Enrico Motta, Enric Plaza, Guus Schreiber, Rudi Studer, and Bob Wielinga. The unified problem-solving method description language UPML (version 1.0.7). University of Karlsruhe, Karlsruhe, Germany, July 1998.

[Fensel, 1997] Dieter Fensel. An ontology-based broker: Making problem-solving method reuse work. In *Proc. IJCAI Workshop on Problem-Solving Methods for Knowledge-Based Systems*, Nagoya, Japan, August 1997.

[Fernández *et al.*, 1997] M. Fernández, A. Gómez-Pérez, and N. Juristo. METHONTOLOGY: From ontological art towards ontological engineering. In *Working Notes of the AAAI Spring Symposium on Ontological Engineering*, Stanford, CA, March 1997. Stanford University, AAAI Press.

[Fikes and Nilsson, 1971] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):189–208, 1971. Also in: [Allen *et al.*, 1990, pages 88–97].

[Fikes *et al.*, 1972] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4):251–288, 1972.

[Fikes *et al.*, 1991] Richard Fikes, Mark Cutkosky, Tom Gruber, and Jeffrey Van Baalen. Knowledge sharing technology—project overview. Technical Report KSL 91-71, Stanford University, Stanford, CA, November 1991.

[Filman *et al.*, 1983] Robert E. Filman, John Lamping, and Fanya S. Montalvo. Meta-language and meta-reasoning. In *Proc. 8th IJCAI*, pages 365–369, Karlsruhe, Germany, August 1983. William Kaufmann.

[Finin *et al.*, 1992] Tim Finin, Don McKay, and Rich Fritzson. An overview of KQML: A knowledge query and manipulation language. Technical report, UMBC, Baltimore, MD, March 1992.

[Finin *et al.*, 1993] Tim Finin, Jay Weber, Gio Wiederhold, Michael Genesereth, Richard Fritzson, Donald McKay, James McGuire, Richard Pelavin, Stuart Shapiro, and Chris Beckauthor. Specification of the KQML agent-communication language. Technical report, The DARPA Knowledge Sharing Initiative External Interfaces Working Group, June 1993.

[Finin *et al.*, 1997] Tim Finin, Yannis Labrou, and James Mayfield. KQML as an agent communication language. In Jeffrey M. Bredshaw, editor, *Software Agents*, chapter 14, pages 291–316. AAAI Press/MIT Press, Menlo Park, CA/Cambridge, MA, 1997.

[Fisher, 1994] M. Fisher. A survey of concurrent METATEM—the language and its applications. In D. M. Gabbay and H. J. Ohlbach, editors, *Proc. 1st International Conference on Temporal Logic*, pages 480–505. Springer, 1994. LNAI 827.

[Forbus, 1984] Kenneth D. Forbus. Qualitative process theory. *Artificial Intelligence*, 24:85–168, 1984.

[Fox *et al.*, 1989] Mark S. Fox, Norman Sadeh, and Can Baykan. Constraied heuristic search. In *Proc. 11th IJCAI*, pages 309–315, Detroit, MI, August 1989. Morgan Kaufmann.

[Gallier, 1986] Jean H. Gallier. *Logic for Computer Science.* Harper and Row, New York, NY, 1986.

[Garey and Johnson, 1979] Michael R. Garey and David S. Johnson. *Computers and Intractability.* W. H. Freeman, New York, NY, 1979.

[Genesereth and Ketchpel, 1994] Michael R. Genesereth and Steven P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):48–53, 147, July 1994.

[Genesereth and Singh, 1993] Michael R. Genesereth and Narinder Singh. A knowledge sharing approach to software interoperation. Report Logic-93-12, Stanford University, Stanford, CA, February 1993.

[Genesereth *et al.*, 1992] Michael R. Genesereth, Richard E. Fikes, Daniel Bobrow, Ronald Brachman, Thomas Gruber, Patrick Hayes, Reed Letsinger, Vladimir Lifschitz, Robert MacGregor, John McCarthy, Peter Norvig, Ramesh Patil, and Len Schubert. Knowledge interchange format version 3.0 reference manual. Report Logic-92-1, Stanford University, Stanford, CA, June 1992.

[Genesereth, 1991] Michael R. Genesereth. Knowledge interchange format. In *Proc. 2nd KR*, pages 599–600, Cambridge, MA, 1991. Morgan Kaufmann.

[Gennari *et al.*, 1998] John H. Gennari, William Grosso, and Mark Musen. A method-description language: An initial ontology with examples. In *Proc. 11th Workshop on Knowledge Acquisition, Modeling and Management*, Banff, Canada, April 1998.

[Georgeff, 1982] Michael P. Georgeff. Procedural control in production systems. *Artificial Intelligence*, 18(2):175–201, March 1982.

[Georgeff, 1987] Michael P. Georgeff. Planning. *Annual Reviews in Computing Science*, 2:359–400, 1987. Also in: [Allen *et al.*, 1990, pages 5–25].

[Ghallab *et al.*, 1998] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. *PDDL—The Planning Domain Definition Language.* Yale University, New Haven, CT, March 1998. Draft 1.0.

[Ginsberg, 1986] Allen Ginsberg. A metalinguistic approach to the construction of knowledge base refinement systems. In *Proc. 5th AAAI*, pages 436–441, Philadelphia, PA, August 1986. Morgan Kaufmann.

[Ginsberg, 1987] Matthew L. Ginsberg, editor. *Readings in Nonmonotonic Reasoning.* Morgan Kaufmann, Los Altos, CA, 1987.

[Ginsberg, 1993] Matt Ginsberg. *Essentials of Artificial Intelligence.* Morgan Kaufmann, San Francisco, CA, 1993.

[Ginsberg, 1996a] Matthew L. Ginsberg. Do computers need common sense? In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *Proc. 5th KR*, pages 620–626, Cambridge, MA, November 1996. Morgan Kaufmann.

[Ginsberg, 1996b] Matthew L. Ginsberg. A new algorithm for generative planning. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *Proc. 5th KR*, pages 186–197, Cambridge, MA, November 1996. Morgan Kaufmann.

[Golding *et al.*, 1987] Andrew Golding, Paul S. Rosenbloom, and John E. Laird. Learning general search control from outside guidance. In *Proc. 10th IJCAI*, pages 334–337, Milan, Italy, August 1987. Morgan Kaufmann.

[Gómez-Pérez, 1998] A. Gómez-Pérez. Knowledge sharing and reuse. In Liebowitz, editor, *Handbook of Applied Expert Systems.* CRC, 1998.

[Green, 1969] Cordell Green. Application of theorem proving to problem solving. In Donald E. Walker and Lewis M. Norton, editors, *Proc. 1st IJCAI*, pages 219–239, Washington, D.C., August 1969. Morgan Kaufmann. Also in: [Allen *et al.*, 1990, pages 67–87].

[Gruber, 1992] Thomas R. Gruber. Ontolingua: A mechanism to support portable onotologies. Technical Report KSL 91-66, Stanford University, Stanford, CA, June 1992.

[Gruber, 1993a] Thomas R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. Technical Report KSL 93-04, Stanford University, Stanford, CA, August 1993.

[Gruber, 1993b] Thomas R. Gruber. A translation approach to portable ontologies. *Knowledge Acquisition*, 5(2):199–220, 1993.

[Gruninger and Fox, 1994] Michael Gruninger and Mark S. Fox. An activity ontology for enterprise modelling. In *Proc. Workshop on Enabling Technologies—Infrastructures for Collaborative Enterprises*. West Virginia University, 1994.

[Gruninger *et al.*, 1997] Michael Gruninger, C. Schlenoff, A. Knutilla, and S. Ray. Using process requirements as the basis for the creation and evaluation of process ontologies for enterprise modeling. *ACM SIGGROUP Bulletin Special Issue on Enterprise Modelling*, 18(3), 1997.

[Guha and Lenat, 1990] R. V. Guha and Douglas B. Lenat. Cyc: A midterm report. *AI Magazine*, 11(3):32–59, Fall 1990.

[Guha and Lenat, 1994] R. V. Guha and Douglas B. Lenat. Enabling agents to work together. *Communications of the ACM*, 37(7):127–142, July 1994.

[Haggith, 1995] Mandy Haggith. A meta-level framework for exploring conflicts in multiple knowledge bases. In John Hallam, editor, *Hybrid Problems, Hybrid Solutions*, pages 87–98. IOS Press, Amsterdam, The Netherlands, 1995.

[Harel *et al.*, 1982] David Harel, Dexter Kozen, and Rohit Parikh. Process logic: Expressiveness, decidability, completeness. *Journal of Computer and System Sciences*, 25:144–170, 1982.

[Harel, 1984] David Harel. Dynamic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic Vol. II*, chapter 10, pages 497–604. D. Reidel Publishing Company, 1984.

[Hayes, 1974] Patrick J. Hayes. Some problems and non-problems in representation theory. In *Proc. AISB Summer Conference*, pages 63–79, University of Sussex, 1974. Also in: [Brachman and Levesque, 1985, pages 4–22].

[Hendrix, 1973] Gary G. Hendrix. Modeling simultaneous actions and continuous processes. *Artificial Intelligence*, 4:145–180, 1973. Also in: [Weld and de Kleer, 1990, pages 64–82].

[Hintikka, 1962] J. Hintikka. *Knowledge and Belief*. Cornell University Press, Ithaca, NY, 1962.

[Hirst, 1991] Graeme Hirst. Existence assumptions in knowledge representation. *Artificial Intelligence*, 49(1–3):199–242, May 1991.

[Howe and Dreilinger, 1997] Adele E. Howe and Daniel Dreilinger. SAVVYSEARCH: A metasearch engine that learns which search engines to query. *AI Magazine*, 18(2):19–25, Summer 1997.

[Huhns and Singh, 1998] Michael N. Huhns and Munindar P. Singh, editors. *Readings in Agents*. Morgan Kaufmann, San Francisco, CA, 1998.

[Ihrig and Kambhampati, 1997] Laurie H. Ihrig and Subbarao Kambhampati. Storing and indexing plan derivations through explanation-based analysis of retrieval failures. *Journal of Artificial Intelligence Research*, 7:161–198, November 1997.

[Jennings, 1996] Nicholas R. Jennings, editor. *Foundations of Distributed Artificial Intelligence*. Wiley, New York, NY, 1996.

[Joslin and Pollack, 1996] David Joslin and Martha E. Pollack. Is "early commitment" in plan generation ever a good idea. In *Proc. 13th AAAI*, pages 1188–1193, Portland, OR, August 1996. AAAI Press/The MIT Press.

[Kambhampati and Yang, 1996] Subbarao Kambhampati and Xiuping Yang. On the role of disjunctive representations and constraint propagation in refinement planning. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *Proc. 5th KR*, pages 135–146, Cambridge, MA, November 1996. Morgan Kaufmann.

[Kambhampati *et al.*, 1996] Subbarao Kambhampati, Suresh Katukam, and Yong Qu. Failure-driven dynamic search control for partial order planners: An explanation-based approach. *Artificial Intelligence*, 88(1–2):253–315, December 1996.

[Kambhampati, 1997] Subbarrao Kambhampati. Challenges in bridging plan synthesis paradigms. In *Proc. 15th IJCAI*, pages 44–49, Nagoya, Japan, August 1997. Morgan Kaufmann.

[Kautz and Selman, 1992] Henry Kautz and Bart Selman. Planning as satisfiability. In Bernd Neumann, editor, *Proc. 10th ECAI*, pages 359–363, Vienna, Austria, August 1992. Wiley.

[Kautz and Selman, 1996] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. 13th AAAI*, pages 1194–1201, Portland, OR, August 1996. AAAI Press/The MIT Press.

[Kingston *et al.*, 1996] John Kingston, Nigel Shadbolt, and Austin Tate. CommonKADS models for knowledge-based planning. In *Proc. 13th AAAI*, pages 477–482, Portland, OR, August 1996. AAAI Press/The MIT Press.

[Konolige, 1986] Kurt Konolige. *A Deduction Model of Belief*. Morgan Kaufmann, San Mateo, CA, 1986.

[Kornfeld, 1979] William A. Kornfeld. ETHER–a parallel problem solving system. In *Proc. 6th IJCAI*, pages 490–492, Tokyo, Japan, August 1979. William Kaufmann.

[Kornfeld, 1981] William A. Kornfeld. The use of parallelism to implement a heuristic search. In *Proc. 7th IJCAI*, pages 575–580, Vancouver, Canada, August 1981. University of British Columbia, William Kaufmann.

[Kripke, 1963] S. Kripke. Semantical analysis of modal logic. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[Kuokka and Harada, 1995a] Daniel Kuokka and Larry Harada. Matchmaking for information agents. In *Proc. 14th IJCAI*, pages 672–678, Montréal, Canada, August 1995. Morgan Kaufmann.

[Kuokka and Harada, 1995b] Daniel Kuokka and Larry Harada. On using KQML for matchmaking. In *Proc. 1st International Conference on Multi-Agent Systems*, pages 239–245, San Francisco, CA, June 1995. AAAI Press/MIT Press.

[Kuokka, 1990] Daniel Kuokka. *The Deliberative Integration of Planning, Execution, and Learning.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1990.

[Labrou and Finin, 1997] Yannis Labrou and Tim Finin. A proposal for a new KQML specification. TR CS-97-03, University of Maryland Baltimore County, Baltimore, MD, February 1997.

[Laird *et al.*, 1987] John E. Laird, Allen Newell, and Paul S. Rosenblum. SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, 1987.

[Larkin *et al.*, 1980] Jill H. Larkin, John McDermott, Dorothea P. Simon, and Herbert A. Simon. Models of competence in solving physics problems. *Cognitive Science*, 4(4):317–345, October 1980.

[Laske, 1986] Otto E. Laske. On competence and performance notions in expert system design: A critique of rapid prototyping. In *Proc. 6th International Workshop Expert Systems and their Applications*, pages 257–297, Avignon, France, April 1986.

[Leckie and Zukerman, 1991] Christopher Leckie and Ingrid Zukerman. Learning search control rules for planning: An inductive approach. In Lawrence A. Birnbaum and Gregg C. Collins, editors, *Proc. 8th International Workshop on Machine Learning*, pages 422–426, Evanston, IL, June 1991. Northwestern University, Morgan Kaufmann.

[Lecoeuche *et al.*, 1996] Renaud Lecoeuche, Oliver Catinaud, and Catherine Gréboval-Barry. Competence in human beings and knowledge-based systems. In *Proc. 10th Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada, November 1996.

[Lee *et al.*, 1996] Jintae Lee, Micheal Grunninger, Yan Jin, Thomas Malone, Austin Tate, Gregg Yost, and other members of the PIF Working Group. The PIF process interchange format and framework version 1.1. Working Paper #194, MIT Center for Coordination Science, Cambridge, MA, May 1996.

[Lee *et al.*, 1998] Jintae Lee, Michael Grunninger, Yan Jin, Thomas Malone, Austin Tate, and Gregg Yost. The Process Interchange Format and framework. *The Knowledge Engineering Review*, 13(1):91–120, March 1998.

[Lenat *et al.*, 1983] Douglas B. Lenat, Randall Davis, Jon Doyle, Michael Genesereth, Ira Goldstein, and Howard Schrobe. Reasoning about reasoning. In Frederick Hayes-Roth, Donald A. Waterman, and Douglas B. Lenat, editors, *Building Expert Systems*, chapter 7, pages 219–239. Addison-Wesley, Reading, MA, 1983.

[Lenat, 1995] Douglas B. Lenat. CYC: A large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11):33–38, November 1995.

[Lesp'erance, 1989] Yves Lesp'erance. A formal account of self-knowledge and action. In *Proc. 11th IJCAI*, pages 868–874, Detroit, MI, August 1989. Morgan Kaufmann.

[Levesque, 1984] Hector J. Levesque. A logic of implicit and explicit belief. In *Proc. 4th AAAI*, pages 198–202, Austin, TX, August 1984. University of Texas, William Kaufman.

[Lifschitz, 1986] Vladimir Lifschitz. On the semantics of STRIPS. In Michael P. Georgeff and Amy L. Lansky, editors, *Proc. Workshop on Reasoning about Actions and Plans*, pages 1–9, Timberline, Oregon, July 1986. Morgan Kaufmann. Also in: [Allen *et al.*, 1990, pages 523–530].

[Loveland, 1978] Donald W. Loveland. *Automated Theorem Proving: A Logical Basis.* Fundamental Studies in Computer Science Vol. 6. North-Holland, Amsterdam, The Netherlands, 1978.

[Lydiard, 1996] Terri Lydiard. Using IDEF3 to capture the air campaign planning process. AIAI, University of Edinburgh, Scotland, March 1996.

[Maes and Nardi, 1988] Pattie Maes and Daniele Nardi, editors. *Meta-Level Architectures and Reflection.* North-Holland, Amsterdam, The Netherlands, 1988.

[Maes, 1986] Pattie Maes. Introspection in knowledge representation. In *Proc. 7th ECAI, Vol. I*, pages 256–269, Brighton, UK, July 1986.

[Malone *et al.*, 1997] Thomas W. Malone, Kevin Crowston, Jintae Lee, Brian Pentland, Chrysanthos Dellarocas, George Wyner, John Quimby, Charley Osborn, and Abraham Bernstein. Tools for inventing organizations: Toward a handbook of organizational processes. MIT, Cambridge, MA, 1997.

[Mayer *et al.*, 1992] R. J. Mayer, T. P. Cullinane, P. S. deWitte, W. B. Knappenberger, B. Perakath, and M. S. Wells. Information integration for concurrent engineering (IICE) IDEF3 process description capture method report. Report AL-TR-1992-0057, Armstrong Laboratory, Logistics Research Division, 1992.

[McAllester and Rosenblitt, 1991] David McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proc. 9th AAAI*, pages 634–639, Anaheim, CA, August 1991. AAAI Press/The MIT Press.

[McCarthy and Hayes, 1969] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In Bernhard Meltzer and Donald Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Edinburgh, Scotland, 1969. Also in: [Allen *et al.*, 1990, pages 393–435].

[McCarthy, 1980a] John McCarthy. Applications of circumscription to formalizing commonsense knowledge. *Artificial Intelligence*, 28:89–116, 1980.

[McCarthy, 1980b] John McCarthy. Circumscription—a form of nonmonotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[McDermott and Doyle, 1980] Drew McDermott and Jon Doyle. Non-monotonic logic i. *Artificial Intelligence*, 13:41–72, 1980. Also in: [Ginsberg, 1987, pages 111–126].

[Minton and Carbonell, 1987] Steven Minton and Jaime G. Carbonell. Strategies for learning search control rules: An explanation-based approach. In *Proc. 10th IJCAI*, pages 228–235, Milan, Italy, August 1987. Morgan Kaufmann.

[Minton *et al.*, 1985] Steven Minton, Philip J. Hayes, and Jill Fain. Controlling search in flexible parsing. In *Proc. 9th IJCAI*, pages 785–787, Los Angeles, CA, August 1985. Morgan Kaufmann.

[Minton *et al.*, 1987] Steven Minton, Jaime G. Carbonell, Oren Etzioni, Craig A. Knoblock, and Daniel R. Kuokka. Acquiring effective search control rules: Explanation-based learning in the PRODIGY system. In Pat Langley, editor, *Proc. 4th International Workshop on Machine Learning*, pages 122–133, Irvine, CA, June 1987. University of California, Morgan Kaufmann.

[Minton *et al.*, 1989] Steven Minton, Craig A. Knoblock, Daniel R. Kuokka, Yolanda Gil, Robert L. Joseph, and Jaime G. Carbonell. PRODIGY 2.0: The manual and tutorial. Technical Report CMU-CS-89-146, Carnegie Mellon University, Pittsburgh, PA, May 1989.

[Moore, 1985] Robert C. Moore. A formal theory of knowledge and action. In Jerry R. Hobbs and Robert C. Moore, editors, *Formal Theories of the Commonsense World*, chapter 9, pages 319–358. Ablex, Norwood, NJ, 1985. Also in: [Allen *et al.*, 1990, pages 480–519].

[Morgenstern, 1987] Leora Morgenstern. Knowledge preconditions for actions and plans. In *Proc. 10th IJCAI*, pages 867–874, Milan, Italy, August 1987. Morgan Kaufmann.

[Murray and Porter, 1989] Kenneth S. Murray and Bruce W. Porter. Controlling search for the consequences of new information during knowledge integration. In Alberto Maria Segre, editor, *Proc. 6th International Workshop on Machine Learning*, pages 290–295, Ithaca, NY, June 1989. Cornell University, Morgan Kaufmann.

[Musen, 1989] Mark A. Musen. Automated support for building and extending expert models. *Machine Learning*, 4:349–377, 1989.

[Neches *et al.*, 1991] Robert Neches, Richard Fikes, Tim Finin, Thomas Gruber, Ramesh Patil, Ted Senator, and William R. Swartout. Enabling technologies for knowledge sharing. *AI Magazine*, 12(3):36–56, Fall 1991.

[Newell and Simon, 1963] Allen Newell and Herbert A. Simon. GPS, a program that simulates human thought. In E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 279–293. R. Oldenbourgh KG, 1963. Also in: [Allen *et al.*, 1990, pages 59–66].

[Newell and Simon, 1976] Allen Newell and Herbert Simon. Computer science as empirical enquiry. *Communications of the ACM*, 19:113–126, 1976.

[Newell, 1982] Allen Newell. The knowledge level. *Artificial Intelligence*, 18(1):87–127, January 1982.

[Nilsson, 1980] Nils J. Nilsson. *Principles of Artificial Intelligence*. Tioga, Palo Alto, CA, 1980.

[Nodine and Unruh, 1997] Marian Nodine and Amy Unruh. Facilitating open communication in agent systems: The InfoSleuth infrastructure. In N. Singh, A. Rao, and m. Wooldridge, editors, *Proc. 4th International Workshop on Agent Theories, Architectures, and Languages*, pages 281–295, Providence, RI, July 1997.

[Nodine *et al.*, 1998] Marian Nodine, Brad Perry, and Amy Unruh. Experience with the InfoSleuth agent architecture. In Brian Logan and Jeremy Baxter, editors, *Proc. AAAI Workshop on Software Tools for Developing Agents*, Madison, WI, January 1998. AAAI Press.

[O-Plan TF, 1997] AIAI, University of Edinburgh, Edinburgh, Scotland. *Task Formalism Manual*, January 1997. Version 3.1.

[Orfali *et al.*, 1997] Robert Orfali, Dan Harkey, and Jeri Edwards. *Instant* CORBA. Wiley, New York, NY, March 1997.

[Pease and Carrico, 1997] R. Adam Pease and Todd M. Carrico. Core plan representation. Armstrong Lab Report AL/HR-TP-96-9631, Armstrong Laboratory, US Air Force, January 1997. Object Modeling Working Group.

[Pednault, 1989] Edwin P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In Ronald J. Brachman, Hector J. Levesque, and Raymond Reiter, editors, *Proc. 1st KR*, pages 324–332, Toronto, Canada, 1989. Morgan Kaufmann.

[Penberthy and Weld, 1992] J. Scott Penberthy and Daniel S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In Bernhard Nebel, Charles Rich, and William Swartout, editors, *Proc. 3rd KR*, pages 103–114, Cambridge, MA, October 1992. Morgan Kaufmann.

[Peot and Smith, 1992] Mark A. Peot and David E. Smith. Conditional nonlinear planning. In James Hendler, editor, *Proc. 1st International Conference on Artificial Intelligence Planning Sytems*, pages 189–197, College Park, MD, June 1992. Morgan Kaufmann.

[Polyak and Tate, 1998] Stephen T. Polyak and Austin Tate. Rationale in planning: Causality, dependencies, and decisions. *The Knowledge Engineering Review*, 13(3):247–262, 1998.

[Pryor and Collins, 1996] Louise Pryor and Gregg Collins. Planning for contingencies: A decision-based approach. *Journal of Artificial Intelligence Research*, 4:287–339, May 1996.

[Pryor, 1996] Louise Pryor. Opportunity recognition in complex environments. In *Proc. 13th AAAI*, pages 1147–1152, Portland, OR, August 1996. AAAI Press/The MIT Press.

[Reece *et al.*, 1994] Glen A. Reece, Austin Tate, David I. Brown, Mark Hoffman, and Rebecca E. Burnard. The PRECiS environment. University of Edinburgh, Scotland, March 1994.

[Reiter, 1980] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[Robinson, 1965] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Rosenblum *et al.*, 1993] Paul S. Rosenblum, John E. Laird, and Allen Newell, editors. *The* SOAR *Papaers: Readings on Integrated Intelligence*, volume I & II. MIT Press, Cambridge, MA, 1993.

[Russell and Norvig, 1995] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Upper Saddle River, NJ, 1995.

[Secker, 1988] Judith A. Secker. Use of O-Plan for oil platform construction project planning. AIAI-PR 22, AIAI, University of Edinburgh, Edinburgh, Scotland, June 1988.

[Seel, 1989] N. Seel. *Agent Theories and Architectures*. PhD thesis, Surrey University, Guildford, UK, 1989.

[Selman *et al.*, 1992] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proc. 10th AAAI*, pages 440–446, San Jose, CA, July 1992. AAAI Press/The MIT Press.

[Selman, 1994] Bart Selman. Near-optimal plans, tractability, and reactivity. In Jon Doyle, Erik Sandewall, and Pietro Torasso, editors, *Proc. 4th KR*, pages 521–529, Bonn, Germany, May 1994. Morgan Kaufmann.

[Shanahan, 1997] Murray Shanahan. *Solving the Frame Problem*. MIT Press, Cambridge, MA, 1997.

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

[Singh, 1993a] Narinder Singh. A Common Lisp API and facilitator for ABSI. Report Logic-93-4, Stanford University, Stanford, CA, January 1993.

[Singh, 1993b] Narinder P. Singh. Implementation details for the new ABSI facilitator. Stanford University, Stanford, CA, April 1993.

[Smith, 1977] Reid G. Smith. The CONTRACT NET: A formalism for the control of distributed problem solving. In *Proc. 5th IJCAI*, page 472, Cambridge, MA, August 1977. MIT, William Kaufmann.

[Smith, 1982] Brian C. Smith. *Reflection and Semantics in a Procadural Language*. PhD thesis, MIT, Cambridge, MA, 1982. Prologue in: [Brachman and Levesque, 1985, pages 32–39].

[SPAR, 1997] DARPA/Rome Laboratory. *Planning Initiative Shared Planning and Activity Representation*–SPAR, October 1997. Version 0.1.

[Stader, 1997] Jussi Stader. A tool set for enterprise modelling. In *Proc. 6th International Conference on Interfaces*, Montpellier, France, May 1997. EC2 & Developpement, Paris, France.

[Swartout, 1983] William R. Swartout. XPLAIN: A system for creating and explaining expert consulting programs. *Artificial Intelligence*, 21(3):285–325, September 1983.

[Tate *et al.*, 1990] Austin Tate, James Hendler, and Mark Drummond. A review of AI planning techniques. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 26–49. Morgan Kaufmann, San Mateo, CA, 1990.

[Tate *et al.*, 1994] Austin Tate, Brian Drabble, and Richard Kirby. O-Plan2: An open architecture for command, planning and control. In Monte Zweben and Mark S. Fox, editors, *Intelligent Scheduling*, chapter 7, pages 213–239. Morgan Kaufmann, San Francisco, 1994.

[Tate *et al.*, 1998] Austin Tate, Stephen T. Polyak, and Peter Jarvis. TF method: An initial framework for modelling and analysing planning domains. In *Proc. Knowledge Engineering and Acquisition for Planning: Bridging Theory and Practice*, Pittsburgh, PA, June 1998. Carnegie-Mellon University, AAAI Press.

[Tate, 1975] Austin Tate. *Using Goal Structure to Direct Search in a Problem Solver*. PhD thesis, University of Edinburgh, Edinburgh, Scotland, 1975.

[Tate, 1995] Austin Tate. Integrating constraint management into an AI planner. *Artificial Intelligence in Engineering*, 9:221–228, 1995.

[Tate, 1996a] Austin Tate. Representing plans as a set of constraints — the <I-N-OVA> model. In Brian Drabble, editor, *Proc. 3rd International Conference on Artificial Intelligence Planning Sytems*, pages 221–228, Edinburgh, Scotland, May 1996. AAAI Press.

[Tate, 1996b] Austin Tate. Towards a plan ontology. *AI*IA Notiziqe (Quarterly Publication of the Associazione Italiana per l'Intelligenza Artificiale)*, 9(1):19–26, March 1996.

[Tate, 1998] Austin Tate. Roots of SPAR—shared planning and activity representation. *The Knowledge Engineering Review*, 13(1):121–128, March 1998.

[Uschold *et al.*, 1996] Mike Uschold, Martin King, Stuart Moralee, and Yannis Zorgios. The enterprise ontology. Technical Report AIAI-TR-195, AIAI, University of Edinburgh, Edinburgh, Scotland, August 1996.

[Uschold *et al.*, 1998] Mike Uschold, Martin King, Stuart Moralee, and Yannis Zorgios. The Enterprise ontology. *The Knowledge Engineering Review*, 13(1):31–90, March 1998.

[Valente, 1994] André Valente. Planning. In Joost Breuker and Walter Van de Velde, editors, *CommonKADS Library for Expertise Modelling*, chapter 10, pages 213–229. IOS Press, Amsterdam, 1994.

[Valente, 1995] André Valente. Knowledge-level analysis of planning systems. *SIGART Bulletin*, 6(1):33–41, January 1995.

[van Harmelen and Balder, 1992] Frank van Harmelen and J. R. Balder. $(ML)^2$: A formal language for KADS models of expertise. *Knowledge Acquisition*, 4(1), March 1992.

[van Harmelen and ten Teije, 1998] Frank van Harmelen and Annette ten Teije. Characterising problem-solving methods by gradual requirements: Overcoming the yes/no distinction. In *Proc. 8th Knowledge Engineering: Methods and Languages*, Karlsruhe, Germany, January 1998. University of Karlsruhe.

[VanLehn and Jones, 1991] Kurt VanLehn and Randolph M. Jones. Learning physics via explanation-based learning of correctness and analogical search control. In Lawrence A. Birnbaum and Gregg C. Collins, editors, *Proc. 8th International Workshop on Machine Learning*, pages 110–114, Evanston, IL, June 1991. Northwestern University, Morgan Kaufmann.

[Veloso *et al.*, 1995] Manuela Veloso, Jaime Carbonell, Alicia Perez, Daniel Borrajo, Eugene Fink, and Jim Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Theoretical and Experimental Artificial Intelligence*, 7(1), 1995.

[Voß *et al.*, 1990] Angi Voß, Werner Karbach, Uwe Drouven, and Darius Lorek. Competence assessment in configuration tasks. In *Proc. 9th ECAI*, pages 676–681, Stokholm, Sweden, August 1990. Pitman.

[Warren, 1976] David H. D. Warren. Generating conditional plans and programs. In *Proc. AISB*, pages 344–354, Edinburgh, Scotland, July 1976. University of Edinburgh.

[Wavish, 1992] P. Wavish. Exploiting emergent behaviour in multi-agent systems. In E. Werner and Y. Demazeau, editors, *Proc. 3rd European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds*, pages 297–310. Elsevier Science Publishers, 1992.

[Wefald and Russell, 1989] Eric H. Wefald and Stuart J. Russell. Adaptive learning of decision-theoretic search control knowledge. In Alberto Maria Segre, editor, *Proc. 6th International Workshop on Machine Learning*, pages 408–411, Ithaca, NY, June 1989. Cornell University, Morgan Kaufmann.

[Weld and de Kleer, 1990] Daniel S. Weld and Johan de Kleer, editors. *Readings in Qualitative Reasoning about Physical Systems.* Morgan Kaufmann, San Mateo, CA, 1990.

[Weld, 1996] Daniel S. Weld. Planning-based control of software agents. In Brian Drabble, editor, *Proc. 3rd International Conference on Artificial Intelligence Planning Sytems*, pages 268–274, Edinburgh, Scotland, May 1996. AAAI Press.

[Wickler and Pryor, 1996] Gerhard Wickler and Louise Pryor. On competence and meta-knowledge. In Milind Tambe and Piotr Gmytrasiewicz, editors, *Proc. AAAI Workshop on Agent Modeling*, pages 98–104, Portland, OR, August 1996. AAAI Press, Menlo Park, CA.

[Wielinga and Breuker, 1986] Bob J. Wielinga and Joost A. Breuker. Models of expertise. In *Proc. 7th ECAI, Vol. I*, pages 306–318, Brighton, UK, July 1986.

[Wielinga *et al.*, 1992] B. J. Wielinga, A. Th. Schreiber, and J. A. Breuker. KADS: A modelling approach to knowledge engineering. *Knowledge Acquisition*, 4(1):5–53, March 1992.

[Wielinga (ed) *et al.*, 1994] Bob Wielinga (ed), Hans Akkermans, Heshem Hassan, Olle Olsson, Klas Orsvärn, Guus Schreiber, Peter Terpstra, Walter Van de Velde, and Steve Wells. Expertise model definition document. Report KADS-II/M2/UvA/026/5.0, University of Amsterdam, Amsterdam, The Netherlands, June 1994.

[Wilensky, 1981] Robert Wilensky. Meta-planning: Representing and using knowledge about planning in problem solving and natural language understanding. *Cognitive Science*, 5(3):197–233, July 1981.

[Wilkins, 1982] David E. Wilkins. Using knowledge to control tree searching. *Artificial Intelligence*, 18(1):1–51, January 1982.

[Wilkins, 1988] David E. Wilkins. *Practical Planning*. Representation and Reasoning Series. Morgan Kaufmann, San Mateo, CA, 1988.

[Winston, 1992] Patrick Henry Winston. *Artificial Intelligence*. Addison-Wesley, Reading, MA, 3rd edition, 1992.

[Witten *et al.*, 1994] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Van Nostrand Reinhold, New York, NY, April 1994.

[Wooldridge and Jennings, 1995] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theories and practice. *The Knowledge Engineering Review*, 10(2):115–152, June 1995.

[Wooldridge, 1994] Michael Wooldridge. Coherent social action. In A. G. Cohn, editor, *Proc. 11th ECAI*, pages 279–283, Amsterdam, The Netherlands, August 1994. Wiley.

[Zaniolo, 1991] C. Zaniolo. The logical data language (LDL): An integrated approach to logic and databases. Technical Report STP-LD-328-91, MCC, Austin, TX, 1991.