Programming Rational Agents in GOAL

Draft

May 19, 2011

Contents

Preface

The GOAL agent programming language is a programming language for programming multi-agent systems. It offers a rich set of language elements and features for writing agent programs.

The GOAL platform is distributed with a diverse set of environments for which agents can be programmed. These environments include among others the classic *Blocks World* environment, a dynamic variant of the *Blocks World* where users can interfere called the *Tower World* environment, a *Wumpus World* environment, an elevator simulator, and, more challenging environments such as the UNREAL TOURNAMENT 2004 environment.

Several of these environments are also particularly useful for teaching courses on agent programing. We have used in particular the *Wumpus World*, the elevator simulator, and the UNREAL TOURNAMENT 2004 environment in teaching. This last environment has been used in a large student project with over 60 students. Students programmed a GOAL multi-agent system to control a team of bots in a capture the flag scenario that were run against each other in a final competition.

GOAL is also applied to control robots. It is currently, for example, being used for controlling the academic version of the Nao robots from Aldebaran.

We are continuously developing and improving GOAL. Therefore, we suggest the reader to regularly check the GOAL website: `http://mmi.tudelft.nl/~koen/goal`. In order to be able to improve GOAL and its development environment, we very much appreciate your feedback on this manual and your experience with working with GOAL. Please mail your comments to us via `goal@mmi.tudelft.nl`.

Koen V. Hindriks, Utrecht, March, 2011

Acknowledgements
Getting to where we are now would not have been possible without many others who contributed to the GOAL project. I would like to thank everyone who has contributed to the development of GOAL, by helping to implement the language, developing the theoretical foundations, or contributing to extensions of GOAL. The list of people who have been involved one way or the other in the GOAL story so far, all of which I would like to thank are: Lăcrămioaria Aştefănoaei, Frank de Boer, Mehdi Dastani, Wiebe van der Hoek, Catholijn Jonker, Rien Korstanje, Nick Kraayenbrink, John-Jules Ch. Meyer, Peter Novak, M. Birna van Riemsdijk, Tijmen Roberti, Nick Tinnemeier, Wietske Visser, Wouter de Vries. Special thanks go to Paul Harrenstein, who suggested the acronym GOAL to me, and Wouter Pasman, who did most of the programming of the GOAL interpreter.

Chapter 1

Rational Agents

The goal of this book is to introduce a *toolbox* that is useful for developing *rational agents*. The toolbox that we will introduce here is based on the conviction that it is useful to take an *engineering stance based on common sense* to develop such agents. The toolbox consists of an *agent programming language* that incorporates key concepts derived from common sense, such as beliefs and goals, *programming patterns or idioms* that facilitate the software engineering task of designing a program, and of some of the *techniques* developed within Artificial Intelligence, such as knowledge representation, learning and planning.

One of the goals of Artificial Intelligence, in its early days, has been to create a machine that like humans is able to use *common sense* to decide on an action to perform next. Achieving this goal has been much more difficult than initially conceived[1] and instead of focusing on this grand vision, research in Artificial Intelligence has been mostly concentrated on various subareas that are related to and seem necessary to achieve this goal, including topics such as knowledge representation, planning, learning, vision, and, for example, natural language processing. Artificial Intelligence has become more of an *engineering* discipline aimed at developing "smart" applications than a discipline aimed at implementing "machine intelligence". Recently, however, there has been renewed interest in developing a machine with "artificial general intelligence" [60, 26]. Although most Artificial Intelligence researchers today thus would not consider their work as contributing directly to the early grand vision, it is interesting to trace some of the early thoughts on creating such machines. For now, we refer the reader to [50], chapter 1.

One of the key challenges for Artificial Intelligence is to actually develop rational agents. An approach is needed that facilitates the construction of such computational entities in a principled way. Ideally, such an approach is based on a theory of what rational agents are. We believe such an approach can be provided by providing a programming framework for rational agents, and we explore some of the ideas that inspire such an approach below. As has been well put in [39], our "concern, however, is not with providing [a cognitively plausible model], but in constructing simple yet adequate models which will allow effective control of behaviour."

1.1 What is an Agent?

If we want to use the notion of an agent to develop software, it is important to get a clearer picture of what an agent is exactly. Generally speaking, *an agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators* [50]. This definition of an agent is not particular to Artificial Intelligence, but it does identify some of the concerns that need to be dealt with by an agent designer. In order to develop an agent that will be successful in achieving its design objectives, it will be important to identify the *percepts* that the agent may receive and inform it about the *current state* of its environment

[1] Consider, for example, the early ambitions which clearly speak from such facts as the name General Problem Solver that Newell and Simon used for their system discussed in [43].

and to identify which *actions* an agent may perform to *change* its environment and *how* these actions affect the environment.[2] This concept of agent suggests an agent should have some basic abilities to process percepts and select actions, but does not yet address the fundamental problem of Artificial Intelligence: How to select *the right thing to do*? The agent itself remains a blackbox and therefore is too generic to base a theory of agent programming on.

Various other characteristics have been identified to differentiate software agents from other types of software programs [63]. Agents are *situated* in an environment, are *reactive* in that they are able to perceive their environment *and* respond in a timely fashion to perceived changes in their environment, are *proactive* in that they are goal-directed and *take the initiative* by performing actions that achieve their goals, and, finally, agents are *social* in that they communicate with other agents and are part of a multi-agent system. The notion of agency that is defined by these characteristics has also been labeled the *weak* notion of agency. Any software program that is situated, reactive, proactive and social thus would be an agent in this sense. Although we think these notions are useful to differentiate agents from non-agents, the notions themselves are too broad to be of direct use in engineering an agent. Additional explanation is required to clarify what it means to respond in a timely fashion, and to take the initiative, and how to implement these characteristics in a software agent.

One notion that has consistently been used by many researchers to differentiate agents from other software entities is the notion of *autonomy*. [63] say that agents that *operate without the intervention of others*, and *have control over their actions and internal state* are autonomous. This definition of what it means to be autonomous seems inspired by a perceived difference between the agent-oriented and object-oriented programming paradigm: whereas objects do not have control over their states nor actions as methods provided by that object may be called by any other object that has access to it, this is not the case with agents. Agents supposedly decide on their actions autonomously. [63] propose autonomy as a defining feature of agents but does not clarify the role this notion can have in the design of agents. Various attempts to do so have been based on the idea that autonomy is a gradual notion and that different levels of autonomy may be identified. The idea is that an agent may be autonomous with respect to some decisions but should ask e.g. a user for assistance if it lacks the authority to decide on its own. Exploring the related issues has given rise to research on what is called *adjustable* autonomy.

Other notions of agency have been based on exploiting the common sense notions that Artificial Intelligence in its early days tried to formalize and implement in a machine. The idea has been to implement agents by explicitly modeling their beliefs, desires, goals, intentions, plans, etc. A key source of inspiration has been the work of Dennett [21] on what he called the "intentional stance". The idea of agents as *intentional systems* has been picked up by various researchers, some of the more prominent examples are [54, 55] and [63].

Interestingly, definitions of agency have also been discussed in the context of dynamical systems theory, which typically is more oriented towards physical and biological systems. Although we are most interested here in *software* agents there are clear relations between this type of work and robotics in Artificial Intelligence. An agent is defined in [2] as a system that is a distinct *individual* that stands out in its environment and has its own identity, *interacts with its environment* and initiates activity, and has *goals or norms* that are pursued by the system. Defining an agent as an individual means to set it apart from its environment and other agents. A key aspect of agency in [2] is the asymmetry between an agent and its environment. [2] assume that an agent always is coupled with an environment and an agent interacts with its environment. The interaction between an agent and its environment is asymmetrical, however, as an agent initiates actions to change its environment but not vice versa. An agent may be characterized as the "center of influence" in the terminology of [2]. In our context, however, interactional asymmetry is described more fruitfully using intentional terminology than the physical terminology of [2]. Interactional asymmetry then means that an agent initiates and *chooses to perform* actions intentionally in its environment. The environment itself only responds to the agent's initiatives but does not itself

[2]Note that an environment does not need to be a *physical* environment. An agent may be a physical *robot* acting in a physical environment but may also be an entity that is part of a *software* environment such as a softbot that searches the World Wide Web.

make such intentional choices.[3]

1.2 Applying the Intentional Stance

In Artificial Intelligence, various common sense concepts are used to explain some of the core research areas. For example, planning research develops techniques to construct *plans* for an agent to achieve a *goal*, and work on knowledge representation develops languages to represent and reason with the *knowledge* of an agent. Although the intuitions that guide this research are derived from the everyday, common sense meanings of these concepts, the techniques and languages developed within Artificial Intelligence have produced *formal* counterparts of these concepts that are not as rich in meaning and deviate in other ways. Even so, in practice our everyday intuitions have proven useful for developing specific applications when using these formal notions (cf. [54]).

The key notions to explain what a rational agent are involve precisely these same concepts of knowledge, belief and goals, and the idea naturally arises that it would be most useful to *use* these concepts not only to *describe* but also to *engineer* such agents. Engineering agents in terms of the beliefs and goals they have, and the choice of action they derive from these beliefs and goals may be advantageous for several reasons. As [41] states:

> It is useful when the ascription [of intentional notions] helps us understand the structure of the machine, its past or future behavior, or how to repair or improve it. (p. 1)
>
> The belief and goal structure is likely to be close to the structure the designer of the program had in mind, and it may be easier to debug the program in terms of this structure [...]. In fact, it is often possible for someone to correct a fault by reasoning in general terms about the information in a program or machine, diagnosing what is wrong as a false belief, and looking at the details of the program or machine only sufficiently to determine how the false belief is represented and what mechanism caused it to arise. (p. 5)

The idea to use common sense notions to build programs can be traced back to the beginnings of Artificial Intelligence. Shoham, who was one of the first to propose a new programming paradigm that he called *agent-oriented programming*, cites McCarthy about the usefulness of ascribing such notions to machines [41, 55]. One of the first papers on Artificial Intelligence, also written by McCarthy, is called *Programs with Common Sense* [40]. It has been realized that in order to have machines compute with such notions it is imperative to precisely specify their meaning [55]. To this end, various logical accounts have been proposed, mainly using modal logic, to clarify the core common sense meaning of these notions [17, 37, 48]. These accounts have aimed to precisely capture the essence of a conceptual scheme based on common sense that may also be useful and applicable in specifying rational agent programs. The first challenge thus is to provide a well-defined semantics for the notions of belief, goal and action which can also provide a computational interpretation of these notions useful for programming agents.

One of the differences between the approach promoted here and earlier attempts to put common sense concepts to good use in Artificial Intelligence is that we take an *engineering stance* (contrast [40] and [55]). The concepts are used to introduce a new agent programming language that provides useful programming constructs to develop agent programs. The second challenge is to provide agent programming language that is practical, transparent, and useful. It must be practical in the sense of being easy to use, transparent in the sense of being easy to understand, and useful in the sense of providing a language that can solve real problems.

What is the *intentional stance*? According to Dennett, it is a strategy that "consists of treating the object whose behavior you want to predict as a rational agent with beliefs and desires (...)"

[3] [2] defines interactional asymmetry in terms of structural coupling: "An agent is a system that systematically and repeatedly modulates its structural coupling with the environment." From a software engineering point of view, this characterization may be too strong however as most agent systems do not dynamically change their coupling with an environment (if coupled at all).

[21].[4]

A more detailed description of the strategy is offered by Dennett [21]:

> first you decide to treat the object whose behavior is to be predicted as a rational agent; then you figure out what beliefs that agent ought to have, given its place in the world and its purpose. Then you figure out what desires it ought to have, on the same considerations, and finally you predict that this rational agent will act to further its goals in the light of its beliefs. A little practical reasoning from the chosen set of beliefs and desires will in many - but not all - instances yield a decision about what the agent ought to do.

The intentional stance is here promoted as an *engineering stance* or *design stance*. We will label certain parts of programs that implement rational agents *knowledge*, *beliefs*, and *goals*. Of course, at certain times you may find yourself wondering and asking yourself: Are these things we have called beliefs not just statements stored in a database? Are these things that we call goals not just simple statements in a second database? And, to be as explicit as possible, of course, you would be right. *We* have *labelled* these databases belief and goal base and the statements that are stored in these databases beliefs and goals. There is no sense in which we would want to defend a position that these statements are *really* beliefs or goals. What we advocate here is a particular stance, a way of looking at what's inside an agent, by means of concepts such as belief and goal. In a sense, by taking such a pragmatic stance, one cannot argue then anymore; who would say we are not allowed to use this mentalistic language if it suits our purposes for describing agents. Only those apparently offended by it. In reply, we may point out however that we have only promoted an *engineering stance*.

Does this leave no room for discussion then? There is, as is apparent from the different approaches that have been proposed and defended within the field of autonomous agents. The point is that we still have to at least clarify that our use of mentalistic language matches to some extent with common sense and our basic understanding of these concepts. It simply does not make sense to label just about anything a goal. Here, we return to the points made above: The use of mentalistic terminology, from our perspective, should be *useful*, and we believe it will only be so if there is a sufficient "overlap" in our common use of concepts such as beliefs and goals with the way these concepts are used within agent-oriented programming (and, for that matter, agent technology more broadly). Achieving this goal is not a simple task, and requires a careful design and explanation of how we can apply our common sense intuitions to engineer rational agents.

1.3 What is Common Sense?

If we want to use common sense to design and develop agents, we first need to make more explicit what we mean by common sense. In a sense, this is clear to all of us, as it is an understanding that is *common* to us all. It is useful to make this understanding explicit, however, in order to be able to discuss which elements are useful as part of a toolbox to develop agents and which elements are less useful. It turns out that this is harder than expected and to a certain extend an analogy may be drawn here with explicating the underlying grammar of natural language: Although we all know how to use natural language, it is not easy at all to explicate the underlying rules of grammar that determine what is and what is not a meaningful sentence. We somehow learn and apply the rules of language *implicitly*, and a similar story applies to the use of common sense notions such as beliefs, goals, etc. Children of early age, for example, are not able to apply these concepts adequately, which explains why it is sometimes hard to make sense of their explanations.

The concepts of common sense that we are interested in are often labelled *folk psychology*.[5] We explain what must have gone on in the mind of an agent by means of ascribing it beliefs and

[4] Dennett continues to make a claim as to what it is to be a "true believer": it is "to be an *intentional system*, a system whose behavior is reliably and voluminously predictable via the intentional strategy".

[5] It is tempting to use the label *rational psychology* instead, as [24] does, to emphasize the *rationality* assumed within our common sense view of human psychology. However, this label may also easily give rise to confusion as it may be used to refer to a specific kind of scientific psychology. For example, [22] uses *rational psychology* to refer

goals.[6] Folk psychology assumes that agents are *rational* in the sense that agents perform actions to further their goals, taking into account their beliefs about the current state of the world. It is for this reason that we have used the label *rational agent* to refer to the computational entities we want to engineer, and we turn now to exploring this notion in more detail.

1.3.1 What is Rationality?

The use of intentional notions such as beliefs and goals presupposes *rationality* in the agent that is so described [21].

The beliefs of an agent, if they are to count as rational, should satisfy a number of conditions.[7] The beliefs that a *rational* agent maintains should be *true*. That is, the beliefs of an agent should reflect the actual state of the environment or *correspond* with the way the world actually is. Using a bit of formal notation, using p to denote a proposition that may be either true or false in a state and $\mathbf{bel}(p)$ to denote that an agent believes p, ideally it would be the case that whenever $\mathbf{bel}(p)$ then also p.[8] As has been argued by several philosophers, most of our common sense beliefs must be true (see e.g. [21, 24]), which is not to say that agents may not maintain exotic beliefs that have little justification such as the belief that pyramids cannot have been built without the help of extraterrestrial life. The point is rather that most of the beliefs that are common to us all must be true in order for our explanations of behavior to make sense at all.

There is also sort of a *rational pressure* on an agent to maintain beliefs that are as complete as possible. The beliefs of an agent may be said to be *complete* if an agent has a "complete picture" of the state it is in. In general, such a requirement is much too strong, however. We do not expect an agent to have beliefs with respect to every aspect of its environment, including beliefs about, for example, a list of all items that are in a fridge. The sense of completeness that is meant here refers to all *relevant* beliefs that an agent *reasonably* can have. This is somewhat vague and depends heavily on context, but an agent that acts without taking relevant aspects of its environment to which it has perceptual access into account may be said to be irrational. For example, intuitively, it would be strange to see an agent that has agreed to meet a person at a certain location go to this location while seeing the person he is supposed to meet go somewhere else.

Motivational attitudes such as desires and goals are also subject to rational pressures.[9] However, it also seems obvious that desires or goals are not subject to the same rational pressures as beliefs. Desires nor goals need to be true or complete in the sense that beliefs should be. Agents are to a large extent free (autonomous?) to adopt whatever desires or goals they see fit. Of course, there may be pressures to adopt certain goals but for completely different reasons. These reasons include, among others, previous agreements with other agents, the need to comply with certain norms, and the social organization that an agent is part of. The freedom that an agent is granted otherwise with respect to adopting desires seems almost limitless. An agent may desire to be rich while simultaneously desiring to live a solitary live as Robinson Crusoe (which would require little or no money at all). Although desires may be quite out of touch with reality or even inconsistent, in contrast the goals that an agent adopts must conform with certain *feasibility conditions*. These

to "the study of all possible minds.", although, surprisingly, the notion of rationality does not play a big role in the view exposed.

[6] Agents may have all kinds of mental attitudes, such as expecting, fearing, etc. However, in the remainder we will *mainly* focus on the notions of belief and goal, which seem to be at the core of our folk psychology and moreover will turn out to be sufficient for developing a theory of rational agents.

[7] Rationality associated with beliefs has also been called *theoretical rationality* [47].

[8] It is usual in the logic of knowledge, or *epistemic logic*, to say in this case that an agent *knows* that p. Using $\mathbf{know}(p)$ to denote that an agent knows p, knowledge thus is defined by: $\mathbf{know}(p)$ iff $\mathbf{bel}(p)$ and p (cf. [23, 42]). Such a definition, most would agree, only approximates our common sense notion of knowledge. It lacks, for example, any notion that an agent should be able to provide *adequate justification* for the beliefs it maintains.The latter has been recognized as an important defining criteria for knowledge since Plato, who defined knowledge as *true justified belief*. Although this definition for all practical purposes would probably be good enough, there are still some unresolved problems with this definition.

[9] Rationality associated with motivational attitudes is called *practical rationality*, and, encompasses all aspects of rationality other than theoretical rationality according to Pollock [47], including rationality of emotions, for example.

conditions include that goals should be *consistent* in the sense that one goal of agent must not exclude the accomplishment of another goal. For example, it is not rational for an agent to have a goal to go to the movies and to finish reading a book tonight if accomplishing both would require more time than is available. A second condition that goals need to satisfy is that *the means should be available to achieve the goals* of the agent. An agent would not be rational when it adopts a goal to go to the moon without any capabilities to do so, for example.

As argued in [45], *goals* (and intentions) should be related in an appropriate way to "aspects of the world that the agent has (at least potentially) some control over." For example, an agent may wish or desire for a sunny day, but an agent cannot rationally adopt a goal without the capabilities to control the weather.

Note that this requirement may, from the perspective of the agent, be relative to the beliefs that the agent holds. If an agent believes he is able to control the weather as he pleases, that agent may be said to rationally adopt a goal to change the weather as he sees fit. It is kind of hard to judge such cases, though, since the agent may be accused of holding irrational beliefs that do not have any ground in observable facts (statistically, for example, there most likely will not be any correlation between efforts undertaken by the agent and the weather situation).

1.3.2 First-Order and Higher-Order Intentional Systems

Agents as intentional systems have beliefs and goals to represent their environment and what they want the environment to be like. An intentional system that only has beliefs and goals about its environment but no beliefs and goals *about* beliefs and goals is called a *first-order* intentional system [21]. As observers, looking at such agents from an *external* perspective, we can represent the mental content of an agent by sentences that have the logical form:

$$\mathbf{bel}(p) \quad : \quad \text{the agent } \textit{believes} \text{ that } p \tag{1.1}$$

$$\mathbf{goal}(p) \quad : \quad \text{the agent } \textit{has a goal} \text{ (or } \textit{wants}\text{) that } p \tag{1.2}$$

where p is instantiated with a statement about the environment of the agent. For example, an agent may believe that there are ice cubes in the fridge and may want some of these ice cubes to be in his drink. Note that an agent that would describe his own mental state would use sentences of a similar logical form to do so.

A *second-order* intentional system can have beliefs and goals *about* beliefs and goals of itself and other agents. Here we have to slightly extend our formal notation to include *agent names* that refer to the agent that has the beliefs and goals. Sentences to describe the second-order beliefs and goals of an agent a about the beliefs and goals of an agent b have the logical form:

$$\mathbf{bel}(a, \mathbf{bel}(b, p)) \quad : \quad a \textit{ believes} \text{ that } b \textit{ believes} \text{ that } p \tag{1.3}$$

$$\mathbf{bel}(a, \mathbf{goal}(b, p)) \quad : \quad a \textit{ believes} \text{ that } b \textit{ wants} \text{ that } p \tag{1.4}$$

$$\mathbf{goal}(a, \mathbf{bel}(b, p)) \quad : \quad a \textit{ wants} \text{ that } b \textit{ believes} \text{ that } p \tag{1.5}$$

$$\mathbf{goal}(a, \mathbf{goal}(b, p)) \quad : \quad a \textit{ wants} \text{ that } b \textit{ wants} \text{ that } p \tag{1.6}$$

The first two sentences attribute a belief to agent a about another agent b's beliefs and goals. The third and fourth sentence express that agent a wants agent b to believe respectively want that p. The agent names a and b may be identical, and we may have $a = b$. In that case, the sentence $\mathbf{bel}(a, \mathbf{bel}(a, p))$ attributes to a the belief that the agent itself believes p. Such agents may be said to *introspect* their own mental state, as they have beliefs about their own mental state [41]. Similarly, the sentence $\mathbf{goal}(a, \mathbf{bel}(a, p))$ means that agent a wants to believe (know?) that p. In the latter case, interestingly, there seems to be a rational pressure to want to *know* something instead of just wanting to believe something. I may want to believe that I win the lottery but I would be rational to act on such a belief only if I know it is true that I will win the lottery. An agent that simply wants to believe something may engage in *wishful thinking*. In order to avoid that an agent fools itself it should want to know what it believes and in order to ensure this a check is needed that what is believed also is the case.

It is clear that one can go on and similarly introduce third-order, fourth-order, etc. intentional states. For example, agent *a* may *want* that agent *b* *believes* that agent *a* *wants* agent *b* to assist him in moving his furniture. This sentence expresses a third-order attitude. Such attitudes seem relevant, for instance, for establishing cooperation. It also has been argued that third-order attitudes are required to be able to understand *communication* between human agents [27, 28].

In the current draft of this book, GOAL agents are *first-order* intentional system. A first-order intentional system has beliefs and goals but no beliefs and goals *about* beliefs and goals. Using the formal notation for *believe* introduced above, this means that **bel**(p) may be true of a GOAL agent but the operator **bel** cannot be *nested*. For example, **bel**($\neg$**bel**(p)) where $\neg$ denotes negation is never true of a GOAL agent, and such an agent thus is not aware of its own lack of knowledge (in other words, it does not have any *introspective* capabilities). This also means that a GOAL agent cannot have beliefs and goals about the beliefs and goals of *other* agents.

It has been argued by philosophers that the ability to maintain higher-order beliefs and goals is a mark of *intelligence* [21],and a prerequisite for being *autonomous*. The intuition is that an agent that does not want to want to clean the house cannot be said to be free in its choice of action, or autonomous. Interestingly, developmental psychologists have contributed a great deal to demonstrating the importance of second-order intentionality, or the ability to *metarepresent* [24]. For example, in a well-known experiment called the *false belief task* children of four have been shown to be able to represent false beliefs whereas children of age three are not.[10] Finally, it has also been argued that without addressing the problem of common sense, including in particular the human ability to metarepresent, or "to see one another as minds rather than as bodies", Artificial Intelligence will not be able to come to grips with the notion of intelligence [61]. As mentioned already above, one argument is that intelligence requires the ability to communicate and cooperate with other agents, which also seems to require higher-order intentionality.[11]

Thus, in a sense, the rational agents discussed here may be compared with three-year olds that lack some of the mental representation abilities of four-year olds. Some work on extending GOAL agents with the ability to represent the mental states of other agents has been reported in [35]. One reason for limiting the orders of nesting is related to the computational complexity that results from allowing arbitrary nesting [23]. Another reason for being cautious about introducing such nestings concerns the fact that human adults are reported to also have great difficulty with higher-order intentionality, a fact that is, for example, exploited often by comedians and in sitcoms.

1.4 Notes

The idea of using the intentional stance to develop rational agents has been proposed in various papers by Shoham. Shoham's view is that agents are "formal versions of human agents", see for example, [54].[12] He argues that artificial agents should have "formal versions of knowledge and belief, abilities and choices, and possibly a few other mentalistic-sounding qualities".

Central in the early work on agent-oriented programming in [54] has been *speech act theory*. According to [54], "AOP too can be viewed as a rigorous implementation of a fragment of direct-speech-act theory". This theory has initially also played a large role in the design of a generic communication language for rational agents, see Chapter 6 for a more detailed discussion.

[10] Children of both ages were asked to indicate where a puppet would look for chocolate in a cabinet with several drawers after having witnessed the puppet putting the chocolate in a drawer, seeing the puppet leave the room, and the chocolate being placed in another drawer in the absence of the puppet. Children of age three consistently predict that upon return the puppet will look in the drawer where the chocolate actually is and has been put after the puppet left, whereas four-year olds predict the puppet will look in the drawer where it originally placed the chocolate.

[11] [41] writes that "[i]n order to program a computer to obtain information and co-operation from people and other machines, we will have to make it ascribe knowledge, belief, and wants to other machines and people".

[12] The title of this book chapter, *Implementing the Intentional Stance*, does not refer to what we have been after: a reorientation of the intentional stance to a design stance for developing rational agents. It should not be taken to mean that we are after an implementation of an agent that is itself able to adopt the intentional stance, which, as discussed in the main text would at least require higher-order intentionality [24].

Our view on designing and programming agents as one particular form of *implementing the intentional stance* derives from [54]. The *intentional stance* has been first discussed by Daniel Dennett in [21]. The usefulness of ascribing mental attitudes to machines, however, was already discussed by McCarthy [41].

The concept of *autonomy*, although it has been used to differentiate agents from other software entities, is one of the notions in the literature about which there is little consensus. [63] defines autonomy as the ability of an agent to control its actions and internal state. In [14] autonomy is defined in terms of the ability and permission to perform actions. The basic intuition is that the more freedom the agent has in choosing its actions, the more autonomous it is. In a Robotics context, [6] states that autonomy "refers to systems capable of operating in the real-world environment without any form of external control for extended periods of time" where "capable of operating" means that the system performs some function or task. It is interesting to note that none of these definitions explicitly refers to the need for a system to be adaptive though generally this is considered to be an important ability of autonomous systems. Minimally, adaptivity requires *sensing* of the agent's environment in which it operates. In our opinion definitions and discussions of autonomy thus should at least reference the ability to sense an environment.

1.5 Exercises

1.5.1

An agent in the weak sense is an entity that is situated, reactive, proactive and social. The text of this chapter claims that these characteristics do not provide support for an agent designer to develop an agent. Do you agree? Discuss two of these characteristics. and explain and motivate why you think they are or are not helpful in the agent design process.

1.5.2

Discuss whether the notion of *autonomy* can have a *functional* role in the design of agents. If you think autonomy does have a role to play, explain the role the notion can have in design and motivate this by providing two reasons that support your point of view. If you think autonomy does *not* have a functional role, provide two arguments why this is not the case.

1.5.3

Discuss whether a robotic industrial manipulator such as the IRB 6400 spot welding robot, Electrolux's Trilobite household robot, and the Google car are autonomous. Use the Internet to find out more about these robotic systems and identify which features makes them autonomous or not according to you.

1.5.4

Is *mobility* a property required for an agent to be autonomous? In your answer, provide separate answers and argumentation for physical agents, i.e. robots, and for software agents, i.e. softbots.

1.5.5

Discuss whether a *rational* agent may be expected to satisfy the following axioms:

1. Axiom of *positive introspection*: if $\mathbf{bel}(p)$, then $\mathbf{bel}(\mathbf{bel}(p))$.
2. Axiom of *negative introspection*: if $\neg\mathbf{bel}(p)$, then $\mathbf{bel}(\neg\mathbf{bel}(p))$.

1.5.6

It has been argued that higher-order beliefs about opponents in games are crucial for good game play, i.e. winning (see e.g. [24]).What number of nestings do you think are required for each of the following games?

1. Hide-and-seek
2. Memory
3. Go Fish ("kwartetten" in Dutch)
4. Chess

1.5.7

Do you think the ability to have second- or higher-order beliefs are a prerequisite for being able to *deceive* another agent? Motivate your answer by means of an example.

1.5.8

The cognitive psychologists Tversky and Kahneman asked participants to read the following story:

> Linda is 31 years old, single, outspoken, and very bright. She majored in philosophy. As a student, she was deeply concerned with issues of discrimination and social justice, and also participated in anti-nuclear demonstrations.

They were then asked to rank-order a number of possible categories in order of likelihood that Linda belonged to each. The following three categories were included:

- bank teller
- feminist
- feminist bank teller

Before reading on, you might want to order the categories yourself first.
Most participants ranked feminist bank teller as more probable than either bank teller or feminist. This, however, is incorrect, because all feminists bank tellers belong to the larger categories of feminists and bank tellers! This is one of the examples Tversky and Kahneman used to demonstrate that humans ask whether people are rational. If so, why, or, why not?

Chapter 2

Mental States

A rational agent maintains a *mental state* to represent the current state of its environment and the state it wants the environment to be in. The representation of the current state determines the informational state of the agent, and consists of the *knowledge* and *beliefs* of an agent.[1] The representation of the desired state determines the *motivational* state of the agent, and consists of the *goals* of an agent.[2] A mental state thus is made up of the knowledge, beliefs and goals of an agent. The mental attitudes discussed in this chapter are first-order attitudes, and do not allow for e.g. second-order beliefs about beliefs or goals. Agents equipped with such mental states are first-order intentional systems (cf. Chapter 1).

2.1 Representing Knowledge, Beliefs and Goals

One of the first steps in developing and writing a GOAL agent is to design and write the knowledge, beliefs and goals that an agent needs to meet its design objectives. Of course, the process of doing so need not be finished in one go but may need several iterations during the design of an agent. It is however important to get the representation of the agent's knowledge, beliefs and goals right as the actions chosen by the agent will depend on its knowledge, beliefs and goals. More specifically, the action specifications and action rules discussed in Chapters 3 and 4 also depend on it. To this end, we need a *knowledge representation language* that we can use to describe the content of the various mental attitudes of the agent. Although GOAL is not married to any particular knowledge representation language, here *Prolog* will be used to present an example GOAL agent.

2.1.1 Example Domain

As a running example in this chapter, we will use one of the most famous domains in Artificial Intelligence known as the *Blocks World* [62, 56, 50]. In its most simple (and most common) form, in the Blocks World an agent can move and stack cube-shaped blocks on a table by controlling a robot gripper. One important fact is that the robot gripper is limited to holding one block at a time and cannot hold a stack of blocks. For now, we will focus on the configuration of blocks on the table; we will provide the details related to the gripper later in Chapter 3. A block can be directly on top of at most one other block. That is, a block is part of a stack and either located on top of a single other block, or it is sitting directly on top of the table. These are some of the basic "laws" of the Blocks World. See Figure 2.1 for an example Blocks World configuration. This figure graphically depicts both the initial state or configuration of blocks as well as the goal state. It is up to the agent to realize the goal state by moving blocks in the initial state and its successor states. We add one assumption about the table in the Blocks World: we assume that the table is

[1] Knowing and believing are also called *informational attitudes* of an agent. Other informational attitudes are, for example, expecting and assuming.

[2] Wanting is a *motivational attitude*. Other motivational attitudes are, for example, intending and hoping.

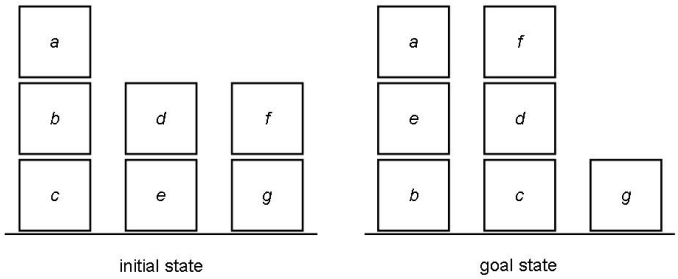

Figure 2.1: Example Blocks World problem taken from [56].

large enough to be able to place all blocks directly on the table. This is another basic "law" of our version of the Blocks World.[3]

An agent that has the task to achieve a goal state in the Blocks World needs to be able to represent the configuration of blocks. It needs to be able to represent such configurations for two reasons: to keep track of the current state of the Blocks World and to describe the goal state that it wants to reach. As we will be using a first-order language, we need to introduce specific predicates to be able to fully specify a configuration of blocks. Typically, it is a good idea to introduce predicates that correspond with the most *basic concepts* that apply in a domain. More complicated concepts then may be *defined* in terms of the more basic concepts. We will use this strategy here as well.[4]

One of the most basic concepts in the Blocks World is that a block is *on top of* another block or is *on* the table. To represent this concept, we introduce a binary predicate:

```
on(X, Y)
```

The predicate `on(X, Y)` means that *block* `X` *is (directly) on top of* `Y`. For example, `on(a,b)` is used to represent the fact that block `a` is on block `b` and `on(b,table)` is used to represent that block `b` is on the table. This is our informal definition of the predicate `on(X, Y)`. To understand what is implied by this definition, a few comments on some aspects of this definition are in order. First, only *blocks* can be on top of something else in the Blocks World. Second, `X` must be *directly* on top of `Y` meaning that there can be no other block in between block `X` and `Y`. Third, `Y` may refer *either* to a block *or* to the table. And, fourth, there can be *at most one* block on top of another block. Note that it is up to the agent designer to stick to these rules and to use the predicate `on` to express exactly this and nothing else.

There is one additional thing to note about the examples we have used. When we say that `on(b,table)` means that block `b` is on the table we have implicitly assumed that all blocks have *unique names* that we can use to refer to each block. This assumption is very useful since by introducing such names we then can refer to each block and are able to distinguish one block from another (as every block is assumed to have another name). It thus greatly simplifies our task as

[3]This assumption is dropped in so-called *slotted* Blocks World domains where there are only a finite number of slots to place blocks.

[4]It can be shown that the basic concept *above* is sufficient in a precise sense to completely describe arbitrary, possibly infinite configurations of the Blocks World; that is not to say that everything there is to say about a Blocks World configuration can be expressed using only the *above* predicate [18]. Here we follow tradition and introduce the predicates *on* and *clear* to represent Blocks World configurations.

designer to design an agent controller for the Blocks World. In addition we will assume we have names *for all* blocks in the Blocks World; and, as we have already been using it, we use the label `table` to refer to the table in the Blocks World. This in effect is a *domain closure* assumption for the Blocks World. In order to make explicit which blocks are present in the Blocks World we introduce the unary predicate `block`. To represent an instance of the Blocks World domain with exactly 7 blocks, as in Figure 2.1, we then can use:

```
block(a). block(b). block(c). block(d). block(e). block(f). block(g).
```

Of course, we need to implicitly assume here that *there are no other blocks*. We cannot explicitly state this using Prolog (we would need a quantifier to do so) but this assumption is built into Prolog's semantics (by means of the Closed World assumption, see also the box discussing this assumption below). As a side effect, note that the `block` predicate also allows us to distinguish blocks from the table.

Using the `on` predicate makes it possible to represent the state a Blocks World is in. For example, the configuration of blocks in the initial state in Figure 2.1 can be represented by:

```
on(a,b). on(b,c). on(c,table). on(d,e). on(e,table). on(f,g). on(g,table).
```

Blocks World Problem
Using the `on` predicate we can actually *define* the notion of a state in the Blocks World. A *state* is a *configuration of blocks* and may be defined as a set of facts of the form `on(X,Y)` that is consistent with the basic "laws" of the Blocks World (i.e. at most one block is directly on top of another, etc.). Assuming that the set B of blocks is given, we can differentiate *incomplete or partial* states from *completely* specified states. A state that contains a fact `on(X,Y)` for each block `X` $\in B$ in the set B is called *complete*, otherwise it is a *partial* state.
As both the initial state and the goal state are configurations of blocks, it is now also possible to formally define what a *Blocks World problem* is. A Blocks World problem simply is a pair $\langle B_{initial}, G \rangle$ where $B_{initial}$ is the initial state and G denotes the goal state. The labels $B_{initial}$ and G have been intentionally used here to indicate that the set of facts that represent the initial state correspond with the initial beliefs and the set of facts that represent the goal state correspond with the goal of an agent that has the task to solve the Blocks World problem.

The third predicate that we introduce is the unary predicate:

```
clear(X)
```

`clear(X)` means that `X` is clear. That is, if `X` is a block `clear(X)` means there is no other block on top of it. Obviously, given a complete state, the information that a block is clear is implicitly present and can be derived. Just by looking at Figure 2.1 it is clear that blocks `a`, `d`, and `f` are clear in the initial state. The predicate is introduced mainly to facilitate reasoning about the Blocks World. In order to deal with the limitations of the gripper it is important to be able to conclude that a block is clear. It is possible to *define* the clear concept in terms of the `on` predicate by the following clause:

```
clear(X) :- block(X), not( on(_, X) ).
```

Floundering
It is important to take the *floundering* problem of Prolog systems into account when defining the procedure `clear(X)`. Floundering would cause problems if we would have defined the `clear` predicate by

```
clear(X) :- not( on(Y, X) ), block(X).
```

Although this definition would still work fine when `X` is instantiated, it would give the wrong answer when `X` would not be instantiated. That is, the query `clear(a)` in the initial state of Figure 2.1 yields true, but the query `clear(X)` would yield false and return no answers! An easy rule of thumb to avoid problems with floundering is to make sure that each variable in the body of a rule first appears in a positive literal. We have achieved this by introducing the positive literal `block(X)` first which binds `X` before the variable is used within the second, negative literal `not(on(_, X))`. Moreover, in our definition we have used a *don't care* variable `_` instead of the variable `Y` above.

The `clear` predicate is introduced in part because an agent needs to be able to conclude that a block `X` can be placed on top of another block `Y` or on the table. This can only be done if there is no other block on top of block `Y`, i.e. block `Y` is clear, or there is room on the table to put the block on. In both cases we would like to say that the block or the table is clear to indicate that a block can be moved onto either one. The informal meaning of `clear(X)` therefore is made slightly more precise and `clear(X)` is interpreted as that there is a clear space on `X` to hold a block [50]. In other words, the table is also said to be clear as we have assumed that the table always has room to place a block. The definition of `clear(X)` above, however, does not cover this case (verify yourself that `clear(table)` does not follow; also check that even when the `block(X)` literal in the body of the clause defining `clear(X)` is removed it still does not follow that `clear(table)`). We therefore need to add the following fact about the table:

```
clear(table)
```

This ends our design of a language to represent the Blocks World. That is not to say that there are no logical extensions we can introduce that may be useful to represent and reason about the Blocks World. In fact, below we will introduce a definition of the concept of a *tower*. And there are still other useful concepts that may be introduced (see the Exercise Section where the binary predicate `above` is defined).

2.1.2 Mental State

We are now ready to define the mental state of our agent for controlling the robot gripper in the Blocks World. A mental state of a GOAL agent consists of *three* components: knowledge, beliefs and goals. As discussed, knowledge and beliefs represent the current state of the agent's environment and goals represent the desired state. We will first discuss the knowledge and beliefs of our Blocks World agent and thereafter discuss the goal state of the agent.

The information that an agent has about its environment consists of the *knowledge* and of the *beliefs* the agent has. The main difference between knowledge and beliefs in GOAL is that knowledge is *static* and cannot change at runtime and belief is *dynamic* and may change at runtime. An immediate consequence of this difference is that the knowledge base that contains an agent's knowledge cannot be used to keep track of the current state of the environment. Because the state of the environment will change instead this state needs to be represented by means of the agent's beliefs.

As we will explain in more detail in Chapter 3, only *facts* such as `on(a,b)` can be modified at runtime and *rules* such as `clear(X) :- block(X), not( on(_, X) )` cannot be modified. As a rule of thumb, therefore, it is a good idea to put rules and definitions into the knowledge base. Such rules typically consist of *domain knowledge* that represents the basic "laws" or the "logic"

of the domain. The knowledge base may also contain facts that do not change. An example of a fact in the Blocks World that might be added to the knowledge base because it is always true is `clear(table).`

```
knowledge{
   block(a). block(b). block(c). block(d). block(e). block(f). block(g).
   clear(table).
   clear(X) :- block(X), not(on(_, X)).
}
```

The **knowledge** keyword is used to indicate that what follows is the knowledge of the agent. This part of the agent program is also called the **knowledge** section and is used to create the *knowledge base* of the agent. Apart from the `clear(table)` fact the **knowledge** section also consists of facts of the form `block(X)` to represent which blocks are present in the Blocks World. All blocks present in Figure 2.1 are enumerated in this section. By putting these facts in the **knowledge** section the assumption is made implicitly that no blocks will be introduced later on nor will any of these blocks be removed from the environment (why?). The **knowledge** section also includes the domain knowledge about when a block is clear. The rule `clear(X) :- block(X), not( on(_, X) )` represents this domain knowledge.

One thing to note is that the `block` facts are represented as separate facts that are separated using the period symbol as is usual in Prolog. The code listed above in the **knowledge** section actually represents a Prolog program. In fact, the content of a **knowledge** section must be a Prolog program and respect the usual syntax of Prolog. It is also possible to use most of the built-in operators of the Prolog system that is used (in our case, SWI Prolog [58]). For a list of all built-in operators that can be used, consult the Appendix **??**.

Closed World Assumption
It is important to realize that the rule `clear(X) :- block(X), not(on(_, X))` can only be used to correctly infer that a block is clear if the state represented by the agent's beliefs is *complete*. The point is that the negation `not` is the *negation by failure* of Prolog. This means that `block(X), not(on(_, X))` succeeds for every block `block` about which there is *no* information whether `on(_, block)` holds. In the absence of any information that `on(_, block)` it follows that `clear(block)` since in that case no proof can be constructed for `on(_, block)` and *negation as failure* applies.
Another way to make the same point is to observe that Prolog supports the *Closed World Assumption*. Informally, making the Closed World Assumption means that anything not known to be true is assumed to be false. In our example, this means that if there is no information that there is a block on top of another, it is assumed that there is no such block. This assumption, of course, is only valid if all information about blocks that are on top of other blocks is available and represented in the belief base of the agent. In other words, the state represented in the belief base of the agent needs to be complete.
A final remark concerns how an agent can maintain a complete state representation in its belief base. An agent can only keep track of the complete state of the environment if that state is *fully observable* for the agent.
The lesson to be learned here is that domain knowledge needs to be carefully designed and basic assumptions regarding the domain may imply changes to the way domain knowledge is to be represented. In our example, if an agent cannot keep track of the complete state of the Blocks World, then the rule for `clear(X)` needs to be modified.

Facts that may change, however, must be put in the belief base of the agent. The initial state of Figure 2.1 thus may be represented in the **beliefs** section of a GOAL agent.

```
beliefs{
   on(a,b). on(b,c). on(c,table). on(d,e). on(e,table). on(f,g). on(g,table).
}
```

The **beliefs** keyword is used to indicate that what follows are the *initial* beliefs of the agent. This part of the agent program is also called the **beliefs** section. The facts that are included in this section are used to initialize the belief base of the GOAL agent. These facts may change because, for example, the agent performs actions. A similar remark as for the **knowledge** section applies to the **beliefs** section: the code within this section must be a Prolog program.

Perception
Since facts about the current state of the environment typically are collected through the sensors available to the agent, the **beliefs** section of a GOAL agent program often is empty. The reason is simply that usually the initial state is unknown and the initial beliefs need to be obtained by perceiving the initial state of the environment.

The **beliefs** section above consists of facts only. This is not required, however, and a **beliefs** section may also include rules. The definition of `clear(X)` in the **knowledge** section might have also been moved to the **beliefs** section. In fact, since rules cannot be modified anyway, just from an informational perspective nothing would change. Below we will clarify, however, that when goals are taken into account as well the choice to put the definition of `clear(X)` in the **beliefs** section does change things. To repeat, as a rule of thumb, it is a good idea to include rules in the knowledge base and not in the belief base; the main reason for making exceptions to this rule will be discussed below.

The discussion about the Closed World Assumption and the definition of the `clear(X)` predicate should make clear that the knowledge base in isolation is not sufficient to make correct inferences about the Blocks World. Only by combining the knowledge with the beliefs the agent will be able to do so. This is exactly what happens in GOAL. That is, a GOAL agent derives conclusions from the combination of its knowledge and beliefs. This combination allows an agent to infer facts about the current state that it believes it is in by using the rules present in the **knowledge** section. For example, from the knowledge present in the **knowledge** section above in combination with the beliefs in the **beliefs** section the agent may derive that `clear(a)` holds. It is able to infer that block `a` is clear by means of the rule `clear(X) :- block(X), not(on(_, X))` and the absence of any information in the belief base that there is a block on top of `a`.

Prolog Definitions
It is not possible to *define* one and the same predicate in both the **knowledge** as well as in the **beliefs** section of an agent program. That is, one and the same predicate can not occur in the head of a Prolog rule that appears in the **knowledge** section as well as in the head of a Prolog rule that appears in the **beliefs** section. Even if this were possible it is not considered good practice to define a Prolog predicate at two different places. It is best practice to keep clauses that define a predicate (called a *procedure* in Prolog) close together.

We now turn to the representation of *goals*. For our example domain, illustrated in Figure 2.1, it is rather straightforward to write down a clause that represents the goal state. We thus get:

```
goals{
   on(a,e), on(b,table), on(c,table), on(d,c), on(e,b), on(f,d), on(g,table).
}
```

Note that in a GOAL agent program the **goals** keyword is used to indicate that what follows is wanted by the agent. The goal(s) present in this section of a GOAL agent program define the initial *goal base* of the agent. A goal base thus consists of all goals that the agent initially wants to achieve.[5] Intuitively, what is wanted should not be the case yet. The main reason is that a

[5]Note that if the goal base is implemented as a Prolog database, the storage of *negative* literals is not supported and goals to achieve states where a condition does *not* hold cannot be explicitly specified.

rational agent should not spent its resources on trying to realize something that is already the case. It is easy to verify that the clause above represents the goal state depicted in Figure 2.1 and is different from the initial state.

Types of Goals
Goals that are not currently the case but need to be realized at some moment in the future are also called *achievement goals*. In order to realize such goals, an agent needs to perform actions to *change* the current state of its environment to ensure the *desired* state. A goal of realizing a particular configuration of blocks different from the current configuration is a typical example of an achievement goal. Achievement goals are just one type of goal among a number of different types of goals. At the opposite of the goal spectrum are so-called *maintenance goals*. In order to realize a maintenance goal, an agent needs to perform actions, or, possibly, refrain from performing actions, to *maintain* a specific condition in its environment to ensure that this condition does *not* change. A typical example in the literature is a robot that needs to maintain a minimum level of battery power (which may require the robot to return to a charging station). In between are goals that require an agent to maintain a particular condition until some other condition is achieved. An agent, for example, may want to keep unstacking blocks until a particular block needed to achieve a goal configuration is clear (and can be moved to its goal position).

The careful reader will have noted one important difference between the representation of the goal state in the **goals** section above and the representation of the initial state in the **beliefs** section. The difference is that the goal state of Figure 2.1 is represented as a single *conjunction* in the **goals** section. That is, the atoms describing the goal state are separated by means of a comma `,`. The reason is that each of the facts present in the **goals** section need to be achieved *simultaneously*. If these facts would have been included as clauses separated by the period-symbol this would have indicated that the agent has *multiple, independent goals*. The following specification of the goals of an agent thus is *not* the same as the specification of the goal state above:

```
goals{
   on(a,e). on(b,table). on(c,table). on(d,c). on(e,b). on(f,d). on(g,table).
}
```

The main difference concerns the number of goals. The conjunctive goal counts as a *single* goal, whereas the 7 atoms separated by the period symbol count as 7 distinct, independent goals. To have two separate goals `on(a,b)` and `on(b,c)` instead of a single conjunctive goal `on(a,b), on(b,c)` is quite different. Whereas having two separate goals does not pose any restrictions on the order of achieving them, combining these same goals into a single conjunctive goal does require an agent to achieve both (sub)goals *simultaneously*. This also restricts the actions the agent can perform to achieve the goal. In the first case with two separate goals, the agent may, for example, put `a` on top of `b`, remove `a` again from `b`, and put `b` on top of `c`. Obviously, that would not achieve a state where `a` is on top of `b` which is on top of `c` at the same time.[6] It thus is important to keep in mind that, from a logical point of view, the period-symbol separator in the **beliefs** (and **knowledge** section) means the same as the conjunction operator represented by the comma-symbol, but that the meaning of these separators is different in the **goals** section. In the **goals** section the conjunction operator is used to indicate that facts are part of a *single* goal whereas the period-symbol separator is used to represent that an agent has *several different* goals that need not be achieved simultaneously. As separate goals may be achieved at different

[6]These observations are related to the famous *Sussman anomaly*. Early planners were not able to solve simple Blocks World problems because they constructed plans for subgoals (parts of the larger goal) that could not be combined into a plan to achieve the main goal. The Sussman anomaly provides an example of a Blocks World problem that such planners could not solve, see e.g. [25].

times it is also allowed that single goals when they are taken together are *inconsistent*, where this is not allowed in the **beliefs** section of an agent. For example, an agent might have the two goals `on(a,b)` and `on(b,a)`. Obviously these cannot be achieved simultaneously, but they can be achieved one after the other.

Goals Must Be Ground
The goals in a **goals** section must be ground, i.e. they should not contain any free variables. The main issue is that it is not clear what the meaning of a goal with a free variable is. For example, what would the goal `on(X,table)` mean? In Prolog, the reading would depend on whether `on(X,table)` is a clause (or rule with empty body) or a query. In the former case, a goal `on(X,table)` would be implicitly universally quantified and the meaning would be that *everything* should be on the table (but how does an agent compute *everything*?). Taking `on(X,table)` to be a query would yield an existential reading and would mean that the goal would be to have *something* on the table. Both readings give rise to problems, however, since we want to be able to *store* `on(X,table)` in a database supported by the underlying knowledge representation language *and* we want to be able to use `on(X,table)` as a *query* to verify whether it is believed by the agent (to check whether the goal has been achieved). Using Prolog this gives rise to both readings. That is, the storage in a database assumes implicit universal quantification and the use as query assumes existential quantification. We can not have it both ways and therefore using Prolog it is not possible to offer support for goals with free variables.

The content of a **goals** section thus does not need to be a Prolog program because the comma separator `,` may be used to construct conjunctive goals. The conjuncts in a goal may be atoms such as above. They may also be rules, however. We could, for example, have written the following program code:

```
goals{
   on(a,e), on(b,table), on(c,table), on(d,c), on(e,b), on(f,d), on(g,table),
   (clear(X) :- not(on(_, X))).
}
```

The first thing to note here is that additional brackets are needed to be able to parse the rule for `clear(X)` correctly. The second thing is that the rule has been added as a conjunct to the larger conjunctive goal to ensure is will be associated with these other subgoals. Intuitively, it is clear that it should be possible to infer that `clear(a)`, `clear(f)`, and `clear(g)` are derived goals, and that is also in fact the case in GOAL. At the same time, however, it is not so clear why the rule for `clear(X)` should be included in the **goals** section as part of the larger goal of the agent. First of all, the definition of `clear` seems to be *conceptual knowledge* and it would be more adequate as well as natural to have to specify this type of knowledge only once. That is, since the rule for `clear` is already present in the knowledge base, why should it be repeated in the goal base? Second, also from a designer's perspective such duplication of code seems undesirable. Duplication of code increases the risk of introducing bugs (by e.g. changing code at one place but forgetting to change it also at another place).

This motivates the practice in GOAL to support reasoning with goals in combination with (background) knowledge. If a rule, in the example above the rule for `clear` is already present in the knowledge base, the rule does not also need to be included in the goal base. Queries with respect to a goal in the **goals** section in GOAL are always performed in combination with the available knowledge. This means that e.g. given the goal state of Figure 2.1 represented as above it is possible to infer `clear(a)` from that goal using the rule for `clear` in the **knowledge** section.

Another illustration of the usefulness of combining conceptual, background knowledge with goals to infer derived goals can be provided by the *tower* concept in the Blocks World. It turns out that this concept is particularly useful for defining when a block is in position or misplaced. In

order to provide such a definition, however, we need to be able to derive that an agent has the goal of realizing a particular tower. It is clear that this cannot be derived from the information present in the **goals** section above. We first need to define and introduce the conceptual knowledge related to the notion of a tower. The following code provides a definition of the `tower` predicate:

```
tower([X]) :- on(X, table).
tower([X,Y|T]) :- on(X, Y), tower([Y| T]).
```

The rules that define the `tower` predicate specify when a list of blocks `[X|T]` is a tower. The first rule `tower([X]) :- on(X,table)` requires that the basis of a tower is grounded on the table. The second rule recursively defines that whenever `[Y|T]` is a tower, extending this tower with a block `X` on top of `Y` also yields a tower; that is, `[X,Y|T]` is a tower. Observe that the definition does not require that block `X` is clear. As a result, a stack of blocks that is part of a larger tower also is considered to be a tower. For example, it is possible to derive that `tower([b,c])` using the facts representing the initial state depicted in Figure 2.1.

Following the rule of thumb introduced earlier, the rules for `tower` should be included in the **knowledge** section. The benefit of doing this is that it makes it possible to also use the tower concept to derive goals about towers. For example, it is possible to infer from the representation of the goal state of Figure 2.1 that the agent wants `tower([e,b])`. The agent can infer this (sub)goal by means of the rules that define the `tower` predicate in the knowledge base and the (sub)goals `on(b,table)` and `on(e,b)` in the goal base.

Summarizing, and putting everything together, we obtain the part of the agent program that specifies the initial mental state of the agent, see Table 2.2.

```
knowledge{
   block(a). block(b). block(c). block(d). block(e). block(f). block(g).
   clear(table).
   clear(X) :- block(X), not(on(_,X)).
   tower([X]) :- on(X,table).
   tower([X,Y|T]) :- on(X,Y), tower([Y|T]).
}
beliefs{
   on(a,b). on(b,c). on(c,table). on(d,e). on(e,table). on(f,g). on(g,table).
}
goals{
   on(a,e), on(b,table), on(c,table), on(d,c), on(e,b), on(f,d), on(g,table).
}
```

Figure 2.2: Mental State of a Blocks World Agent That Represents Figure 2.1

Where to Put Rules?
The rule of thumb introduced is to put rules in the **knowledge** section. This facilitates reasoning with background or domain knowledge in combination with both the agent's beliefs as well as its goals. Sometimes it may be undesirable, however, to combine background knowledge with goals as it may be possible to infer goals that should not be inferred. To give an example, although it may be perfectly reasonable to require that the block at the top of a goal tower is clear, it is just as reasonable to not require this. However, by inserting the rule for `clear(X)` in the knowledge base, the agent will always infer that it is desired that the top of a goal tower is clear. The difference, of course, is that the agent would sometimes have to remove a block to achieve this goal whereas in the absence of such goals no such action would be needed. For example, when `tower([a,b,c])` is believed to be true and the agent wants to realize `tower([b,c])`, the agent would need to move `a` to the table in order to make sure that `b` is clear. If that is not intended, it may be better to put the rule for `clear` in the **beliefs** section instead of the **knowledge** section as in that case no `clear` goals will be inferred (only *knowledge* is combined with goals).
The choice to move the rule for `clear` to the **beliefs** section prevents the inference that `clear(b)` is a goal in the example where the agent wants `tower([b,c])`. As the careful reader will have noted, however, this solution does not resolve the problem completely since the cause of the problem in fact stems from the Closed World Assumption. Although we cannot derive `clear(b)` anymore by moving the rule, we can still derive `not(on(a,b))`. Even stronger, we can also derive `not(on(_,b))` which expresses that there should not be any block on top of `b`.

2.2 Inspecting an Agent's Mental State

Agents that derive their choice of action from their beliefs and goals need the ability to inspect their mental state. In GOAL, *mental state conditions* provide the means to do so. These conditions are used in action rules to determine which actions the agent will consider performing (see Chapter 4).

2.2.1 Mental Atoms

A mental state condition consists of *mental atom*s. A mental atom either is a condition of the belief base or on the goal base of an agent. Conditions on the belief base are of the form `bel(`φ`)` and conditions on the goal base are of the form `goal(`φ`)` where φ is a query of the knowledge representation language that is used (e.g. a Prolog query).

Informally, `bel(`φ`)` means that the the agent believes that φ. In GOAL, that is the case when φ follows from the agent's knowledge and beliefs. In other words, `bel(`φ`)` holds whenever φ can be derived from the content of the agent's belief base *in combination with* the content of the knowledge base. Continuing the example of Figure 2.2, in the initial state it follows that `bel(clear(a))`. In this state, the agent thus believes that block `a` is clear.

Informally, `goal(`φ`)` means that the agent has a goal that φ. In GOAL, that is the case when φ follows from *one* of the agent's goals and its knowledge. That is, `goal(`φ`)` holds whenever φ can be derived from *a single goal* in the goal base of the agent *in combination with* the content of the knowledge base.[7] In the example of Figure 2.2 it follows that `goal(tower([e,b]))`. This follows from the definition of the `tower` predicate in the **knowledge** section and the (sub)goals `on(b,table)` and `on(e,b)` that are part of the single goal present in the goal base in Figure 2.2.

[7] This reading differs from that provided in [11] where the goal operator is used to denote *achievement goals*. The difference is that a goal is an achievement goal only if the agent *does not believe that* φ. The goal operator `goal` introduced here is a more basic operator. It can be used in combination with the belief operator `bel` to *define* achievement goals.

2.2.2 Mental State Conditions

Mental state conditions are constructed using mental atoms. Mental atoms `bel(`φ`)` and `goal(`φ`)` are mental state conditions. And so are the *negations* `not(bel(`φ`))` and `not(goal(`φ`))` of these mental atoms. Plain mental atoms and their negations are also called *mental literals*. A mental state condition then can be defined as a conjunction of mental literals, where single mental literal are also included. For example,

```
goal(on(b,table)), not(bel(on(b,table))
```

is a mental state condition. It expresses that the agent has a goal that block `b` is on the table but does not believe that this is the case (yet). It is not allowed to use disjunction in a mental state condition.

> Although disjunction cannot be used to combine mental state conditions, intuitively a disjunctive mental state condition of the form $\psi_1 \vee \psi_2$ would provide two different reasons for selecting an action: either if ψ_1 is the case, possibly select the action, or else possibly select the action if ψ_2 is the case. In such cases, multiple action rules can be used to select an action in either of these cases. Also note that negation can be distributed over conjunction and disjunction, and we have that $\neg(\psi_1 \wedge \psi_2)$ is equivalent to $\neg\psi_1 \vee \neg\psi_2$. By using this transformations, and the fact that instead of disjunction equivalently multiple rules may be introduced, negation in combination with conjunction is sufficient.

Goals as in the example that need to be achieved because they are not yet the case are also called *achievement goals*. Achievement goals are important reasons for choosing to perform an action. For this reason they are frequently used in GOAL programs and it is useful to introduce an additional operator `a-goal(`φ`)` that is an abbreviation of mental state conditions of the form `goal(`φ`), not(bel(`φ`))`.[8]

$$\texttt{a-goal(}\varphi\texttt{)} \stackrel{df}{=} \texttt{goal(}\varphi\texttt{), not(bel(}\varphi\texttt{))}$$

Interestingly, this operator provides what is needed to express that a block is misplaced as a block is misplaced whenever the agent believes that the block's current position is different from the position the agent wants it to be in.[9] As the position of the tower which a block is part of is irrelevant, the fact that a block `X` is not in position can be represented by `a-goal(tower([X|T]))` where `T` is a tower. `a-goal(tower([X|T]))` expresses that in the goal state block `X` is on top of the tower `T` but the agent does not believe that this is already so in the current state. The concept of a misplaced block is important for defining a strategy to resolve a Blocks World problem, since - assuming goal states are complete - only misplaced blocks have to be moved, and can be expressed easily and elegantly in GOAL using mental state conditions.[10]

Another useful mental state condition is `goal(`φ`), bel(`φ`)` which expresses that a (sub)goal has been achieved. Instantiating the template φ with `tower([X|T])`, this condition expresses that the current position of a block `X` corresponds with the position it has in the goal state.[11]

[8] See [31] for a discussion of this definition.

[9] Actually, here the difference between *knowledge* and *belief* is important as we normally would say something is misplaced only if we *know* that the block is in a different position. That is, an agent's beliefs about the block's position must also correspond with the actual position of the block. If, in fact, the block is in the desired position, in ordinary language, we would say that the block is *believed to be misplaced* but that in fact it is not.

[10] Another important concept, that of a block that is in a *self-deadlock* is also easily expressed by means of the mental state condition `a-goal(tower([X|T])), goal-a(above(X,Y))`. The operator `goal-a` is defined below. The first conjunct `a-goal(tower([X|T]))` expresses that `X` is misplaced and the second conjunct `goal-a(above(X,Y))` expresses that `X` is above some block `Y` in both the current state as well as in the goal state. This concept is just as important for solving Blocks World problems since any self-deadlocked block needs to be moved at least twice to reach the goal state. Moving such a block to the table thus will be a necessary move in every plan.

[11] Note that it would not be possible to express this using an achievement goal operator. The `goal-a` operator may be used to define the concept of a deadlock [56]; see also the previous footnote.

In this case φ is a (sub)goal that is achieved and we call such a (sub)goal a *goal achieved*. The operator `goal-a(`φ`)` is introduced as an abbreviation to denote such goals.

$$\texttt{goal-a(}\varphi\texttt{)} \stackrel{df}{=} \texttt{goal(}\varphi\texttt{), bel(}\varphi\texttt{)}$$

The condition `a-goal(tower([X,Y|T])), bel(tower([Y|T])` provides another useful example of a mental state condition. It expresses that the achievement goal to construct a tower `tower([X,Y|T]))` has been realized except for the fact that block `X` is not yet on top of tower `[Y|T]`. It is clear that whenever it is possible to move block `X` on top of block `Y` the agent would get closer to achieving (one of) its goals. Such a move, moreover, would be a *constructive move* which means that the block would never have to be moved again. As the possibility to make a move may be verified by checking whether the precondition of the move action holds (see chapter 3), in combination with the mental state condition, we are able to verify whether a constructive move can be made. This condition therefore is very useful to define a strategy for solving Blocks World problems, and will be used in the chapter 4 to define a rule for selecting actions.

Finally, the expression `not(goal(true))` may be used to verify that an agent has *no* goals. This can be explained as follows: if `true` does not follow from any of the goals of the agent, this means the agent cannot have any goals, since `true` would follow from any goal. In other words, `not(goal(true))` means that the goal base of the agent is empty.

2.3 Notes

Mental states discussed in this chapter consist of the knowledge, beliefs and goals of an agent about its environment. An agent with such a mental state is a first-order intentional system (cf. Chapter 1). Such agents maintain representations of their environment but not about their own or other agent's beliefs. Of course, when other agents are part of an agent's environment, the presence of these other agents and what they do may also be represented in the mental state of the agent. In various situations it may be useful, however, to be able to represent other agent's beliefs and goals. This would require the ability to represent other agent's beliefs and goals, and agents that are second- or higher-order intentional systems. Agents would also require some kind of Theory of Mind, to be able to reason with their beliefs about other agents. In Chapter 6 we will introduce the tools to represent other agent's mental states.

2.4 Exercises

```
knowledge{
  block(a). block(b). block(c). block(d). block(e).
  clear(X) :- block(X), not(on(Y,X)).
  clear(table).
  tower([X]) :- on(X,table).
  tower([X,Y|T]) :- on(X,Y), tower([Y|T]).
  above(X,Y) :- on(X,Y).
  above(X,Y) :- on(X,Z), above(Z,Y).
}
beliefs{
  on(a,d). on(b,table). on(c,table). on(d,b). on(e,a).
}
goals{
  on(a,b), on(b,c), on(c,d), on(d,table).
}
```

Figure 2.3: Mental State

2.4.1

Consider the mental state in Figure 2.3. Will the following mental state conditions succeed? Provide all possible instantiations of variables, or otherwise a conjunct that fails.

1. `goal(above(X,d), tower([d,X|T]))`
2. `bel(above(a,X)), goal(above(a,X))`
3. `bel(clear(X), on(X,Y), not(Y=table)), goal(above(Z,X))`
4. `bel(above(X,Y), a-goal(tower([X,Y|T]))`

2.4.2

As explained,

`a-goal(tower([X|T))`

may be used to express that block `X` is *misplaced*. A block, however, is only misplaced if it is part of the goal state. This is not the case for block `e` in Figure 2.3. Block `e` is, however, *in the way*. This means that the block prevents moving another block that is misplaced, and therefore needs to be moved itself (in the example in Figure 2.3, we have that `on(e,a)` where `a` is misplaced but cannot be moved without first moving `e`).
Provide a mental state condition that expresses that a block `X` is *in the way* in the sense explained, and explain why it expresses that block `X` is in the way. Keep in mind that the block below `X` may also not be part of the goal state! Only use predicates available in Figure 2.3.

Chapter 3

Actions and Change

> An underlying premise in much work addressing the design of intelligent agents or programs is that such agents should (either implicitly or explicitly) hold beliefs about the true state of the world. Typically, these beliefs are incomplete, for there is much an agent will not know about its environment. In realistic settings one must also expect an agent's beliefs to be incorrect from time to time. If an agent is in a position to make observations and detect such errors, a mechanism is required whereby the agent can change its beliefs to incorporate new information. Finally, an agent that finds itself in a dynamic, evolving environment (including evolution brought about by its own actions) will be required to change its beliefs about the environment as the environment evolves.
>
> Quote from: [13]

An agent performs actions to effect changes in its environment in order to achieve its goals. Actions to change the environment typically can be performed only in specific circumstances and have specific effects. The conditions which need to be true to successfully execute an action and the way the state of the environment changes are captured by an action specification. A programmer writing a GOAL agent program needs to provide action specifications for all actions that are part of the agent's action repertoire to affect the agent's environment.

3.1 Action Specifications

Unlike other programming languages, but similar to planners, actions that may be performed by a GOAL agent need to be specified by the programmer of that agent. GOAL does provide some special built-in actions but typically most actions that an agent may perform are made available by the environment that the agent acts in. Actions are specified in the **actionspec** section of a GOAL agent. These actions are called *user-defined actions*. Actions are specified by specifying the conditions when an action can be performed and the effects of performing the action. The former are also called *preconditions* and the latter are also called *postconditions*. The **actionspec** section in a GOAL agent program consists of a set of STRIPS-style specifications [38] of the form:

actionname{
  `pre{`*precondition*`}`
  `post{`*postcondition*`}`
}

The *action* specifies the *name* of the action and its *arguments or parameters* and is of the form *name[args]*, where *name* denotes the name of the action and the *[args]* part denotes an *optional* list of parameters of the form $(t_1, \ldots, t_n)$, where the t_i are terms. The parameters

of an action may contain free variables which are instantiated at runtime. Actions without any parameters consist of a name only without brackets. One of the most simple action specifications is the following:

```
skip{
  { true }
  { true }
}
```

This action specification specifies the action `skip` without parameters that can always be performed and has no effect. The **true** precondition indicates that `skip` can always be performed, whereas the **true** postcondition indicates that the action has no effect. It would also have been possible to simply write an empty pre- and postconditions {}; in other words, there is no need to write **true** explicitly.

If an agent is connected to an environment, user-defined actions will be sent to the environment for execution. In that case it is important that the name of an action corresponds with the name the environment expects to receive when it is requested to perform the action.

Continuing the Blocks World example introduced in Chapter 2, we will now describe the robot gripper that enables the agent that controls this gripper to pick up and move blocks. The robot gripper can pick up at most one block at a time. It thus is not able to pick up one (let alone more) stack of blocks. As is common in the simplified version of the Blocks World, we abstract from all physical aspects of robot grippers. In its most abstract form, the gripper in the Blocks World is modeled as a robot arm that can perform the single action of picking up and moving one block to a new destination in one go. That is, the picking up and moving are taken as a single action that can be performed instantaneously. This action can be specified in GOAL as follows:

```
actionspec{
   move(X,Y) {
     pre{ clear(X), clear(Y), on(X,Z), not(X=Y) }
     post{ not(on(X,Z)), on(X,Y) }
   }
}
```

The **actionspec** keyword indicates the beginning of the action specification section in a GOALagent. In the code fragment above the single action `move(X,Y)` has been specified.

Multiple Action Specifications for One and the Same Action

GOAL allows multiple action specifications for the same action. Providing multiple action specifications for one and the same action may be useful to distinguish cases. For example, in the wumpus environment [50], the agent may turn either left or right. The `turn` action does not have any preconditions and is always enabled. Of course, the effects of turning left or right are different. An easy way to differentiate these cases is to provide two action specifications for the `turn` action as follows:

```
actionspec{
   turn(X) {
           pre{ X=left, ori(Ori), leftOri(OriNew) }
           post{ not(ori(Ori)), ori(OriNew) }
   }
   turn(X) {
     pre{ X=right, ori(Ori), rightOri(OriNew) }
     post{ not(ori(Ori)), ori(OriNew) }
   }
}
```

Note that even though the `turn` action can always be performed the preconditions in the action specifications are not empty. To be able to specify the effects of the action, the new orientation is computed by means of the `leftOri` and the `rightOri` predicates. It also is possible to instantiate the variable parameter of the `turn` action. In other words, writing `turn(left)` instead of `turn(X)` would have been ok, which would also allow for removing the conditions `X=left` and `X=right` from the preconditions.

3.1.1 Preconditions

The keyword **pre** indicates that what follows is a *precondition* of the action. Preconditions are queries of the knowledge representation language that is used (e.g. a Prolog query). Preconditions are used to verify whether it is possible to perform an action. A precondition φ is evaluated by verifying whether φ can be derived from the belief base (as always, in combination with knowledge in the knowledge base). Free variables in a precondition may be instantiated during this process just like executing a Prolog program returns instantiations of variables. An action is said to be *enabled* whenever its precondition is believed to be the case by the agent.[1]

The precondition for the `move` action specifies that in order to be able to perform action `move(X,Y)` of moving `X` on top of `Y` both `X` and `Y` have to be clear. The condition `not(X=Y)` expresses that it is not possible to move a block or the table on top of itself.

The condition `on(X,Z)` in the precondition does *not* specify a condition that needs to be true in order to be able to perform the `move` action in the environment. The condition checks on top of what `Z` block `X` currently is located. It retrieves in variable `Z` the thing, i.e. block or table, that block `X` is currently on. The reason for including this condition in the precondition is that it is needed to be able to specify the effects of the `move` action. After moving block `X` the block obviously no longer is where it was before. This fact about `X` thus must be removed after performing the action, which only is possible if the current fact is retrieved from the agent's belief base first.

It is best practice to include in the precondition of an action specification only those conditions that are *required* for the successful execution of an action. These conditions include conditions that are needed to specify the effects of an action as the `on(X,Z)` atom in the specification of the `move` action. In other words, a precondition should only include conditions that need to hold in order to be able to successfully perform the action. Conditions such as, for example, conditions that specify when it is a good idea to choose to perform the action should be specified in the **program** section of the agent. These are strategic considerations and not strictly required for the successful execution of action. Including such conditions would prevent an action from being selected even when the action could be successfully executed in the environment.[2]

Preconditions do not always need to include all conditions that are required for the successful execution of an action. Although it is good practice to provide complete action specifications, sometimes there are pragmatic reasons for not providing all required preconditions. For example, the precondition for the `move` action does not require that `X` is a block. Strictly speaking, the precondition including the condition `block(X)` would result in a better match with constraints in the Blocks World because only blocks can be moved in this environment. Adding the `block(X)` condition would have resulted in a more *complete* action specification. However, when efficiency is considered as well, it makes sense to provide a precondition that checks as few conditions as possible. It may also be that a precondition is better readable when not all details are included. We thus may trade off completeness of an action specification for efficiency and readability. A programmer should always verify, however, that by not including some of the actual preconditions of an action the program still operates as expected. As we will see in Chapter 4, the mental state conditions of an action rule that provide the reason for selecting a move action prevent variable `X` from ever being instantiated with `table`. Dropping the condition `block(X)` from the precondition therefore does not do any harm.

Another illustration of a condition that is not included in the precondition of the `move` action is related to specific moves that are redundant. The precondition of the `move` action does not require that `Y`, the place to move `X` onto, is different from `Z`, the thing `X` stands on top of (retrieved by the `on(X,Z)` atom). In this case, there is nothing really lacking in the precondition, but the

[1] Note that because an agent may have false beliefs the action may not actually be enabled in the environment. An agent does not have direct access to environments. Instead it needs to rely on its beliefs about the environment to select actions. From the perspective of an agent an action thus is enabled if it believes it is.

[2] Because a precondition is checked against the beliefs of an agent it may not actually be the case that an action can be performed successfully. Still a precondition should specify those conditions that would guarantee successfull execution. In that case, since an agent does not have direct access to the environment, for all the agent knows, the action can be performed successfully if it believes the precondition to be true.

condition may be added to prevent particular redundant moves. As the precondition does require that `Y` is clear, whenever we have that both `on(X,Z)` and `Y=Z` it must be the case that `Y` is equal to the `table`. We thus would have that a move of `X` onto the `table` is performed whereas block `X` already is on the table. Clearly, performing such a move is redundant. For this reason, although not strictly required, we might decide to add the condition `not(Y=Z)`.

Actions are enabled when their preconditions hold. But there is one additional general condition that must be fulfilled before an action is enabled: All variables that occur in the parameters of the action and in the postcondition must be instantiated to ensure the action and its effects are well-specified. That is, all free variables in parameters of an action as well as in the postcondition of the action must have been instantiated with ground terms. This is also the case for built-in actions. For the `move` action specified above, this implies that all the variables `X`, `Y`, and `Z` need to be instantiated with ground terms that refer either to a block or to the table.

Completely Instantiated Actions
The reason that the variables that occur in an action's parameters or in its postcondition must be instantiated with ground terms is that otherwise it is not clear what it means to execute the action. What, for example, does it mean to perform the action `move(X,table)` with `X` a variable? One option would be to select a random item and try to move that. Another would be to try and move all items. But shouldn't we exclude tables? We know intuitively that the `move` action moves blocks only. But how would the agent program know this?
[50] requires that all variables in the precondition also appear in the parameters of the action. This requires that instead of the `move(Block,To)` action specified above an action `moveFromTo(Block,From,To)` must be specified as the `From` variable needs to included in the action's parameters. This restriction is lifted in GOAL and no such constraints apply. As mentioned, in order to ensure an action's postcondition is closed, all variables that occur in the postcondition must also occur in either the precondition or the action's parameters. Note that any variables in action parameters that do not occur in the precondition must be instantiated by other means before the action can be performed; in a GOAL agent program this can be done by means of mental state conditions.

3.1.2 Postconditions

The keyword **post** indicates that what follows is a *postcondition* of the action. Postconditions are conjunctions of literals (Prolog literals when Prolog is used as the knowledge representation language). A *postcondition* specifies the effect of an action. In GOAL, effects of an action are changes to the mental state of an agent. The effect φ of an action is used to update the beliefs of the agent to ensure the agent believes φ after performing the action. In line with STRIPS terminology, a *postcondition* φ is also called an *add/delete list* (see also [25, 38]). Positive literals φ in a postcondition are said to be part of the add list whereas negative literals `not(`φ`)` are said to be part of the delete list. The effect of performing an action is that it updates the belief base by *first* removing all facts φ present in the delete list and *thereafter* adding all facts present in the add list. Finally, as an action can only be performed when all free variables in the postcondition have been instantiated, each variable present in a postcondition must also be present in the action parameters or precondition of the action.

Closed World Assumption (Continued)
The STRIPS approach to updating the state of an agent after performing an action makes the Closed World Assumption. It assumes that the database of beliefs of the agent consists of *positive* facts only. Negative information is inferred by means of the Closed World Assumption (or, more precisely, negation as failure in the case we use Prolog). Negative literals in the postcondition are taken as instructions to remove positive information that is present in the belief base. The action language ADL does not make the Closed World Assumption and allows to add negative literals to an agent's belief base [50].

The postcondition `not(on(X,Z)), on(X,Y)` in the action specification for `move(X,Y)` consists of one negative and one positive literal. The add list in this case thus consists of a single atom `on(X,Y)` and the delete list consists of the atom `on(X,Z)`. It is important to remember that all variables must have been instantiated when performing an action and that both `on(X,Y)` and `on(X,Z)` are only templates. Upon performing the action, the *immediate* effect of executing the action is to update the agent's mental state by first removing all atoms on the delete list and thereafter adding all atoms on the add list. For the `move` action this means that the current position `on(X,Z)` of block `X` is removed from the belief base and thereafter the new position `on(X,Y)` is added to it.

Inconsistent Postconditions
A remark is in order here because, as discussed above, it may be the case that `Z` is equal to `Y`. In that case the postcondition would be of the form `not(on(X,Y)), on(X,Y)` where `Y` is substituted for `Z`. This postcondition is a contradiction and, from a logical point of view, can never be established. However, though logically invalid, nothing would go wrong when a move action where `Y` and `Z` are equal (`Y = Z = table`) would be executed. The reason is that the STRIPS semantics for updating the beliefs of an agent given the postcondition is a *procedural* semantics. The procedure for updating proceeds in steps, which, in this case, would first delete `on(X,Y)` from the belief base if present, and thereafter would add `on(X,Y)` (again, if initally present already, and nothing would have changed). Although this procedural semantics prevents acute problems that might arise because of contradictory postconditions, it is most of the time considered best practice to prevent such situations from arising in the first place. This therefore provides another reason for adding `not(Y=Z)` to the precondition.

3.1.3 Updating an Agent's Goals

Performing an action may also affect the goal base of an agent. As a rational agent should not invest resources such as energy or time into achieving a goal that has been realized, such goals are removed from the goal base. That is, goals in the goal base that have been achieved as a result of performing an action are removed. Goals are removed from the goal base, however, only if they have been *completely* achieved. The idea here is that a goal φ in the goal base is achieved only when all of its subgoals are achieved. An agent should not drop any of these subgoals before achieving the overall goal as this would make it impossible for the agent to ensure the overall goal is achieved at a single moment in time. The fact that a goal is only removed when it has been achieved implements a so-called *blind commitment strategy* [48]. Agents should be committed to achieving their goals and should not drop goals without reason. The default strategy for dropping a goal in GOAL is rather strict: only do this when the goal has been completely achieved. This default strategy can be adapted by the programmer for particular goals by using the built-in **drop** action.

For example, a state with belief base and goal base with the following content is rational:

```
beliefs{
   on(b,c). on(c, table).
}
```

```
goals{
   on(a,b), on(b,c), on(c, table).
}
```

but after successfully performing the action `move(a,b)` in this state, the following state would result if we would not remove goals automatically:

```
beliefs{
   on(a,b). on(b,c). on(c, table).
}
goals{
   on(a,b), on(b,c), on(c, table).
}
```

This belief state in combination with the goal state is *not* rational. The point is that the goal has been achieved and, for this reason, should be removed as a rational agent would not invest any resources in it. GOAL implements this *rationality constraint* and automatically removes achieved goals, resulting in our example case in a state with an empty goal base:

```
beliefs{
   on(a,b). on(b,c). on(c, table).
}
goals{

}
```

3.2 Built-in Actions

In addition to the possibility of specifying user-defined actions, GOAL provides several built-in actions for changing the beliefs and goals of an agent, and for communicating with other agents. Here we discuss the built-in actions for modifying an agent's mental state. The `send` action for communicating a message to another agent is discussed in Chapter 6. There are four built-in actions to modify an agent's mental state.

We first discuss the actions for changing the agent's goal base. The action:

```
adopt(φ)
```

is the built-in **adopt** action for adopting a new goal. The precondition of this action is that the agent does not believe that φ is the case, i.e. in order to execute **adopt**`(`φ`)` we must have `not(bel(`φ`))`. The idea is that it would not be rational to adopt a goal that has already been achieved. The effect of the action is the addition of φ as a single, new goal to the goal base. We remark here that the goal base consists of conjunctions of positive literals only. As a consequence, the action **adopt**`(not(posliteral))` does *not* insert a goal `not(posliteral)` in the goal base. Note that this action also does *not remove* `posliteral` from the goal base; for removing a goal the **drop**`(posliteral)` action should be used.

The action:

```
drop(φ)
```

is the built-in **drop** action for dropping current goals of an agent. The precondition of this action is always true and the action can always be performed. The effect of the action is that any goal in the goal base from which φ can be derived is removed from the goal base. For example, the action **drop**`(on(a,e))` would remove all goals in the goal base that entail `on(a,e)`; in the example agent of Figure 2.2 the only goal present in the goal base would be removed by this action.

We now turn to actions to change the belief base of an agent. The action:

```
insert(φ)
```

is the built-in **insert** action for inserting φ into the agent's beliefs. In the case that Prolog is used as a knowledge representation language, φ must be a conjunction of literals. The semantics of **insert** in this case is similar to the update associated with the postcondition of an action. The **insert** action *adds* all positive literals that are part of φ to the belief base of the agent and removes all negative literals from the belief base. For example, **insert**`(not(on(a,b)), on(a,table))` *removes* the literal `on(a,b)` from the belief base and *adds* the positive literal `on(a,table)`. The **insert**(φ) action modifies the belief base such that φ follows from the belief base, whenever it is possible to modify the belief base in such a way.

Note that the goal base may be modified as well if due to such changes the agent would come to believe that one of its goals has been completely achieved; in that case, the goal is removed from the goal base, as discussed above.

Finally, the action:

```
delete(φ)
```

is the built-in **delete** action for removing φ from the agent's beliefs. The **delete** action is kind of the inverse of the **insert** action. Instead of adding literals that occur positively in φ it removes such literals, and, instead of removing negative literals it adds such literals.

The **delete** action does not add expressivity, but it does enhance readability of a program. We consider it best practice to *only use **insert** to add positive literals and only use **delete** to remove positive literals.* Instead of writing **insert**`(not(on(a,b)), on(a,table))` it is better to use the two actions **insert**`(on(a,table))` and **delete**`(on(a,b))`. Multiple actions can be combined into a single, complex action by the + operator. Using this operator we can write:

```
insert(on(a,table)) + delete(on(a,b))
```

which would have exactly the same effect as the single action **insert**`(not(on(a,b)), on(a,table))`. The + operator can be used to combine as many actions as needed into a single, complex action, as long as it respects the requirement that at most one user-defined action is included in the list of actions combined by +. The actions combined by + are executed in order of appearance (see also Section 4.5).

Inserting Versus Deleting
If Prolog is used as the knowledge representation, the effects of the **insert** action are in fact the same as the update associated with the postcondition according to the STRIPS semantics. And, because a Closed World Assumption is made, in effect, "inserting" negative information is the same as deleting positive information. It thus would be possible to program all intended changes to the belief base by means of the **insert** action only.
One thing to note about the **delete**(φ) action is that it removes *all* positive literals that occur in φ. The **delete** action does not support minimal change in the sense that as little information is removed to ensure that φ is not believed anymore. For example, **delete**`(p, q)` removes both `p` and `q` instead of only removing either `p` or `q` which would be sufficient to remove the belief that `p` and `q` both are true.

Finally, like all other actions, built-in actions must be closed when executed. That is, all variables must have been instantiated. This is also true for actions combined by the + operator.

3.3 Notes

The strategy for updating an agent's goals in GOAL is a *blind commitment strategy*. Using this strategy, an agent only drops a goal when it believes the goal has been achieved. In [48] and [17] various other strategies are introduced. One issue with an agent that uses a blind commitment strategy is that such an agent rather fanatically commits to a goal. It would be more rational if an agent would also drop a goal if it would come to believe the goal is no longer achievable. For example, an agent may drop a goal to get to a meeting before 3PM if it misses the train. However, providing automatic support for removing such goals when certain facts like missing a train are believed by the agent is far from trivial. GOAL provides the **drop** action which can be used to remove goals by writing action rules with such actions as conclusions.

3.4 Exercises

```
main: towerBuilder{
  knowledge{
    block(a). block(b). block(c). block(d). block(e). block(f).
    clear(table).
    clear(X) :- block(X), not(on(Y,X)), not(holding(X)).

    above(X,Y) :- on(X,Y).
    above(X,Y) :- on(X,Z), above(Z,Y).

    tower([X]) :- on(X,table).
    tower([X,Y|T]) :- on(X,Y), tower([Y|T]).
  }
  beliefs{
    on(a,table). on(b,c). on(c,table). on(d,e). on(e,f). on(f,table).
  }
  goals{
    on(a,table), on(b,a), on(c,b), on(e,table), on(f,e).
  }

  main module{
    program{
      % pick up a block that needs moving
      if a-goal(tower([X|T])), bel((above(Z,X);Z=X)) then pickup(Z).
      % constructive move
      if a-goal(tower([X,Y|T])), bel(tower([Y|T])) then putdown(X,Y).
      % put block on table
      if bel(holding(X)) then putdown(X,table).
    }
  }

  actionspec{
    pickup(X){
      pre{                }
      post{               }
    }
    putdown(X,Y){
      pre{                }
      post{               }
    }
  }
}
```

Figure 3.1: Incomplete Blocks World Agent Program

3.4.1

Consider the incomplete agent program for a Blocks World agent in Figure 3.1. Suppose that the agent's gripper can hold one block at a time, only clear blocks can be picked up, and blocks can only be put down on an empty spot. Taking these constraints into account, provide a full action specification for:

1. `pickup(X).`
2. `putdown(X,Y).`

Chapter 4

GOAL Agent Programs

This chapter provides an introduction into the basic language elements that can be used to program a GOAL agent. The GOAL language is introduced by means of an example that illustrates what a GOAL agent program looks like. The example of the Blocks World introduced in Chapter 2 will be used again for this purpose. We like to think of the Blocks World as the "hello world" example of agent programming (see also [56]). It is both simple and rich enough to demonstrate the use of various programming constructs in GOAL.

4.1 The Structure of an Agent Program

A GOAL agent decides which action to perform next based on its knowledge, beliefs and goals. As discussed in Chapter 2, the knowledge, beliefs and goals of an agent make up its *mental state*. A GOAL agent inspects and modifies this state at runtime similar to a Java method which operates on the state of an object. Agent programming in GOAL therefore can also be viewed as *programming with mental states*.

In a Blocks World the decision amounts to where to move a block, in a robotics domain it might be where to move to or whether to pick up something with a gripper or not. Such a decision typically depends on the current state of the agent's environment as well as general knowledge about this environment. In the Blocks World an agent needs to know what the current configuration of blocks is and needs to have basic knowledge about such configurations, such as what a tower of blocks is, to make a good decision. The former type of knowledge is typically *dynamic* and changes over time, whereas the latter typically is *static* and does not change over time. In line with this distinction, in GOAL two types of knowledge of an agent are distinguished: *conceptual or domain knowledge* stored in a *knowledge base* and *beliefs about the current state* of the environment stored in a *belief base*.

A decision to act will usually also depend on the goals of the agent. In the Blocks World, a decision to move a block on top of an existing tower of blocks would be made, for example, if it is a goal of the agent to have the block on top of that tower. In a robotics domain it might be that the robot has a goal to bring a package somewhere and therefore picks it up. Goals of an agent are stored in a *goal base*. The goals of an agent may change over time, for example, when the agent adopts a new goal or drops one of its goals.

As a rational agent should not pursue goals that it already believes to be achieved, such goals need to be removed. GOAL provides a built-in mechanism for doing so based on a so-called *blind commitment strategy*.

To select an action, a GOAL agent needs to be able to *inspect* its knowledge, beliefs and goals. An action may or may not be selected if certain things follow from an agent's mental state. For example, if a block is misplaced, that is, the current position of the block does not correspond with the agent's goals, the agent may decide to move it to the table. A GOAL programmer needs to write conditions called *mental state conditions* that were introduced in Chapter 2 in order to

verify whether the appropriate conditions for selecting an action are met. In essence, writing such conditions means specifying a *strategy* for action selection that will be used by the GOAL agent. Such a strategy is coded in GOAL by means of *action rules* which define when an action may or may not be selected.

After selecting an action, an agent needs to *perform* the action. Performing an action in GOAL means changing the agent's mental state. An action to move a block, for example, will change the agent's beliefs about the current position of the block. The effects of an action on the mental state of an agent need to be specified explicitly in a GOAL agent program by the programmer except for built-in actions. As discussed in Chapter 3, the conditions when an action can be performed, i.e. its *preconditions*, and the effects of performing the action, i.e. its *postcondition*), need to be specified. Whether or not a real (or simulated) block will also be moved in an (simulated) environment depends on whether the GOAL agent has been adequately connected to such an environment. Percepts received from an environment can be used to verify this. GOAL provides so-called *event* or *percept rules* to process received percepts and update an agent's mental state.

Goal Agent Program
A GOAL *agent program* consists of five sections:

knowledge: a set of concept definitions or domain rules, which represents the conceptual or domain knowledge the agent has about its environment. These definitions and rules are used to create the *knowledge base* of the agent.

beliefs: a set of facts and rules called *beliefs*, representing the initial state of affairs. These are used to create the *initial belief base* of the agent.

goals: a set of goals, representing in what state the agent wants to be. These goals are used to create the *initial goal base*.

module section: a set of modules, including at least one out of two special modules: the *main* or *event* module. Modules must have a *program* section that contains action rules. Action rules define a strategy or policy for action selection.

action specification: a specification of the pre- and post-conditions for each action the agent can perform in an environment. A pre-condition specifies *when* an action can be performed and a post-condition specifies the *effects* of performing the action.

Most of these sections are *optional* and do not need to be present in an agent program. The abscence of the knowledge section means that no knowledge has been provided to fill the knowledge base in the main agent program. Knowledge may still be available within one of the modules, though. The abscence of the beliefs or goals section simply means that the initial belief and goal bases will be empty. The abscence of an action specification section means that the agent cannot perform any other actions than those provided as built-in actions by GOAL.

The only requirement a GOAL agent program must satisfy is that it contains a *main* module *or* an *event* module. Modules contain rules for selecting actions and define a strategy or policy for handling events and for acting in an environment. These two modules are special as they are automatically entered in each *reasoning cycle* of the agent.

Each reasoning cycle, first, the event module is entered and any action rules in that module are applied. The rules in an event module are also called *event* or *percept rules*. These rules can be used to specify how percepts received from the environment should modify the agent's mental state. The function of the event module is to handle events that are received from an environment (percepts) or other agents (messages). The idea is that the state of an agent is as up-to-date as possible when it decides on performing actions and events can be used to update the agent's mental state. An agent that is connected to an environment, and should be able to perceive what happens in that environment, should have an event module. Since an agent does not need to be connected to an environment, however, an event module can be absent. The latter is rather

unusual, however, as messages are events too and these are also be handled most effectively in the event module.

After finishing the event module and updating the agent's state, in each reasoning cycle, second, the main module is entered and rules in that module are applied. The idea is that rules in this module are used to define the action selection strategy of the agent for selecting and performing actions in an environment. A well-designed agent that is connected to an environment in principle should have a main module that provides an agent with such a strategy.

The agent program will also simply be called *agent*. The term agent thus will be used both to refer to the program text itself as well as to the execution of such a program, i.e. the *running process*. This allows us to say that an agent consists of various modules (first sense) as well as that an agent performs actions (second sense). It should be clear from the context which of the two senses is intended.

An Extended Backus-Naur Form syntax definition (cf. [52]) of a GOAL agent program is provided in Table 4.1.

Remarks on Prolog Syntax
Do not use nested negations, i.e. `not(not(...))` in a program, this is bad practice and will produce an error when present in a postcondition of an action. Also do not include negations of Prolog predefined symbols such as `not(member(...))` in a postcondition of an action as this will modify the meaning of the symbol.
One additional remark needs to be made concerning the specification of the syntax of conjunctions (*poslitconj* and *litconj*) in an agent program: Certain operators require additional brackets when used in a conjunction. More specifically, the operators `,`, `;`, `->`, `:-`, `?-`, and `-->` can only be used with additional brackets.

4.2 A "Hello World" Example: The Blocks World

In order to explain how a GOAL agent works, we will design an agent that is able to effectively solve Blocks World problems. To this end, we now somewhat more precisely summarize the Blocks World domain (see also Chapter 2). The Blocks World is a simple environment that consists of a finite number of blocks that are stacked into *towers* on a table of *unlimited* size. It is assumed that each block has a unique label or name $a, b, c, \ldots$. Labelling blocks is useful because it allows us to identify a block uniquely by its name. This is much simpler than having to identify a block by means of its position with respect to other blocks, for example. Typically, labels of blocks are used to specify the current as well as goal configurations of blocks, a convention that we will also use here. Observe that in that case labels define a unique feature of each block and they cannot be used interchangeably as would have been the case if only the color of a block would be a relevant feature in any (goal) configuration. In addition, blocks need to obey the following "laws" of the Blocks World: (i) a block is either on top of another block or it is located somewhere on the table; (ii) a block can be directly on top of at most one other block; and, (iii) there is at most one block directly on top of any other block (cf. [18]).[1] Although the Blocks World domain defines a rather simple environment it is sufficiently rich to illustrate various features of GOAL and to demonstrate that GOAL allows to program simple and elegant agent programs to solve such problems.

To solve a Blocks World problem involves deciding which actions to perform to transform an initial state or configuration of towers into a goal configuration, where the exact positioning of towers on the table is irrelevant. A Blocks World problem thus defines an action selection problem which is useful to illustrate the action selection mechanism of GOAL. See Figure 2.1 for an example problem. Here we assume that the only action available to the agent is the action of moving one block that is on top of a tower onto the top of another tower or to the table (see also Chapter 3). A block on top of a tower, that is, a block without any block on top of it, is said to be *clear*. As there is always room to move a block onto the table, the table is also said to be clear.

[1] For other, somewhat more realistic presentations of this domain that consider e.g., limited table size, and varying sizes of blocks, see e.g. [29]. Note that we also abstract from the fact that a block is hold by the gripper when it is being moved; i.e., the `move` action is modeled as an instantaneous move.

program	::=	**main:** *id* **{** [**knowledge{** *kr-spec* **}**] [**beliefs{** *kr-spec* **}**] [**goals{** *poslitconj** **}**] **main module{** *module* **}** **event module{** *module* **}** [**action-spec{** *actionspec*$^+$ **}**] **}**
module	::=	[**knowledge{** *kr-spec* **}**] [**goals{** *poslitconj** **}**] **program**[*option-order*]**{** *macro** *actionrule*$^+$ **}**
		[**action-spec{** *actionspec*$^+$ **}**]
kr-spec	::=	a legal kr specification (e.g. a Prolog program)
poslitconj	::=	*atom* {**,***atom*}* **.**
litconj	::=	*literal* {**,***literal*}* **.**
literal	::=	*atom* | **not(***atom***)**
atom	::=	*predicate*[*parameters*]
parameters	::=	**(***term*{**,***term*}***)**
term	::=	a legal term in the kr language
option-order	::=	[**order=** (**linear** | **linearall** | **random** | **randomall**)]
macro	::=	**#define** id[parameters] *mentalstatecond* **.**
actionrule	::=	**if** *mentalstatecond* **then** *actioncombo* **.**
mentalstatecond	::=	*mentalliteral* { **,** mentalliteral }*
mentalliteral	::=	**true**| *mentalatom* | **not(** *mentalatom* **)**
mentalatom	::=	**bel (** *litconj* **)** | **goal (** *litconj* **)**
actionspec	::=	action **{ pre{***litconj***} post{***litconj***} }**
actioncombo	::=	action {**+** action}*
action	::=	*user-def action* | *built-in action* | *communication*
user-def action	::=	*id*[parameters]
built-in action	::=	**insert(** *litconj* **)** | **delete(** *litconj* **)** | **adopt(** *poslitconj* **)** | **drop(** *litconj* **)**
communication	::=	**send(** *id* **,** *poslitconj* **)**
id	::=	(**a..z** | **A..Z** | **_** | **$**) { (**a..z** \| **A..Z** \| **_** \| **0..9** \| **$**) }*

Boldface is used to indicate *terminal symbols*, i.e. symbols that are part of an actual program. Italic is used to indicate *nonterminal symbols*. [...] is used to indicate that ... is optional, | is used to indicate a choice, and * and $^+$ denote zero or more repetitions or one or more repetitions of a symbol, respectively. The nonterminal *kr-spec* refers to a module in a knowledge representation language (e.g. a Prolog program, where details depend on the specific Prolog system used; the current implementation of GOAL uses SWI-Prolog [58].

Table 4.1: Extended Backus Naur Syntax Definition of a GOAL Agent

Code listing

```
main: stackBuilder
{ % This agent solves the Blocks World problem of Figure 2.1.
   knowledge{
     % only blocks can be on top of another object.
     block(X) :- on(X, _).
     % a block is clear if nothing is on top of it.
     clear(X) :- block(X), not( on(_, X) ).
     % the table is always clear.
     clear(table).
     % the tower predicate holds for any stack of blocks that sits on the table.
     tower([X]) :- on(X, table).
     tower([X, Y| T]) :- on(X, Y), tower([Y| T]).
   }
   beliefs{
     on(a,b). on(b,c). on(c,table). on(d,e). on(e,table). on(f,g). on(g,table).
   }
   goals{
     on(a,e), on(b,table), on(c,table), on(d,c), on(e,b), on(f,d), on(g,table).
   }
   main module{
     program[order=random]{
           #define constructiveMove(X,Y) a-goal(tower([X,Y| T])), bel(tower([Y|T])).
       #define misplaced(X) a-goal(tower([X| T])).

       if constructiveMove(X, Y) then move(X, Y).
       if misplaced(X) then move(X, table).
     }
   }
   actionspec{
     move(X,Y) {
       pre{ clear(X), clear(Y), on(X,Z), not(X=Y) }
       post{ not(on(X,Z)), on(X,Y) }
     }
   }
}
```

Table 4.2: GOAL Agent Program for solving the Blocks World Problem of Figure 2.1.

The performance of a Blocks World agent can be measured by means of the number of moves it needs to transform an initial state or configuration into a goal state. An agent performs optimally if it is not possible to improve on the number of moves it uses to reach a goal state.[2] Some basic insights that help solving a Blocks World problem and that are used below in the design of an agent that can solve such problems are briefly introduced next. A block is said to be *in position* if the block in the current state is on top of a block or on the table and this corresponds with the goal state, and all blocks (if any) below it are also in position. A block that is not in position is said to be *misplaced*. In Figure 2.1 all blocks except block c and g are misplaced. Observe that only misplaced blocks have to be moved in order to solve a Blocks World problem. The action of moving a block is called *constructive* if in the resulting state that block is in position. It should be noted that in a Blocks World where the table has unlimited size in order to reach the goal state it is only useful to move a block onto another block if the move is constructive, that is, if the move puts the block in position. Also observe that a constructive move always decreases the number of misplaced blocks.[3] A block is said to be a *self-deadlock* if it is misplaced and above another block which it is also above in the goal state; for example, block a is a self-deadlock in Figure 2.1. The concept of self-deadlocks, also called singleton deadlocks, is important because on average nearly 40% of the blocks are self-deadlocks [56].

[2]The problem of finding a minimal number of moves to a goal state is also called the *optimal* Blocks World problem. This problem is NP-hard [29]. It is not within the scope of this chapter to discuss either the complexity or heuristics proposed to obtain near-optimal behavior in the Blocks World; see [32] for an approach to define such heuristics in GOAL.

[3]It is not always possible to make a constructive move, which explains why it is sometimes hard to solve a Blocks World problem optimally. In that case the state of the Blocks World is said to be in a *deadlock*, see [56] for a detailed explanation.

4.3 Selecting Actions: Action Rules

The **program** section in the **main module** section specifies the *strategy* used by the agent to select an action by means of *action rules*. Action rules provide a GOAL agent with the know-how about when to perform an action. An action rule provides a *reason* for performing the action. In line with the fact that GOAL agents derive their choice of action from their beliefs and goals, action rules consist of a mental state condition *msc* and an action *action* and are of the form **if** *msc* **then** *action*. The mental state condition in an action rule determines whether the corresponding action may be considered for execution or not. If (an instantiation of) a mental state condition is true, the corresponding action is said to be *applicable*. Of course, the action may only be executed if it is also *enabled*. An action is enabled if its precondition holds. The precondition of an action is evaluated on the belief base of the agent (as usual, in combination with the knowledge base of the agent). If an action is both applicable and enabled we say that it is an *option*. We also say that action rules *generate options*.

The **program** section of Table 4.2 consists of two action rules. These rules specify a simple strategy for solving a Blocks World problem. The rule

if `a-goal(tower([X,Y|T])), bel(tower([Y|T]))` **then** `move(X,Y)`

specifies that `move(X,Y)` may be considered for execution whenever `move(X,Y)` is a constructive move (cf. the discussion about the mental state condition of this rule in Chapter 2). The rule **if** `a-goal(tower([X|T]))` **then** `move(X,table)` specifies that the action `move(X,table)` of moving block `X` to the table may be considered for execution if the block is misplaced. As these are all the rules, the agent will only generate options that are either constructive moves or move misplaced blocks to the table, and the reader is invited to verify that the agent will never consider moving a block that is in position. Furthermore, observe that the mental state condition of the second rule is weaker than the first. In common expert systems terminology, the first rule *subsumes* the second as it is more specific.[4] This implies that whenever a constructive move `move(X,Y)` is an option the action `move(X,table)` is also an option.

4.4 Rule Order

Multiple action rules may generate multiple action options. Which of these action options are actually performed by the agent depends on the context. By default, action rules that appear in the **main module** are evaluated *in linear order*, that is from top to bottom as they appear in the module. The first rule that generates an option then is applied. The order of rule evaluation can be changed by adding an **order** option to the program section. For example, in Table 4.2 the order of evaluation is changed and set to *random order*. In that case, the GOAL agent will *arbitrarily* select one action out of the set of all options. As a result, such a GOAL agent is nondeterministic and may execute differently each time it is run.

By setting the order of evaluation of rules in a program section to random order, one of the action options generated by all of the rules is randomly chosen. GOAL provides an option associated with the **program** section to evaluate rules in (linear) order. Setting this option can be done simply by adding `[order=random]` after the **program** section keyword. For example, the Blocks World agent of Table 4.2 may perform either `move(a,table)`, `move(d,table)` or `move(f,table)` and when random order evaluation is applied one of these is randomly chosen.

As noted above, whenever a *constructive* move can be made in the Blocks World, it is always a good idea to prefer such moves over others. This provides one reason why it is often effective to use the default linear order of evaluation of rules. In that case, whenever a constructive move can be made the first rule will always fire and a constructive move will be performed by the agent.

The agent in Table 4.3 is a slightly modified agent compared to that of Table 4.2. Apart from the fact that this agent handles only three blocks, it also has the option `[order=linear]`.

[4] Thanks to Jörg Müller for pointing this out.

Code listing

```
main: BlocksWorldAgent
{
   knowledge{
     block(a). block(b). block(c).
     clear(table).
     clear(X) :- block(X), not(on(_,X)).
     tower([X]) :- on(X,table).
     tower([X,Y|T]) :- on(X,Y), tower([Y|T]).
   }
   beliefs{
     on(a,table). on(b,a). on(c,table).
   }
   goals{
     on(a,b), on(b,c), on(c,table).
   }
   main module{
     program[order=linear]{
       #define constructiveMove(X,Y) a-goal(tower([X,Y| T])), bel(tower([Y|T])).
       #define misplaced(X) a-goal(tower([X| T])).

       if constructiveMove(X, Y) then move(X, Y).
       if misplaced(X) then move(X, table).
     }
   }
   actionspec{
     move(X,Y) {
        pre{ clear(X), clear(Y), on(X,Z), not(X=Y) }
        post{ not(on(X,Z)), on(X,Y) }
     }
   }
}
```

Table 4.3: Blocks World Agent with Rule Order.

The behavior of this agent will be to always perform an action that is generated by the first rule if possible. As the action `move(b,c)` and action `move(b,table)` both are enabled but only the first is a constructive move and therefore an option by the first rule, the constructive action `move(b,c)` will be selected for execution. As rule order evaluation by default is linear, there is no need to use the `[order=linear]` option.

Execution Traces of The Blocks World Agent

We will trace one particular execution of the Blocks World agent of Table 4.2 in more detail here. As this GOAL agent selects an arbitrary action when there are more options available, there are multiple traces that may be generated by the agent, three of which are listed below.

In the initial state, depicted also in Figure 2.1, the agent can move each of the clear blocks `a`, `d`, and `f` to the table. Although the the applicability of mental state conditions in a rule is evaluated first, we begin by inspecting the precondition of the `move` action to check whether the action is enabled. It is easy to **verify the precondition** of the `move` action in each of these cases by instantiating the action specification of the `move` action and inspecting the knowledge and belief bases. For example, instantiating `move(X,Y)` with block `a` for variable `X` and `table` for variable `Y` gives the corresponding precondition `clear(a), clear(table), on(a,Z), not(a=table)`. By inspection of the knowledge and belief bases, it immediately follows that `clear(table)`, and we find that by instantiating variable `Z` with `b` it follows that `on(a,Z)`. Using the rule for `clear` it also follows that `clear(a)` and we conclude that action `move(a,table)` is enabled. Similar reasoning shows that the actions `move(d,table), move(f,table), move(a,d), move(a,f), move(d,a), move(d,f), move(f,d), move(f,a)` are enabled as well. The reader is invited to check that no other actions are enabled.

A GOAL agent selects an action using its action rules. In order to verify whether moving the blocks `a`, `d`, and `f` to the table are options we need to **verify applicability** of actions by checking the mental state conditions of action rules that may generate these actions. We will do so for block `a` here but the other cases are similar. Both rules in the program section of Table 4.2 can be instantiated such that the action of the rule matches with `move(a,table)`. As we know that block `a` cannot be moved constructively, however, and the mental state condition of the first rule only allows the selection of such constructive moves, this rule is not applicable. The mental state condition of the second rule expresses that a block `X` is misplaced. As block `a` clearly is misplaced, this rule is applicable. The reader is invited to verify this by checking that `a-goal([a,e,b])` holds in the initial state of the agent.

Assuming that `move(a,table)` is selected from the set of options, the action is executed by **updating the belief base** with the instantiated postcondition `not(on(a,b)), on(a,table)`. This means that the fact `on(a,b)` is removed from the belief base and `on(a,table)` is added. The goal base may need to be updated also when one of the goals has been completely achieved, which is not the case here. As in our example, we have abstracted from perceptions, there is no need to process any and we can repeat the action selection process again to select the next action.

As all blocks except for blocks `c` and `g` are misplaced, similar reasoning would result in a possible trace where consecutively `move(b,table), move(d,table), move(f,table)` are executed. At that point in time, all blocks are on the table, and the first rule of the program can be applied to build the goal configuration, e.g. by executing `move(e,b), move(a,e), move(d,c), move(f,d)`. In this particular trace the goal state would be reached after performing 8 actions.

Additionally, we list the 3 shortest traces - each including 6 actions - that can be generated by the Blocks World agent to reach the goal state:

Trace1 : move(a, table), move(b, table), move(d, c), move(f, d), move(e, b), move(a, e).
Trace2 : move(a, table), move(b, table), move(d, c), move(e, b), move(f, d), move(a, e).
Trace3 : move(a, table), move(b, table), move(d, c), move(e, b), move(a, e), move(f, d).

There are many more possible traces, e.g. by starting with moving block `f` to the table, all of which consist of more than 6 actions.

To conclude the discussion of the example Blocks World agent, in Figure 4.1 the RSG line shows the average performance of the GOAL agent of Table 4.2 that uses random rule order evaluation in terms of the number of moves relative to the number of blocks present in a Blocks World problem.

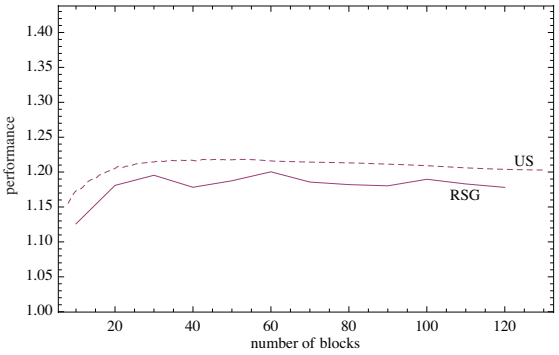

Figure 4.1: Average Performance of a Blocks World GOAL Agent

This performance is somewhat better than the performance of the simple strategy of first moving all blocks to the table and then restacking the blocks to realize the goal state indicated by the US line[5] as the GOAL agent may perform constructive moves whenever this is possible and not only after moving all blocks to the table first.

4.5 Rules with Composed Actions

It is useful to allow more than one action in the action part of a rule. GOAL supports rules with multiple actions. The action part of a rule in the **program** section may consist of any number of actions combined by means of the + operator. Such a combination of multiple actions is called a *composed action* or *combo-action*.

```
insert(on(a,b)) + delete(on(a,c))

goto(5, up) + send(elevator1, dir(up))
```

Both examples combine two actions. The first example combines the two built-in actions **insert** and **delete**. The first action is used to insert a new fact about the position of block `a` and the second action is used to remove the fact `on(a,c)` from the belief base. The second composed action consists of the environment action `goto` of the elevator environment and the built-in **send** action. The `goto` action makes the elevator move to a particular floor and the **send** action is used to communicate the choosen direction to agent `elevator1`.

The actions that are part of a composed action may be put in any order in the action part of a rule. However, the order the actions are put in is taken into account when executing the composed action: The actions are executed in the order they appear.

4.6 Notes

The GOAL programming language was first introduced in [34]. Previous work on the programming language 3APL [33] had made clear that the concept of a declarative goal, typically associated with rational agents, was still missing. Declarative goals at the time were the "missing link" needed to bridge the gap between agent programming languages and agent logics [17, 48, 37]. The programming language GOAL was intended to bridge this gap. The design of GOAL has been influenced by the abstract language UNITY [16]. It was shown how to provide a temporal

[5] Observe that this simple strategy never requires more than $2N$ moves if N is the number of blocks. The label "US" stands for "Unstack Strategy" and the label "RSG" stands for "Random Select GOAL", which refers to the default action selection mechanism used by GOAL.

verification framework for the language as well, as a first step towards bridging the gap. In [31] it has been shown that GOAL agents instantiate Intention Logic [17], relating GOAL agents to logical agents specified in this logic.

The execution mode of GOAL where action options are randomly chosen when multiple options are available is similar to the *random simulation mode* of PROMELA [9]. The *Tropism System Cognitive Architecture* designed for robot control also uses a random generator in its action selection mechanism [7]. As in *Goal*, this architecture based on so-called tropisms ("likes" and "dislikes") also may generate multiple actions for execution from which one has to be chosen. The selection of one action from the chosen set is done by the use of a biased roulette wheel. Each potential action is allocated a space (slot) on the wheel, proportional to the associated tropism value. Consequently, a random selection is made on the roulette wheel, determining the robots action.

A set of action rules is similar to a *policy*. There are two differences with standard definitions of a policy in the planning literature, however [25]. First, action rules do not need to generate options for each possible state. Second, action rules may generate *multiple* options in a particular state and do not necessarily define a function from the (mental) state of an agent to an action. In other words, a strategy of a GOAL agent defined by its action rules does not need to be *universal* and may *underspecify* the choice of action of an agent.[6]

An agent's mental state consists of its knowledge, its beliefs and its goals as explained in Section 4.2. In the current implementation of GOAL these are represented in Prolog [57, 58]. GOAL does not commit to any particular *knowledge representation technology*, however. Instead of Prolog an agent might use variants of logic programming such as Answer Set Programming (ASP; [1]), a database language such as Datalog [15], the Planning Domain Definition Language (PDDL; [25]), or other, similar such languages, or possibly even Bayesian Networks [46]. The only assumption that we will make throughout is that an agent uses a *single* knowledge representation technology to represent its knowledge, beliefs and goals. For some preliminary work on lifting this assumption, we refer the reader to [20].

GOAL does not support explicit constructs to enable the mobility of agents. The main concern in the design of the language is to provide appropriate constructs for programming rational agents whereas issues such as mobility are delegated to the middleware infrastructure layer on top of which GOAL agents are run.

The book *Multi-Agent Programming* [12] provides references to other agent programming languages such as 3APL, Jason, and Jadex, for example.

4.7 Exercises

4.7.1

Consider again the program of Figure 3.1. According to the third program rule, an agent can put a block that he is holding down on the table. This can happen even if the block could also be put in its goal position, which would be more efficient. Change the rule to ensure that a block is only put down on the table if it cannot be put in its goal position directly. (Alternatively, the order of rule evaluation can be set to `linear`; this exercise shows that in principle we can do without this option but in practice using this option is easier and helps avoiding bugs that may be introduced by adding conditions).

4.7.2

The Blocks World agent program of Figure 4.2 does not specify a *policy*. That is, the action rules do not generate an option for every possible state the agent can be in. Here we do not mean the configuration of blocks but the *mental state* of the agent. Modify the agent such that it can handle all belief and goal bases. You may assume that the knowledge base is fixed and correctly specifies the blocks present in the agent's environment.

[6] "Universal" in the sense of [51], where a *universal plan* (or policy) specifies the appropriate action for *every* possible state.

Chapter 5

Environments: Actions & Sensing

The agent of Chapter 4 has been connected to an environment called the Blocks World. The version of the Blocks World environment that we looked at so far is rather simple. By making some additional simplifying assumptions we could focus all our attention on desiging a strategy for selecting move actions. The main assumption that allowed us to do this is that the agent controlling the gripper is the only agent present that moves blocks. Because of this assumption we did not have to consider changes to the block configuration that are not controlled by the agent itself.

Environments in which such changes that are not controlled by the agent take place are said to be *dynamic* and the events that cause these changes are called *dynamic events*. We are going to look first at a simple modification of the Blocks World of Chapter 4 where *another agent* is present that is also able to control the gripper and acts so as to interfere with the goals of the agent we designed in Chapter 4. The presence of other agents provides one reason why agents need to be able to perceive the environment.

We are also going to look at a slightly more complicated version of the Blocks World where another agent (you!) again may interfere with the actions of our GOAL agent. In this version of the Blocks World dynamic events can occur and the agent has to adapt its behavior in response to such events. This dynamic version of the Blocks World will be called the Tower World. In the Tower World multiple actions are available to control the gripper and these actions *take time*. Such actions are also called *durative actions*. As we want our agent to act in real time in this case, this adds some additional complexity to the design of an agent for the Tower World.

5.1 Environments and Agents

By *connecting* an agent to an environment it may gain *control* over (part of) that environment. An agent is connected to an *entity* present in an environment. In the Blocks World the agent is connected to the gripper, and in a gaming environment such as Unreal Tournament an agent is connected to a bot. An agent may also be connected to physical entities such as a robot in the real world. The agent in all of these cases is the *mind* of the entity and the entity itself can be viewed as the *body* of the agent. By connecting to an entity an agent in a sense gets the capabilities of the entity and is able to perceive what the entity is able to see in the environment.

Interfaces to environments and connections of agents with entities in an environment are established by means of a mas file. A mas file consists of three sections, see Table 5.1. The first section is the *environment section*. In this section a reference to a jar file must be provided that enables GOAL to load the interface to the environment. GOAL uses and requires a specific type of interface called the Environment Interface Standard (EIS) to connect to an environment. More details on this interface can be found [5, 4]. The environment interface is loaded automatically when the multi-agent system is launched. In the environment section it is often also possible to specify additional *initialization parameters* of the environment by means of the **init** command. In

Table 5.1, for example, the initial configuration of blocks is loaded from a text file.

Agents are constructed from GOAL agent files which are listed in the *agent files section*. An agent is connected to an entity by means of a *launch rule* in the mas file. Launch rules specify a launch policy for connecting agents to entities in the environment. A simple and often used example of such a launch rule is provided in Table 5.1. This rule is of the form

```
when entity@env do launch <agentName>:<agentFilename>.
```

This rule specifies that an agent is launched for any entity that becomes available in the environment. Launching an agent means creating it from the referenced `agentFilename` and naming it `agentName`. Examples of other types of launch rules are provided in Chapter 6.

```
1  environment{
2    "blocksworld.jar".
3
4    init[configuration="bwconfigEx1.txt"].
5  }
6
7  agentfiles{
8    "stackBuilder.goal".
9  }
10
11 launchpolicy{
12   when entity@env do launch stackbuilder:stackBuilder.
13 }
```

Figure 5.1: A mas file that connects an agent to the Blocks World.

We say that agents are connected to an environment and to an entity rather than *being part* of an environment. We use this terminology to emphasize the *interface* that is present between the agent and the environment. The interface between the agent and the environment is used to *send* commands to the entity in order to perform actions in the environment and to *receive* percepts through the interface in order to observe (parts of) that environment.

We cannot emphasize enough how important it is while designing an agent and a multi-agent system to investigate and gain a proper understanding of the environment to which agents are connected. As a rule of thumb, *the design of a multi-agent system should always start with an analysis of the environment in which the agents act.* It is very important to first understand which aspects are most important in an environment, which parts of an environment can be controlled by agents, and which observations agents can make in an environment. Assuming that an environment interface is present, moreover, two things are already provided that must be used to structure the design of agents. First, a basic vocabulary for representing the environment is provided in terms of a language used to represent and provide percepts to an agent. Second, an interface provides a list of actions that are available to an agent, which, if they are documented well, also come with at least an informal specification of their preconditions and effects. It is only natural to start analyzing this interface, the percepts and actions it provides, and to incorporate this information into the design of an agent program. More concretely, as a rule it is best to start developing an agent by *writing percept rules and action specifications* for the agent.

An agent does *not* need to be connected to an environment and an environment section does not need to be present in a mas file. An agent that is not connected to an environment may perform useful computations like any other type of program can do. The main difference, as noted in Chapter 4, is that agent programming is *programming with mental states*. Note that agents which are not connected to an environment may be part of a larger multi-agent system and can perform useful functions by communicating with other agents that *are* connected to an environment. An agent, for example, might be used to maintain a central point of contact and to collect all available information about the environment from all agents that are connected to

that environment. Such an agent that maintains a more global view can be used to manage and instruct other agents what to do.

5.2 The Blocks World with Two Agents

It is possible to connect multiple agents to one entity and we will exploit this possibility to introduce the need for sensing an environment. We will connect the *stackBuilder* Blocks World agent of Chapter 4 to the gripper in the Blocks World as well as another agent called *tableAgent*.

In order to connect two (or more) agents to an entity we need to slightly change the mas file of Figure 5.1. There is no need to change the environment section; we still want to connect to the Blocks World environment. The first change we need to make is that the agent file for the *tableAgent* needs to be added to the agent files section. The second change we need to make in order to connect two agents to the gripper is that we modify the launch rule for connecting to the gripper entity when it becomes available. We simply add the *tableAgent* to the list of agents that needs to be connected to the gripper, see Figure 5.2 line 9.

```
1  ...
2
3  agentfiles{
4    "stackBuilder.goal".
5    "tableAgent.goal".
6  }
7
8  launchpolicy{
9    when entity@env do launch stackbuilder:stackBuilder, tableagent:tableAgent .
10 }
```

Figure 5.2: A mas file that connects two agents to the gripper in the Blocks World.

The code of the *tableAgent* is provided in Figure 5.3. The main goal of this agent, different from the *stackBuilder* agent, is to put all blocks on the table. The agent is quite simple. The agent program contains only some of the knowledge of the *stackBuilder* agent, has no initial beliefs or goals, and has a few very simple action rules in its **main module**. Compared to the *stackBuilder* agent is has an additional action specification for the action `skip`. It also has an additional module called **event module**. The **event module** is used for handling events and percepts and will be explained shortly.

Because both agents are connected to the gripper in the environment, both agents are able to move blocks. Different from the *stackBuilder* agent the *tableAgent* only moves blocks to the table; see Figure 5.3 line 10. The *tableAgent*, however, does not always move a block but may also sometimes perform the `skip` action; see line 11 in Figure 5.3. The `skip` action is always enabled and changes nothing.[1] The rules are evaluated in random order and therefore the agent will, if a block can be moved to the table, with equal probability perform either a move of a block to the table of the skip action.

We will see that if we give both agents control over the gripper that there is a need to sense the environment in order to achieve an agent's goals. The first thing to note here is that if two agents are connected to the same gripper the agents no longer have *full control* over the things that happen in the Blocks World environment. While designing our first Blocks World agent we implicitly made this assumption. We also assumed we knew the complete initial configuration of the environment and what changes the `move` action brings about in the Blocks World. As a result, we could correctly specify the inital beliefs of the agent and provide an action specification that

[1] Below we will see yet another action named `nil` that is available in the Tower World and has the same pre- and post-condition as the `skip` action here. There is still a major difference between the two actions: the `skip` action is unknown in the environment and has no effect in the environment itself, whereas, as we shall see, the `nil` action *does* have an effect in the Tower World.

```
1  main: tableAgent
2  { % This agent moves blocks to the table while the goal configuration is not achieved.
3     knowledge{
4       block(X) :- on(X, _).
5       clear(table).
6       clear(X) :- block(X), not(on(_,X)).
7     }
8     main module{
9       program[order=random]{
10        if bel(on(X,Y), not(Y=table)) then move(X,table).
11        if true then skip.
12      }
13    }
14    event module{
15      program{
16        forall bel( percept( on(X,Y) ), not( on(X,Y) ) ) do insert( on(X,Y) ).
17        forall bel( on(X,Y), not( percept( on(X,Y) ) ) ) do delete( on(X,Y) ).
18      }
19    }
20    actionspec{
21      move(X,Y) {
22         pre{ clear(X), clear(Y), on(X,Z), not(X=Y) }
23         post{ not(on(X,Z)), on(X,Y) }
24      }
25      skip {
26         pre{ true }
27         post{ true }
28      }
29    }
30 }
```

Figure 5.3: GOAL Agent That Moves Blocks to the Table

exactly matches the effects of a move of a block in the environment. Together with the full control assumption the design of an agent then is greatly simplified. In fact, if an agent has full control, knows the initial configuration completely, and knows what the effects of actions are, there is no need to observe the environment. An agent knows in that case how a configuration of blocks is changed and what the resulting configuration is after performing an action. Of course the agent still needs to do something to maintain a correct representation of its environment: it must update its belief base accordingly. Note that in this case the Blocks World environment is little more than a graphical display of the configuration of blocks in this world. And, as a side remark, also note that the assumption of full control is very useful when it comes to planning ahead: a planner needs to be able to compute configurations that result from actions and this is easier when full control is assumed as we just have argued.

All this changes when full control is lost. When another agent may also change the Blocks World configuration, an agent can no longer derive a correct representation of the state of its environment from an initial configuration and the actions that the agent itself performed. In order to maintain an accurate representation of the environment, an agent will need to be able to *perceive* what changes have taken place because of events that are not under its control.

5.2.1 Processing Percepts and the Event Module

In GOAL, sensing is not represented as an explicit act of the agent but a *perceptual interface* is assumed to be present between the agent and the environment. This interface specifies which percepts an agent will receive from the environment. The percept interface is part of the environment interface to connect GOAL agents to an environment [5, 4]. In general, a GOAL agent does not need to actively perform sense actions, except when the environment makes actions to observe the environment explicitly available to an agent. The general approach for handling percepts built into GOAL is that each time just before a GOAL agent will select an action to perform, the agent

processes *percepts* it receives through its perceptual interface.[2] The steps that are performed if an agent is connected to an environment are:

1. Clear the percept base by removing percepts received in the previous cycle,
2. Check whether any new percepts have been received from the environment,
3. Insert the new percepts into the percept base,
4. Process received percepts and update the mental state of the agent accordingly,
5. Continue and decide which action to perform next.

The Blocks World environment provides a percept interface that returns percepts of the form `on(X,Y)` which indicate which block is on top of another block or on the table. For each instantiation of `on(X,Y)` that holds in the Blocks World environment a percept is generated. In other words, the percepts in the Blocks World provide accurate and complete information about the state of the Blocks World: If a block `X` is directly on top of another block `Y`, a percept `on(X,Y)` will be received, and only then. Each time percepts are received, moreover, this complete set of information is provided.

The percepts that are received from the Blocks World environment are stored in the *percept base* of the agent. It is assumed that percepts are represented in the knowledge representation language that is used by the GOAL agent. A special, reserved predicate `percept/1` is used to differentiate what we will call *perceptual facts* from other facts in the knowledge or belief base. For example, a percept received from the environment that block `a` is on block `b` is inserted into the percept base as the fact `percept(on(a,b))`.

Percepts represent "raw data" received from the environment that the agent is operating in. For several reasons an agent cannot simply add the new information to its belief base.[3] One reason is that the received information may be inconsistent with the information currently stored in the belief base of the agent. For example, receiving information `on(a,table)` that block `a` is on the table may conflict with a fact `on(a,b)` stored in the belief base that block `a` is on block `b`. In order to correctly update the belief base, the fact `on(a,b)` needs to be removed and the fact `on(a,table)` needs to be added to the belief base. A more pragmatic reason is that the perceptual facts received may represent signals which need interpretation and it is more useful to insert this interpretation into the belief base than the fact itself. For example, a percept indicating that the agent bumped against a wall may be interpreted as a failure to move forward and it is more useful to make sure that the fact that the agent is still located at its last known position is present in the belief base than storing the fact that the agent bumped into a wall. Such percepts may be received by the agent in the Wumpus World environment.

The **event module** of an agent is executed in step 4 to process the percepts received from the environment. The processing of percepts is achieved by applying *event* or *percept rules* that are present in this module. These rules make it possible to modify the mental state of the agent in response to receiving particular percepts such as the bump percept in the Wumpus World. By being able to explicitly specify such rules to handle percepts, it is possible to generate more sophisticated responses than just inserting the information received into the belief base of an agent. The agent program listed in Figure 5.3, see lines 14-19, contains an **event module** with two event rules in the corresponding **program** section.

Event rules are action rules and are only named so because of their *function* to handle events. An event rule has the same form as any other rule and can have one out of three forms:

```
if <mental_state_condition> then <action> .
forall <mental_state_condition> do <action> .
listall <Listvar> <- <mental_state_condition> do <action> .
```

[2] GOAL agents perform a so-called *sense-plan-act* cycle. This cycle is also called the *reasoning cycle* of an agent.

[3] Recall that the knowledge base is *static* and cannot be modified. As the belief base is used to represent the current state of an environment, percepts might be used to directly modify the belief base of the agent.

Typically the second type of rule is used in the **event module** for handling percepts. The reason is that we usually want to handle *all* percepts of a particular form and perform an update for each of these percepts. In the Blocks World, for example, for each block `X` in the environment a percept `on(X,Y)` for some `Y` is generated. An agent may, for example, receive the percepts `on(a,b)`, `on(b,c)`, and `on(c,table)`. In order to process all of these percepts and insert these facts into the belief base it must be able to do something for each of the individual percepts.

Rules of the form **`if...then...`** cannot be used for this purpose. These rules are only fired for one out of all possible instantiations. In contrast, rules of the form:

```
forall <mental_state_condition> do <action> .
```

perform the corresponding action for each true instantiation of the condition.

For this reason, such rules are used in the **event**, **module** for the *tableAgent*. When this agent receives percepts `on(a,b)`, `on(b,c)`, and `on(c,table)` and believes that `on(a,c)`, `on(c,b)`, and `on(b,table)`, both rules in the **event**, **module** of *tableAgent* will be applied three times. As a result, each of the perceptual facts `on(a,b)`, `on(b,c)`, and `on(c,table)` will be inserted into the agent's belief base (by the first rule) and the facts `on(a,c)`, `on(c,b)`, and `on(b,table)` that it initially believed will be removed from the agent's belief base (by the second rule). This is exactly what we want and why we need to use the **`forall...do...`** rules instead of rules of the form **`if...then...`**.

Apart from the fact that the **event module** is executed upon receiving percepts from the environment, there are other reasons for putting rules for handling percepts in this module and not in the **main module**. There are a number of differences between the **main module** and the **event module**. These differences derive from the different functions associated with these modules. Whereas the main function of the **main module**is to specify an action selection strategy *for performing actions in an environment*, the main function of the **event module** is to *process events and update the agent's mental state accordingly.*

In order to perform this function, rules in the **event module** can access the content in the percept base of the agent. This can be done in the mental state condition of event rules by means of special literals of the form `percept(`φ`)` which are used to inspect the percept base. These literals should be used in combination with the **bel** operator; see Figure 5.3 for two examples. Rules that use the special keyword `percept` are also called *percept rules*. Whether a percept rule is applicable is verified by inspecting the knowledge, belief and percept base. If the percept rule's condition can be derived from the combination of these three databases, the rule is applicable.

A second difference between the **main module** and **event module** is that *all* rules that occur in the **event module** are evaluated and applied in order. In the event module, each rule thus will be evaluated. This facilitates the processing of percepts and events as an agent will want to process *all* events that contain potential information for making the right action choice. This type of rule evaluation ensures that the newly received information of blocks is inserted into the belief base of the *tableAgent* (by the first percept rule) *and* the old information is removed (by the second percept rule).

5.2.2 Multiple Agents Acting in the Blocks World

If both the *stackBuilder* agent and the *tableAgent* get (partial) control over the gripper in the Blocks World, both agents need to be able to perceive changes in the configuration of blocks. The *stackBuilder* agent thus also will need to be extended with an **event module** for handling percepts. By adding exactly the same **event module** as that of the *tableAgent* this is easily achieved.

There is a second change to the *stackBuilder* agent that we should make to the agent: We should remove its initial belief base. By adding the **event module** of the *tableAgent* to the *stackBuilder* agent this agent now also is able to perceive the configuration and there is no need anymore to insert the initial configuration into the belief base of the agent. In general we do not

```
main: stackBuilder
{ % This agent solves the Blocks World problem of Figure 2.1.
   knowledge{
     % only blocks can be on top of another object.
     block(X) :- on(X, _).
     % a block is clear if nothing is on top of it.
     clear(X) :- block(X), not( on(_, X) ).
     % the table is always clear.
     clear(table).
     % the tower predicate holds for any stack of blocks that sits on the table.
     tower([X]) :- on(X, table).
     tower([X,Y| T]) :- on(X,Y), tower([Y|T]).
   }
   goals{
     on(a,e), on(b,table), on(c,table), on(d,c), on(e,b), on(f,d), on(g,table).
   }
   main module{
     program{
           #define constructiveMove(X,Y) a-goal(tower([X,Y|T])), bel(tower([Y|T])).
       #define misplaced(X) a-goal(tower([X| T])).

       if constructiveMove(X, Y) then move(X, Y).
       if misplaced(X) then move(X, table).
     }
   }
   event module{
     program{
       forall bel( percept( on(X,Y) ), not( on(X,Y) ) ) do insert( on(X,Y) ).
       forall bel( on(X,Y), not( percept( on(X,Y) ) ) ) do delete( on(X,Y) ).
     }
   }
   actionspec{
     move(X,Y) {
       pre{ clear(X), clear(Y), on(X,Z), not(X=Y) }
       post{ not(on(X,Z)), on(X,Y) }
     }
   }
}
```

Figure 5.4: GOAL Agent Program for solving Blocks World Problems.

know what the initial configuration is and even if we would know, because the other agent may be the first to modify it, such an initial belief base would immediately be invalidated. By adding the **event module** and by removing the belief base we make the agent more generic in a sense. The code of the resulting agent is listed in Figure 5.4.

After making these changes, what happens when both agents are connected to the Blocks World environment? Each of these agents will start moving blocks. Performing a `move` action does not take time but is *instantenous*. Because the `move` action in the Blocks World is *instantaneous*, each time an action is performed in the environment it will succeed. This means that the *stackBuilder* agent will perform moves to achieve its goal as before. This case, however, the *tableAgent* may perform actions that interfere with that goal by moving blocks to the table. A tower of blocks that is being built by the *stackBuilder* agent may thus be broken down again by the *tableAgent*.

Will the *stackBuilder* agent still be able to achieve its goal? One important question relevant to answering this question is whether the agents are each treated *fair*. That is, do the agents get to perform the same number of actions each over time? Alternatively, we might assume that the agents are taking turns when performing actions. Answering the question of whether the *stackBuilder* agent will be able to achieve its goal is left as an exercise; see Exercise 5.9.

5.3 The Tower World

The Tower World (see Figure 5.5) that we introduce now is a variant of the simple Blocks World that we have used so far in which moving the gripper, and therefore moving blocks, takes time. This environment, moreover, allows a user (you!) to drop and drag blocks using a mouse at will.

This introduces some new challenges that our previous Blocks World agents are not able to handle. The basic setup of the Tower World environment, however, is very similar to the Blocks World environment. All blocks have equal size, at most one block can be directly on top of another, and the gripper available can hold at most one block at a time.

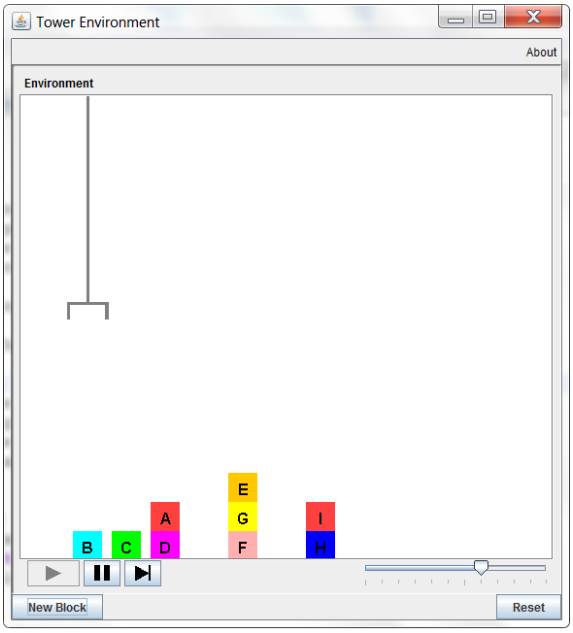

Figure 5.5: The Tower World Interface

Of course, because an agent connected to the gripper in the Tower World does not have full control it will need to be able to observe changes in the Tower World environment similar to the two competing Blocks World agents we discussed above. In the Tower World, however, there are additional reasons why being able to sense the environment is important. One of these is that the completion of actions takes time.

5.3.1 Specifying Durative Actions

Actions that take time are called *durative actions*. Because they take time these actions are not immediately completed as the *instantaneous* `move` action in the Blocks World is. The effects of instantaneous actions are realized immediately upon performing the action. The effects of durative actions, however, are established only after some time has passed. Moreover, these effects are not certain because of other events that may take place while the action is performed. A user may move blocks and insert them at arbitrary places back into the configuration of blocks while the agent tries to do so as well. A user may also remove a block from or put a block in the gripper. Because the agent cannot be sure of pretty much anything anymore there is a need to be able to *monitor the progress* of an action. While performing an action of picking up a block, for example, the agent needs to monitor whether it is still feasible to pick up the block while moving the gripper into a position such that it can pick up and grab the block. If the block becomes obstructed in some way or is moved to another position, moving the gripper to a particular location may no longer may sense. In such cases the agent should *reconsider* its actions and possibly its goals.

A GOAL agent is able to monitor progress of actions. A GOAL agent does not block on an action that it sends to the environment (that is, at least, if the environment itself does not force blocking in some way). While the action is being performed by its body (an entity such as the gripper in the Tower World), the agent can monitor the progress of the action by perceiving the environment.

All actions available in the Tower World are durative. In line with the design advice above, we will start with providing the action specifications for each of the three actions that can be performed. Because these actions are durative and GOAL does not block on these actions, there is a problem that we need to address. It seems that we cannot provide adequate postconditions for such actions. We cannot specify as postcondition of the action of picking up a block that the gripper thereafter holds that block. The point is that the agent then would *immediately* start to believe that the gripper holds the block while the action of picking up the block has only just started. At that time, obviously, such a belief would be simply false. It takes time to pick up the block and only if the action is successfull will this effect be realized. How do we solve this problem?

The solution that we provide is a simple one. Because we do not know and cannot know at the time of starting to pick up a block what the results of the action will be, we simply provide an *empty* postcondition. The idea is that the agent will learn over time what the effects of its (durative) actions are and by monitoring their progress will become aware of the resulting changes in the environment. Effects of durative actions thus are not specified in an action specification but need to be observed by perceiving what happens in the environment.

We are still able to provide adequate preconditions for actions. At the time of initiating an action it is possible to inspect whether necessary conditions for performing the action hold. It thus makes sense, assuming that an agent maintains a reasonably accurate representation of its environment, to add and verify such preconditions. Doing so, as before, moreover provides a clear and reusable specification of actions in the environment and thereby enhances understanding of the agent program.

The Tower World environment provides three actions instead of the one `move` action in the Blocks World. These actions include an action `pickup` for picking up a block with the gripper, an action `putdown` for putting down one block on a particular other block, and a special `nil` action that puts the gripper back in the left-upper most corner of the graphical display of the environment. The `pickup` action has one parameter, the `putdown` action has two, and the `nil` action has no parameters.

In order to provide adequate action specifications, it is always very important to know what kind of percepts are provided to agents when they are connected to an environment. In the Tower World, three kinds of percepts can be received:

```
block(X)
on(X,Y)
holding(X)
```

These three predicates can, as usual, be used to specify actions. There are two conditions that need to be satisfied in order to be able to perform a `pickup` action: the block that needs to be picked up needs to be clear, and the gripper cannot hold a block already. Using the definition of the `clear/1` predicate that we introduced in Chapter 2 we then can provide the following action specification for `pickup`:

```
pickup(X) {
  pre{ clear(X), not(holding(_)) }
  post{ true }
}
```

Although the postcondition could have been left empty in GOAL and we could have simply written **post**{ }, we prefer not to do so. Instead we have provided **true** as postcondition to indicate that we have not simply forgotten to provide a specification of the postcondition but that we explicitly provide an "empty" postcondition for the action.

The action specification for `putdown` is similarly straightforward:

```
putdown(X,Y) {
  pre{ holding(X), clear(Y) }
```

```
    post{ true }
  }
```

The third action specification for the `nil` action is even more straightforward. This action can always be performed and has **true** as precondition. As before, we also have **true** as postcondition.

```
nil {
  pre{ true }
  post{ true }
}
```

The action `nil` is somewhat unusual and rather different from the actions that we have seen before in the context of Blocks World-like environments. This action puts the gripper at a particular position in the graphical interface provided with the Tower World environment (see Figure 5.5). As such, the action is of little interest with respect to our main goal: creating particular block configurations. Also note that the predicates in terms of which percepts are provided by the environment to the agent are unrelated to the exact position of the gripper. The idea is that when an agent has nothing to do anymore to achieve its goals we put the gripper back into its initial starting position. This then will provide an indication that the agent is ready.

5.3.2 Percepts in the Tower World

The percepts that an agent receives from the Tower World environment are exactly the same predicates `block` and `on` we have seen before in the simple Blocks World environment except for one additional new predicate `holding`. The `holding` predicate has one parameter and indicates which block the gripper is holding, if any.

As before we may also assume *full observability*: Every time percepts are received they provide *correct and complete* information about the state, i.e. the configuration of blocks. Given this assumption, because we know that the gripper is either holding a block or it is not, the Closed World assumption can be applied to the predicate `holding`. That is, if the agent does not believe it (the gripper) is holding a block, we can then derive that the gripper is holding no block. As a result, we can reuse the exact same percept rules for the Blocks World introduced above and we add one pair of rules for the predicate `holding` that follows the exact same pattern discussed above. We thus obtain the following code for the **event module**:

```
event module{
  program{
    % assumes full observability
    forall bel( block(X), not(percept(block(X))) ) do delete( block(X) ).
    forall bel( percept(block(X)), not(block(X)) ) do insert( block(X) ).

    forall bel( on(X,Y), not(percept(on(X,Y))) ) do delete( on(X,Y) ).
        forall bel( percept(on(X,Y)), not(on(X,Y)) ) do insert( on(X,Y) ).

        forall bel( holding(X), not(percept(holding(X))) ) do delete( holding(X) ).
        forall bel( percept(holding(X)), not(holding(X)) ) do insert( holding(X) ).
  }
}
```

5.4 Performing Durative Actions in Environments

We have seen in Section 5.3.1 how to specify durative actions. We will discuss other aspects of instructing an entity in an environment by a GOAL agent to perform an action here.

An important question is what happens when a durative action has been initiated and is in progress and another action is send to the environment. Recall that a GOAL agent does not block on a durative action until it completes, and should not do so, because it is important to *monitor the progress* of the action. If, for example, in the Tower World the action `putdown(X,Y)` is not

feasible anymore because a user has put another block than `X` on top of block `Y`, it is important to be able to interrupt and terminate the action `putdown(X,Y)` and initiate another, feasible action instead. In this particular case, it would make sense to put block `X` on the table instead by means of performing the action `putdown(X,table)`, for example. In the Tower World, in case an action is in progress, performing another action *overwrites* the first action and immediately continues with performing the new action.

One example of overwriting in particular is worth mentioning: actions typically overwrite actions of the *same type*. As a rule, actions are of the same type if the action name is the same. The overwriting of the `putdown(X,Y)` where `Y`$\neq$`table` by the action `putdown(X,table)` provides an example. Another example is provided by the `goto` action in the UNREAL TOURNAMENT environment. If this action is performed continuously (the agent very quickly instructs a bot to perform this action) with different parameters, the bot being instructed to do so actually comes to a halt. There is simply no way to comply with such a quickly executed set of instructions for the bot. Finally, of course, you would expect that nothing changes if an agent performs *exactly the same action* again while already performing that action...

It should be noted, however, that different things than overwriting a durative action in progress might happen in other environments with other actions. Less sensible and less frequent possibilities include *queuing* actions that are being sent by the agent and executing them in order and simply *ignoring* new actions that are requested by an agent while another action is still ongoing.

An important alternative, however, which often makes sense is to perform actions *in parallel*. Examples where this is the more reasonable approach abound. In UNREAL TOURNAMENT, for example, a bot should be able to and can move *and* shoot at the same time. A robot should be able to and usually can move, perceive *and* speak at the same time. Of course, you would also expect a gripper to maintain grip on a block it holds while moving...

5.5 Action Selection and Durative Actions

We have gained more insight in how durative actions are handled by entities in an environment. We now need to consider how a GOAL agent should deal with durative actions while selecting actions to perform.

Because durative actions may overwrite each other, it becomes important that an agent continues to perform the same action as long as it has not been completed yet. Otherwise an agent would come to a halt as the bot that is continuously instructed to go in different directions. This requires some kind of focus of the agent. This is a potential problem for a GOAL agent as the choice of action may be unspecified to some extent by the use of action rules **`if...then...`**. At the same time, we also need to take into account that an agent should *reconsider* its choice of action if it is no longer feasible.

Since in the Tower World all actions overwrite each other because a gripper can do at most one thing at the same time, we will continue the design of an agent for this environment again. We will first illustrate the problem of underspecification introduced by action rules. After arguing that some kind of *focus* is needed we will then continue and develop an adequate action selection strategy for our Tower World agent that is able to reconsider its action choices.

5.5.1 Action Selection and Focus

The issue with underspecification can be illustrated by looking again at the action rules that were used in the Blocks World agent of Chapter 4, see Table 4.3. For ease of reference, we repeat the **main module** section of that agent here.[4]

```
main module{
  program[order=linear]{
    #define constructiveMove(X,Y) a-goal(tower([X,Y| T])), bel(tower([Y|T])).
```

[4]Note that the definition of the macro misplaced has been slightly modified to be able to also handle obstructing blocks.

```
    #define misplaced(X) a-goal( on(Y,Z) ), bel( above(X,Z); above(X,Y) ).

    if constructiveMove(X, Y) then move(X, Y).
    if misplaced(X) then move(X, table).
  }
}
```

These rules provide an adequate action selection strategy for the Blocks World agent because the `move` action is instantaneous. Of course, in the Tower World an agent cannot perform *move* actions but has to use combinations of `pickup` and `putdown` actions to simulate that action. But let's assume for the moment that an agent can perform such actions but that they would take time, i.e. assume that the action `move` is *durative*.

It then becomes important to note that the rules underspecify the selection of an action even though the linear order of rule evaluation is used. The use of linear order implies here that if a constructive move can be made it will be made, that is, the first rule will be applied whenever that is possible. In case there are multiple constructive moves that can be made, however, the first rule does not specify *which* of these constructive moves should be selected. A concrete example is provided by an agent that has as beliefs `on(a,table)`, `on(b,table)`, `on(c,table)`, and `on(d,table)` and as goal `on(a,b), on(b,table), on(c,d), on(d,table)`. This agent, if it would use the rules above, could perform two constructive moves: `move(a,b)` and `move(c,d)`. It would select either one of these at random. In fact, the agent thus is and in the Blocks World should be completely indifferent which of these moves would be selected. Any constructive move that can be made instantaneously is as good as any other. It is one of the distinguishing features of GOAL that no such specification is necessary either.

If the move action would be durative, however, we cannot be so indifferent anymore. The point is that by continuously selecting a different constructive move the agent would come to a halt and do nothing sensible anymore because these actions would overwrite each other. In this case, the agent thus should maintain some kind of *focus* and continuously select one and the same action to avoid overwriting actions.[5]

In the Tower World there is an elegant solution for creating this focus on a particular action that corresponds with a simple fact of this environment: *the gripper can hold at most one block at a time*. By focusing on one block to hold, and if holding it, focusing on the place where to put it down, the agent would create the needed focus. In that case, any actions that are unrelated to the block that is being focused on would be disregarded. How can we implement this idea in a GOAL agent without making this agent too rigid? That is, how can we implement this idea while making it possible to easily change focus if events in the Tower World make that necessary?

The general strategy for creating focus is simple: *Introduce a goal that indicates the current focus*. Goals are easily reconsidered and dropped if needed in GOAL which addresses our concern that sometimes the focus on an action needs to be reconsidered. In the Tower World environment this general idea can be implemented by adopting a goal to hold a block. This goal is the perfect candidate for creating the right kind of focus in this environment. Of course, in order to maintain focus we then also need to ensure that we only have a *single* goal to hold one particular block but no others. More concretely, we should have at most one goal of the form `holding(X)` in the agent's goal base. We call such goals *single instance goals*.

The idea of single instance goals (sig) to create focus is implemented very easily in GOAL by means of a simple pattern:

```
if not(goal(<sig>)), <reason_for_adopting(sig)> then adopt(<sig>) .
```

In the Tower World environment, there are two good reasons for adopting a goal to move a block (and thus to hold it). The first is that a constructive move can be made by moving the

[5] Another solution that seems available is to not perform any other actions until a previously initiated action has completed or needs to be interrupted for some reason. We then would have to implement a kind of queuing mechanism in the GOAL agent itself and keep track of whether an action has been completed or not. Although this may be feasible the code needed to achieve this most likely will only provide a rather ad hoc solution for the problem. The solution proposed in the main text appears to be much more elegant.

block, and the second is that the block is misplaced.[6] We thus obtain two rules by instantiating the above pattern where the first rule (making a constructive move) should be preferred over the second.

```
if not(goal(holding(Z)))), constructiveMove(X,Y) then adopt(holding(X)) .
if not(goal(holding(Z)))), misplaced(X) then adopt(holding(X)) .
```

Because we now have two rules for adopting a single instance goal it also is important to make sure that at most one is applied at any time. That is, the rules should be evaluated in linear order to prevent adoption of two goals as we then no longer would have a *single instance* goal. We will see one way to do that below and another, more advanced method for ensuring this by means of modules in Chapter **??**.

5.5.2 Action Selection in the Tower World

In the remainder of this section, we complete the design of our agent that is able to robustly act in the dynamic Tower World environment. Given that we may assume that we have implemented the single instance pattern above and always will have at most a single goal of holding a block at any time, the selection of actions to be performed in the environment turns out to be very simple. The more difficult part that we also still need to implement is the strategy for reconsidering the goal to hold a particular block if unexpected events happen (i.e. a user moves blocks) which make such a goal no longer beneficial.

The selection of actions to perform in the environment, assuming that a goal to hold a block is present if we are not yet holding a block and we only have feasible goals in our goal base, is very straightforward. If we want to hold a block, we simply pick it up. If we are holding a block, then put it in position whenever possible, and otherwise put it somewhere on the table. Otherwise, we do nothing, i.e. put the gripper back in its initial position. Implementing this strategy in GOAL is just as straightforward. Because the rules for adopting single instance goals must be evaluated in linear order, the rules above for doing so have also been included in the **main module** section below. (Recall that the **main module** evaluates rules in linear order.) These rules need to be placed first in the module. (Why? *Hint*: Think about what would happen if no goal to hold a block is present and these rules would have been placed at the end of the **program** section.)

```
main module{
  program{
    #define constructiveMove(X, Y) a-goal( tower([X,Y|T]) ),
             bel( tower([Y|T]), clear(Y), (clear(X) ; holding(X)) ) .
    #define misplaced(X) a-goal( on(Y, Z) ), bel( above(X, Z); above(X, Y) ) .

    if not(goal(holding(Z)))), constructiveMove(X,Y) then adopt(holding(X)) .
    if not(goal(holding(Z)))), misplaced(X) then adopt(holding(X)) .

    if a-goal( holding(X) ) then pickup(X) .

    if bel( holding(X) ) then {
       if constructiveMove(X,Y) then putdown(X, Y) .
       if true then putdown(X, table) .
    }

    if true then nil .
  }
}
```

The final part that we need to design is the strategy for *reconsidering* a goal to hold a block. As discussed above, this may be necessary when a user moves one or more blocks and the current focus is no longer feasible or no longer desirable. When is this the case? Assuming that we want

[6] In this context, note that misplaced means that a block obstructs access to another block that can be put into position or no block can be put into position and at least one block needs to be moved to the table. See also Section 5.5.1 for a definition of the macro `misplaced`.

to hold block X, i.e. holding(X) is present in the agent's goal base, when should we reconsider this goal? Two reasons for dropping the goal are based on different cases where it is no longer feasible to pick up the block. The first case is when the block is no longer clear. The second case is when another block is being held (because the user put a block into the gripper). A third reason for dropping the goal is because the user has been helping the agent and put the block into position. There would be no reason anymore for picking up the block in that case. These three reasons for dropping the goal would already be sufficient for operating robustly in the dynamic Tower World environment but would not yet always generate a *best response* to changes in the environment. A fourth reason for reconsidering the goal is that the target block cannot be put into position but there is another goal that can be put into position. By changing the focus to such a goal the behavior of the agent will be more goal-oriented.

```
% check for reasons to drop or adopt a goal (goal management).
if goal( holding(X) ) then {
  % first reason: cannot pick up block X because it's not clear.
  if not(bel( clear(X) )) then drop( holding(X) ) .
  % second reason: cannot pick up block X because now holding other block!
  if bel( holding(_) ) then drop( holding(X) ) .
  % third reason: block X is already in position, don't touch it.
  if inPosition( X ) then drop( holding(X) ) .
  % fourth reason: we can do better by moving another block constructively.
  listall L <- constructiveMove(Y,Z) do {
        if bel(not(L=[]), not(member([X,_,_],L))) then drop( holding(X) ) .
  }
}
```

Where should we put these rules for dropping a goal in the agent program? Should they be added to the **main module** or rather put in the **event module**? The agent should drop a goal if any of the reasons for doing so apply. If one of the cases applies there is already sufficient reason to remove the goal to hold a particular block. We thus need to make sure that *all* rules are evaluated and applied if they are applicable. Because all rules that are put into the **event module** are always evaluated it is natural to put the rules for dropping the goal into the **event module**. There is a second reason to put these rules in the **event module**: in principle it is best to first update the mental state of the agent and only after doing so deciding on an action to perform in the environment. Since the rules for dropping a goal only affect the mental state of the agent based on this design principle they should also be put in the **event module** rather than in the **main module**. The **main module** is always executed after the **event module** has been executed and therefore is the more logical place to put action rules for selecting environment actions as we have done above.

For technical reasons, i.e. the different styles of rule evaluation in the **main module** and the **event module**, we have placed the rules for adopting a goal in the **main module**. This choice violates the design rule of putting rules that only affect the mental state of the agent in the **event module**. In Chapter **??** on modules, we will see how we can modify the agent program in a way that also complies with this design rule.

This concludes the design of our agent for the dynamic Tower World. A complete listing of the agent can be found in the Appendix (Section 5.10).

5.6 Environments and Observability

The Blocks World and the Tower World variant of it are *fully observable*. This means that the sensors or perceptual interface of the agent provide it with full access to the state of the environment. It is important to understand that full observability is always *relative to the representation of the environment*. In the Tower World, for example, the agent did not have access to the exact position of the gripper. It cannot determine by means of the percepts it receives whether the gripper is hovering just above block b or not. As a consequence, it may revise its goals to pick up block b just before grabbing it because another move can be made that is constructive while b cannot be

moved constructively. In terms of time, then, the action selection of the agent may not be optimal in a dynamic environment such as the Tower World although that is hard to determine.

The Tower World is fully observable *with respect to the representation of that environment* used by the agent and provided via the perceptual interface. The language to represent its environment is simple and consists of just three predicates: `block(X)`, `on(X,Y)`, and `holding(X)`. The percepts provided to the agent every cycle provide it with a complete representation of the state of its environment: all blocks present in the environment are provided using the `block` predicate, all relations of one block directly on top of another block are provided using the `on` predicate, and using the `holding` predicate the state of the gripper is completely specified. In other words, there is no fact that could be added to the list of percepts that each time is provided in terms of this language and these predicates to give the agent a more complete picture of its environment. Only by using a richer language, that e.g. would also be able to represent the position of the gripper, it would be possible to create a refined representation of the environment.

Usually, however, environments do not allow for a complete reconstruction of a representation of their state via the percepts the agent receives from that environment. More often the agent has only a partial view of its environment and can only maintain a reasonably adequate representation of a local part of the environment. In such an environment an agent must typically explore to achieve its goals. The Wumpus World provides an example where initially the agent does not know where the gold is located and the agent needs to explore the grid to locate it. Agents in this case only have *incomplete information* about the state of the environment and need to take this into account when selecting actions to perform next. For example, the agent in the Wumpus World cannot know for certain that the grid cell that the agent is facing is not a pit if it stands on a breeze without additional information. The reasoning to decide on which action to perform next is more complicated in this case than in an environment that is fully observable.

The Wumpus World, however, is a rather special world because there is only one agent and the environment itself is *not* dynamic. By fully exploring the Wumpus World, which in principle is possible if the grid is finite and all grid cells can be reached, the agent *can* construct a complete representation of the environment. In dynamic environments, such as gaming environments - for example Unreal Tournament, it is impossible to ever construct a complete and accurate representation of the environment. An agent's decision making gets even more involved in such cases, as it then must make decisions with incomplete information.

The fact that observability is relative to a representation is an important one in dynamic environments as well. Even in a dynamic environment such as *Unreal Tournament*, for example, certain aspects of the environment *are* fully observable. An example in this case is the position of the agent.

5.7 Summary

Agents are connected to an environment in which they act. We have discussed the need to be able to sense *state changes* by agents in all but a few environments. The only environments in which an agent does not need a sensing capability is an environment in which the agent has full control and actions do not fail. The Blocks World environment in which no other agents operate is an example of such an environment. In this environment a single agent is able to maintain an accurate representation of its environment by means of a correct specification of the effects of the actions it can perform.

Perception is useful when an agent has *incomplete* knowledge about its environment or needs to *explore* it to obtain sufficient information to complete a task. For example, a robot may need to locate an item in a partially observable environment. Such a robot may not even know how its environment is geographically structured and by exploring it would be able to *map* the environment. But even in a simple environment such as the Wumpus World the agent does not have a complete overview of the environment and needs to explore its environment to determine what to do.

Summarizing, perception is useful when an agent acts in an environment in which:

(i) the agent does not have *full control*, and events initiated by the environment may occur (*dynamic environments*),

(ii) *other agents* also perform actions (another kind of dynamic environment),

(iii) the environment is not *fully observable* but e.g. needs to be explored, or

(iv) effects of actions are uncertain (*stochastic* environments), or actions may *fail*.

In such environments the agent is not able to maintain an accurate representation of its environment without perceiving what has changed. For example, if stacking a block on top of another block may fail, there is no other way to notice this than by receiving some signal from the environment that indicates such failure. Throwing a dice is an example of an action with uncertain outcomes where perception is necessary to know what outcome resulted. When other agents also perform actions in an environment a sensing capability is also required to be able to keep track of what may have changed.

Environments may also change because of natural events and may in this sense be viewed as "active" rather than "passive". Again sensing is required to be able to observe state changes due to such events. This case, however, can be regarded as a special case of (iii) other agents acting by viewing the environment as a special kind of agent. The main purpose of sensing thus is to maintain an accurate representation of the environment. Fully achieving this goal, however, typically is not possible as environments may be only *partially observable*.

5.8 Notes

We have discussed to need to be able to perceive state changes, but we have not discussed the perception of *actions* that are performed by agents. It is easier to obtain direct information about the environment state than about actions that are performed in it. In the latter case, the agent would have to derive which changes result from the actions it has observed.

Recently programming with environments has become a topic in the multi-agent literature. See e.g. [44]. Agent-environment interaction has been discussed in various other contexts. See e.g. [3, 53].

5.9 Exercises

5.9.1

1. Will the *stackBuilder* agent of Figure 5.4 be able to achieve its goal when it and the *tableAgent* get control over the gripper in the Blocks World? Assume that the agents are treated fair, and each of the agents can perform the same number of actions over time. Alternatively, you may also make the assumption that the agents take turns.

2. Test in practice what happens when the agents are run using the Goal IDE. Select different middleware platforms in the Run menu of the IDE to run the agents and report what happens. Do the agents always perform the same number of actions?

3. Test the multi-agent Blocks World system again but now distribute the agents physically over different machines. Do the agents always perform the same number of actions?

5.9.2

In dynamic environments such as the Tower World it is particularly important to *test* the agent in all kinds of different scenarios that may occur. To test an agent for all the different kinds of scenarios that may occur, a thorough analysis of what can happen in an environment is needed. For the Tower World environment, explain why the `towerBuilder` agent of Section 5.10 will act adequately in each of the scenarios below. Include references to code lines in the agent program and explain why these lines are necessary and sufficient to handle each scenario.

1. The agent has a goal `holding(X)` and
   (a) the user puts a different block `Y` in the gripper. Consider both the scenario where a constructive move can be made with block `Y` and the scenario where this is not possible.
   (b) the user puts a block on top of block `X`.
   (c) the user changes the configuration such that it becomes infeasible to put block `X` into position.
   (d) the user puts block `X` into position.
2. The agent is holding block `X` and
   (a) the user removes block `X` from the gripper. Consider both the scenario where block `X` is put into position and the scenario where it is made infeasible to pick up block `X` again.
   (b) the user changes the configuration such that it becomes infeasible to put block `X` into position.

5.10 Appendix

```
main: towerBuilder
{
  knowledge{
    % assume there is enough room to put all (max 13 in Tower World) blocks on the table.
    clear(table) . clear(X) :- block(X), not(on(_, X)), not(holding(X)) .
    above(X,Y) :- on(X, Y) . above(X,Y) :- on(X, Z), above(Z, Y) .
    tower([X]) :- on(X, table) . tower([X,Y|T]) :- on(X, Y), tower([Y|T]) .
  }
  goals{
    on(a,b), on(b,c), on(c,table), on(d,e), on(e,f), on(f,table) .
  }
  main module{
    program{
      % moving X on top of Y is a constructive move if that move results in X being in position.
      #define constructiveMove(X, Y) a-goal( tower([X, Y | T]) ),
              bel( tower([Y|T]), clear(Y), (clear(X) ; holding(X)) ) .
      % a block X is misplaced if it prevents moving a block Y in position either because there is a block
      % above the target block, or because a block above Y prevents moving it.
      #define misplaced(X) a-goal( on(Y, Z) ), bel( above(X, Z); above(X, Y) ) .

      % prefer making constructive moves.
      if constructiveMove(X, Y) then adopt( holding(X) ) .
      % if no constructive move can be made aim for moving obstructing block.
      if misplaced(X) then adopt( holding(X) ) .

      % pick up a block if you can and want to.
      if a-goal( holding(X) ) then pickup(X) .
      % put a block you're holding down, ideally where you want it, but otherwise put it on the table.
      if bel( holding(X) ) then {
        if constructiveMove(X,Y) then putdown(X, Y) .
        if true then putdown(X, table) .
      }
      % otherwise, there is nothing to do, so we can move the gripper to the top left corner.
      if true then nil .
    }
  }
  event module{
    program{
      % moving X on top of Y is a constructive move if that move results in X being in position.
      #define constructiveMove(X, Y) a-goal( tower([X, Y | T]) ),
              bel( tower([Y|T]), clear(Y), (clear(X) ; holding(X)) ) .
      % a block is *in position* if it achieves a goal.
      #define inPosition(X) goal-a( tower([X|T]) ) .

      % first, process percepts from Tower World environment, assume full observability.
      forall bel( block(X), not(percept(block(X))) ) do delete( block(X) ) .
      forall bel( percept(block(X)), not(block(X)) ) do insert( block(X) ) .
      forall bel( holding(X), not(percept(holding(X))) ) do delete( holding(X) ) .
      forall bel( percept(holding(X)), not(holding(X)) ) do insert( holding(X) ) .
      forall bel( on(X,Y), not(percept(on(X,Y))) ) do delete( on(X,Y) ) .
      forall bel( percept(on(X,Y)), not(on(X,Y)) ) do insert( on(X,Y) ) .

      % check for reasons to drop or adopt a goal (goal management).
      if goal( holding(X) ) then {
        % first reason: cannot pick up block X because it's not clear.
        if not(bel( clear(X) )) then drop( holding(X) ) .
        % second reason: cannot pick up block X because now holding other block!
        if not(bel( holding(_) )) then drop( holding(X) ) .
        % third reason: block X is already in position, don't touch it.
        if inPosition( X ) then drop( holding(X) ) .
        % fourth reason: we can do better by moving another block constructively.
        listall L <- constructiveMove(Y,Z) do {
          if bel(not(L=[]), not(member([X,_,_],L))) then drop( holding(X) ) .
        }
      }
    }
  }
  actionspec{
    pickup(X) { pre{ clear(X), not(holding(_)) } post{ true } }
    putdown(X,Y) { pre{ holding(X), clear(Y) } post{ true } }
    nil { pre{ true } post{ true } }
  }
}
```

Figure 5.6: Tower Builder agent

Chapter 6

Communicating Rational Agents

6.1 Introduction

In a multi-agent system, it is useful for agents to communicate about their beliefs and goals. Agents may have only a partial view of the environment, and by communicating, agents may inform each other about parts they but other agents cannot perceive. Agents may also use communication to share goals and coordinate the achievement of these goals. Communication is essential in situations where agents have different roles and need to delegate actions to appropriate agents, or when agents with conflicting goals operate in the same environment space and need to coordinate their actions to prevent deadlocks or other less serious but inefficient interactions with other agents.

This chapter will explain how communication works in GOAL, and how to program communicating GOAL agents. First, the creation of a multi-agent system by means of a multi-agent system (mas) file is explained. A mas file is used to *launch* a multi-agent system by creating and connecting agents to an environment. A mas file specifies a *policy* for launching a multi-agent system. The communication primitives and processing of messages are introduced next. Processing messages is in many respects similar to the processing of percepts but there are some small but important differences. There are different forms of communication and messages available in GOAL which are introduced in Section 6.5. Of course, it is important to address the right agents when an agent starts communicating and Section 6.6 discusses how to select the agents that should receive a message. Finally, an example multi-agent system is presented to illustrate the use of communication for coordinating the actions of multiple agents.

6.2 Launching a Multi-Agent System

This section explains how to launch a multi-agent system using a mas file. We have already seen a simple example of two Blocks World agents being connected to an environment in Chapter 5. Here we will discuss in more detail how a set of agents can be launched together to form a running multi-agent system in GOAL. There are various issues that are handled by a mas file. One issue handled by a mas file is *when to launch an agent* as part of a multi-agent system. Agents can be launched upon creation of the multi-agent system, or when an entity in the environment is available that can be controlled by an agent (we say that an entity is *born*). Another issue managed by a mas file is *whether or not to connect an agent to an environment.*

6.2.1 Multi-Agent System Files

A multi-agent system in GOAL is specified by means of a *mas file*. A mas file provides a *recipe* for launching a multi-agent system. In a sense, a mas file is a program on its own. It specifies which environment should be connected to, which agents should be launched and when, how many of those agents should be launched, and which GOAL agent files should be used to launch an agent.

masprogram	::=	[*environment*] *agentfiles launchpolicy*
environment	::=	**environment{** *path*. [*initialization*.] **}**
path	::=	any valid path to an environment file in quotation-marks
initialization	::=	**init**[options]
options	::=	list of options valid with respect to the selected environment
agentfiles	::=	**agentfiles{** *agentfile*. { *agentfile*.}* **}**
agentfile	::=	*path* [*agentparams*] **.**
agentparams	::=	[*nameparam*] | [*langparam*] | [*nameparam* **,** *langparam*] | [*langparam* **,** *nameparam*]
nameparam	::=	**name** = *id*
langparam	::=	**language** = *id*
launchpolicy	::=	**launchpolicy{** { *launch* | *launchrule* }$^+$ **}**
launch	::=	**launch** *basiclaunch* {**,** *basiclaunch*}* **.**
basiclaunch	::=	*agentbasename* [*agentnumber*] **:** *agentref*
agentbasename	::=	***** | *id*
agentnumber	::=	[*number*]
agentref	::=	plain agent file name without extension, or reference name
launchrule	::=	**when** *entitydesc* **do** *launch*
entitydesc	::=	[*nameparam*] | [*typeparam*] | [*maxparam*] | [*nameparam* **,** *typeparam*] | [*typeparam* **,** *nameparam*] | [*maxparam* **,** *typeparam*] | [*typeparam* **,** *maxparam*]
typeparam	::=	**type** = *id*
maxparam	::=	**max** = *number*
id	::=	an identifier starting with a lower-case letter
num	::=	a natural number

Figure 6.1: Mas File Grammar

Multiple agents may, for example, be created using a single GOAL agent file. The mas file specifies an *environment connection policy* that determines which agents should be connected to an entity in an environment. The mas file also specifies a *baptizing policy*: it determines which names will be assigned to each agent. Finally, one important function of a mas file is that it provides the information to locate the relevant files that are needed to run a multi-agent system and the associated environment.

A BNF grammer for mas files is provided in Figure 6.1. A mas file consists of three sections:

1. an *environment* section that specifies which environment interface should be launched,
2. an *agent files* section with references to agent files that are used to create agents, and
3. a *launch policy* section that specifies a policy for naming and launching agents using the agent files in the agent files section.

The environment section is optional. Agents need not be connected to and can run without an environment. If no environment section is present, agents will not be connected to an entity in an environment but will run like any other program. The mas file for the Coffee Factory example that we will introduce below does not have an environment section, for example. Note that even when an environment is present agents do not need to be connected to that environment. If present, the environment section should contain a reference to an environment interface file. GOAL supports the Environment Interface Standard (EIS) and expects a jar-file that conforms with this standard. EIS provides a toolbox for developing environment interfaces that has been developed to facilitate the connection and interaction of agent platforms such as GOAL with environments [5, 4].

A reference to the Elevator environment is provided as an example below. In this case the reference is simply the name of the environment interface file but any valid path name to such a file between quotes can be used.

```
environment{
  "elevatorenv.jar" .
}
```

It is useful to be able to launch an environment with a specific set of initialization parameters. By doing so, setting up an environment is simplified. An environment may have multiple parameters that can be initialized. The documentation of an environment should provide information on the parameters that are available. Parameters are specified using the **init** command followed by key-value pairs of the form `key=value`. The Elevator environment has quite a few parameters. The type of simulation, the number of floors, the number of cars, and other parameters can be initialized. The following is an example specification of parameters for the Elevator environment:

```
environment{
  "elevatorenv.jar" .

  init[Simulation = "Random Rider Insertion", Floors = 10, Cars = 3,
     RandomSeed=635359, Capacity=5, People=50, InsertionTime=260000,
     TimeFactor=0, Controller="GOAL Controller"
  ] .
}
```

The agent files section contains references to one or more GOAL agent files. These are the files that can be used to create agents. Launch rules reference these files for creating agents. An agent file reference is a path to the file including the agent filename itself or simply the filename of the agent file. An example reference to an agent file for the Elevator environment is illustrated below.

```
agentfiles{
  "elevatoragent.goal" .
}
```

The files in the agent files section need to be referenced in the launch policy section. If the agent files section only contains plain filename references such as above, the label used to reference an agent file is simply the filename without the extension. For example, the reference to be used for the elevator agent file above is `elevatoragent`. It is possible to associate additional options with an agent file. The **name** option allows to introduce a new reference label for an agent file. This option is useful because only local changes to the agent files section are needed when an agent file is replaced with a different agent file. References to the file in the launch policy section then do not need to be updated. There is a second **language** option for setting the knowledge representation language that is used by the agent. The default value of this option is `swiprolog`. Here is an example of how to use these options:

```
agentfiles{
  "elevatoragent1.goal" [name=agentfile1, language=swiprolog] .
  "elevatoragent2.goal" [name=agentfile2, language=pddl] .
}
```

This agent files section introduces two agent files. By using the **name** option the first file now needs to be referenced by the label `agentfile1`. The plain filename reference `elevatoragent1` cannot be used anymore. The **language** option indicates that this agent uses SWI Prolog as its knowledge representation language. The reference to the second file is set to `agentfile2`. The **language** option specifies that this agent uses the PDDL language for knowledge representation.[1]

[1] Only language options that are supported by GOAL can be used. Support for PDDL is being developed at the moment and is not yet available. PDDL stands for Planning Domain Definition Language and is a standard language used to provide input to planners.

The launch policy section specifies a policy for launching agents. The section contains one or more *launch rules*. There are two types of lauch rules: *conditional launch rules* and *unconditional launch rules*. We first introduce launch rules without conditions. An unconditional launch rule is of the form **launch**<agentName>:<agentReference>.. To illustrate these rules we use the Coffee Factory multi-agent system because this mas is not connected to an environment. The reason is that unconditional launch rules cannot be used to connect agents to an environment. We will use the Elevator environment to illustrate conditional launch rules.

```
agentfiles{
  "coffeemaker.goal" .
      "coffeegrinder.goal" .
}
launchpolicy{
  launch maker:coffeemaker .
  launch grinder:coffeegrinder .
}
```

This example contains two simple launch rules. When a multi-agent system is launched, all unconditional launch rules are applied before running the multi-agent system. In this case, the first launch rule will create an agent named `maker` using the agent file `coffeemaker` and the second launch rule will create an agent named `grinder` using the agent file `coffeegrinder`.

In the presence of an environment, agents need to be connected to entities in the environment. In order to do so, there first need to be entities present in an environment. Because we do not know when launching a multi-agent system whether this is (already) the case or not, a different approach is used to connect agents to entities. The idea is that each time when a so-called *free* entity is available in the environment, an applicable conditional launch rule is triggered. A launch rule is applicable if the condition of the rule matches with the entity that is available or born. We will discuss these conditions shortly. First, we will look at a simple and often used launch rule that is a straightforward implementation of the idea:

```
launchpolicy{
  when entity@env do launch elevator:fileref .
}
```

A free entity becomes available in the Elevator environment each time when this environment creates a new elevator carriage. The conditional launch rule above is triggered for arbitrary entityies that are available. If there is a free entity, applying the rule creates an agent using the agent file reference `fileref`, baptizes the agent using the base name `elevator`, and connects it to the entity. From that moment on the agent is able to control the entity. If another free entity is available, the rule will create a new agent and connects it to that entity. The same base name is used to baptize the agent but a unique number is attached to distinguish the agent from the other agents. For example, if four carriages are available, agents named `elevator`, `elevator1`, `elevator2`, and `elevator3` will be created.

Entities may already have names themselves which can also be used to name an agent. The name of the entity can be used to name the agent as well by inserting an asterisk for the agent name in a launch rule. This is useful for keeping track of which agent controls which entity.

```
launchpolicy {
  when entity@env do launch *:fileref .
}
```

The asterisk means that the name of the entity in the environment is used as the base name for baptizing agents.

An agent name in a launch rule can be supplied with an additional option that specifies how many agents should be launched and connected to an entity. This option is available when connecting to an entity but it can also be used in unconditional rules. For example, if we want three coffee maker agents in the Coffee Factory mas, the following launch rule will achieve this:

```
launchpolicy{
  launch maker[3]:coffeemaker .
  launch grinder:coffeegrinder .
}
```

The first launch rule instantiates three agents using `maker` as base name. The names for these agents would be `maker`, `maker1`, and `maker2`. The option to launch multiple agents at the same time facilitates launching large numbers of agents without the need to specify a large number of different agent names.

In exactly the same way several agents can be launched when an entity is available:

```
launchpolicy{
  when entity@env do launch elevator[3]:fileref .
}
```

This launch rule will instantiate three agents and associate them with one and the same entity. As a result, if the entity would perceive something, each of these three agents will receive that percept. Moreover, if any of these agents performs an environment action, it will be performed by the entity.

We have already seen an alternative method for connecting multiple agents to an entity in Chapter 5. There we simply listed multiple agent names with associated file references, see also the grammar in Figure 6.1.

The condition of a launch rule may perform additional checks besides verifying that an entity is available. The first check that can be performed is checking the name of an entity. By using the **name** option a launch rule will only trigger if an entity with the specified name is available. A second check is more general and checks the *type* of the entity. A launch rule with a condition on the type of an entity will trigger only if an entity with that type is available. A third check concerns the *number of applications* of a launch rule itself. Each application of a launch rule is counted and the number of applications of a particular rule may be restricted to a certain maximum number. Here is an example of a launch rule for the Elevator environment that uses the last two conditions:

```
launchpolicy{
  when [type=car,max=3]@env do launch elevator:fileref .
}
```

This launch rule specifies that it can be applied at most three times and will only trigger when an entity of type `car` is available.

An example of a rule that uses the name of an entity:

```
launchpolicy{
  when [name=car0]@env do launch elevator:fileref .
}
```

This rule will only be applied if the free entity has the name `car0`.

A final remark on launch policies concerns the order of rules. Rules in the launch policy section are applied in order. This means that the first rule that can be applied will be applied. A different order of rules therefore may generate a different set of agents. For example, we can combine the two rules we just discussed in two different ways. By ordering them as follows:

```
launchpolicy{
  when [type=car,max=3]@env do launch elevator:fileref .
  when [name=car0]@env do launch elevator:fileref .
}
```

it may be that the last rule never gets applied because the first rule has already connected an agent to the entity `car0`. As a result, when four carriages become available, it may be that only three are connected with an agent because the first rule can be applied at most three times and the last rule will not fire. By reversing the order of the rules it is always possible to connect four agents to a carriage, that is, if one of them is called `car0`.

6.2.2 Automatic `agent` and `me` fact generation

A mas file launches potentially many agents and provides each of these agents with a name. How do the agents keep track of other agents that are available, and how do they identify these agents? It is not very practical to hardcode names into an agent, and bad practice to do so. Moreover, because the number of agents that are launched may not be known it may be impossible determine their names when the agents are being developed.

To resolve this issue, GOAL automatically inserts the names of agents into the belief bases of all known agents. A special predicate `agent` is used to insert new naming facts in the belief base of an agent whenever a new agent is created as part of the mas. That is, whenever an agent is launched with a name `name`, the belief base of this and all other agents is populated with a fact `agent(name)`. An agent programmer may thus assume that, at any time, `bel(agent(X))` will result in a substitution for `X` for each agent that is part of the mas.

In order to identify itself, moreover, an agent is also provided with knowledge about its own name. This provides the agent with the ability to distinguish itself from other agents. In addition to the `agent` facts, a special `me` fact is inserted into its beliefbase. It has the form `me(<agentname>)` where `<agentname>` is the name of the agent, as determined by the launch policy.

After launching the `coffee.mas` file to create the Coffee Factory mas and applying both launch rules, the belief base of the coffee maker agent would look like this:

```
beliefs{
  have(water). have(beans).
  canMake(maker, [coffee, espresso]).
  agent(maker).
  agent(grinder).
  me(maker).
}
```

Note that unless an agent wants to actively ignore some agent, it is unwise to *delete* `agent` facts from the belief base.

As a final illustration of mas files and their logic, we provide one more example of a somewhat larger mas file in Figure 6.2.

```
1  environment{
2    "elevatorenv.jar".
3  }
4
5  agentfiles{
6    "elevatoragent.goal" [name=elevatorfile] .
7    "managingagent.goal" [name=managerfile] .
8  }
9
10 launchpolicy{
11   launch manager:managerfile .
12   when [type=car,max=1]@env do launch elevator1:elevatorfile .
13   when [type=car,max=1]@env do launch elevator2:elevatorfile .
14   when [type=car,max=1]@env do launch elevator3:elevatorfile .
15 }
```

Figure 6.2: A more complex mas file

This example uses relative paths to the files and labels to reference those files. One elevator agent will be launched and associated with each entity in the environment of type `car` (at most three times).

After all three elevator agents have been launched, the belief base of `elevator2` will look like

```
beliefs {
  ... % other facts
  agent(manager).
  agent(elevator1).
  agent(elevator2).
  agent(elevator3).
  me(elevator2).
}
```

6.3 Example: The Coffee Factory Multi-Agent System

Throughout this chapter we will illustrate various concepts of multi-agent systems and communication by means of an example. The example multi-agent system that we will use concerns a set of agents that make coffee. We call this multi-agent system the Coffee Factory mas.

The Coffee Factory mas is a multi-agent system in which a coffee maker and a coffee grinder work together to brew a fresh cup of coffee. Optionally, a milk cow can provide milk for making a latte. To make coffee the coffee maker needs *water* and *coffee grounds*. It has water, and *coffee beans*, but not *ground* coffee. Grinding the beans is the task of the coffee grinder. The coffee grinder needs beans, and produces grounds. The programs of the coffee maker and the coffee grinder are listed in Figures 6.6 and 6.7, respectively. Figure 6.3 lists a shorter version of the agent program for the coffee maker that does not include comments.

The agents are designed in such a way that they know which ingredients are required for which products. They know what they can make themselves, but they do not initially know what the other agents can make. This is where communication comes in.

The **knowledge** section reflects the agent's knowledge of which ingredients are necessary for which products. The **beliefs** section holds the agent's beliefs in what it can make. In this case, the `maker` can make `coffee` and `espresso`. The **goals** section states this agent's mission: having `coffee`. Note that this describes a goal *state* (`coffee` being available), not an action (like 'making `coffee`'). Also note that the **event module** contains all communication related action rules, meaning that every round all instances of these action rules are evaluated.

The following simplifying assumptions have been made while designing the Coffee Factory mas:

1. Resources (like `water`, `beans`, `grounds` and `coffee`) cannot be depleted.

2. The agents share the resources in the sense that if one agent has a resource, all agents do. But there is no environment, so agents cannot *perceive* changes in available resources; they have to communicate this. For example, if the coffee grinder makes `grounds`, it will thereafter believe `have(grounds)`, but the coffee *maker* will not have this belief until it gets informed about it.

6.4 Communication: Send Action and Mailbox

A basic form of communication is supported in GOAL by means of a so-called *mailbox semantics*. Messages received are stored in an agent's mailbox and may be inspected by the agent by means of queries on special, reserved predicates `sent(`*agent*,*msg*`)` and `received(`*agent*,*msg*`)` where *agent* denotes the agent the message has been sent to or received from, respectively, and *msg* denotes the content of the message expressed in a knowledge representation language.

The action `send(AgentName, Poslitconj)` is a built-in action to send `Poslitconj` to the agent with given `AgentName`. `Poslitconj` is a conjunction of positive literals. `AgentName` is an atom with the name of the agent as specified in the mas file. Messages that have been sent

```
main: coffeeMaker {
  knowledge{
    requiredFor(coffee, water).
    requiredFor(coffee, grounds).
    requiredFor(espresso, coffee).
    requiredFor(grounds, beans).

    canMakeIt(M, P) :- canMake(M, Prods), member(P, Prods).
  }
  beliefs{
    have(water). have(beans).
    canMake(maker, [coffee, espresso]).
  }
  goals{
    have(coffee).
  }
  main module{
    program {
      if goal(have(P)) then make(P).
    }
  }
  event module {
    program{
      forall bel(agent(A), not(me(A)), not(canMake(A, _))) then sendonce(A, ?canMake(A, _)) .
      forall bel(me(Me), received(A, int(canMake(Me, _))), canMake(Me, Prod))
         then sendonce(A, :canMake(Me, Prod)).
      forall bel(received(Sender, canMake(Sender, Products))) then
         insert(canMake(Sender, Products)) + delete(received(Sender, canMake(Sender, Products))).

      forall bel(agent(A), received(A, have(X))), not(bel(have(X))) then insert(have(X)) .
      forall goal(have(P)), bel(requiredFor(P, R), not(have(R))), bel(canMakeIt(Me, R), me(Me))
         then adopt(have(R)) .
      forall goal(have(P)), bel(requiredFor(P, R), not(have(R))),
         bel(canMakeIt(Maker, R), not(me(Maker))) then sendonce(Maker, !have(R)) .
      forall goal(have(P)), bel(requiredFor(P, R), not(have(R)), me(Me), not(canMakeIt(Me, P)))
         then sendonce(allother, !have(R)).
      forall bel(agent(Machine), received(Machine, imp(have(X))), have(X))
          then sendonce(Machine, :have(X)).
    }
  }
  actionspec{
    make(Prod) {
      pre{ forall(requiredFor(Prod, Req), have(Req)) }
      post { have(Prod) }
    }
  }
}
```

Figure 6.3: The Coffee Maker

are placed in the mailbox of the sending agent, as a predicate of the form `sent(AgentName, Poslitconj)` (note the 't' at the end of `sent`). The message is sent over the selected middleware to the target agent, and after arrival the message is placed there in the form `received(SenderAgentName, Poslitconj)` where `SenderAgentName` is the name of the agent that sent the message. Depending on the middleware and distance between the agents, there may be delays in the arrival of the message. Throughout we will assume that messages always are received by their addressees.

6.4.1 The **send** action

To illustrate the built-in `send` action, let's first consider a simple example multi-agent system consisting of two agents, *fridge* and *groceryplanner*. Agent *fridge* is aware of it's contents and will notify the *groceryplanner* whenever some product is about to run out. The *groceryplanner* will periodically compile a shopping list. At some point, the fridge may have run out of milk, and takes appropriate action:

```
program {
    ...
    if bel(amountLeft(milk, 0)) then send(groceryplanner, amountLeft(milk, 0)).
    ...
}
```

At the beginning of its action cycle, the *groceryplanner* agent gets the following fact inserted in its message base.

```
    received(fridge, amountLeft(milk, 0)).
```

The received messages can be inspected by means of the `bel` operator. In other words, if an agent has received a message M from sender S, then `bel(received(`S, M`))` will be true; the agent believes it has received the message. This also holds for `bel(sent(`R, M`))`, where R is the recipient of the message. This way, the *groceryplanner* can act on the received message:

```
program {
    ...
    if bel(received(fridge, amountLeft(milk, 0))) then adopt(buy(milk)).
}
```

6.4.2 Mailbox management

In contrast with the percept base, mailboxes are *not* emptied automatically. This means that once a message is sent or received, that fact will remain in the message base, even after execution of the above program rule. The consequence of this is that the next action cycle, the *fridge* may again select the shown program rule, sending the same message again, over and over. Also, the *groceryplanner* will keep selecting this program rule.

We have to do something to prevent this. There may be some special cases in which it is preferred to leave the message in the mailbox, for example if the message contains some message counter, so you can review the whole message history. Otherwise it is possible that a new message containing the same content sent to the same recipient will not be seen as a new message. So, we need to remove the `received` when we process them. For this an internal action is added to the action rule.

```
    if bel(received(fridge, amountLeft(milk, 0)))
        then adopt(buy(milk)) + delete(received(fridge, amountLeft(milk, 0))).
```

If the *fridge* sends this message only once, this program rule will be selected only once.

The coffee maker agent from Section 6.2 also provides an example of a similar rule:

```
% process information from other agents on what they can make
forall bel(received(Sender, canMake(Sender, Products)))
    then insert(canMake(Sender, Products)) + delete(received(Sender, canMake(Sender, Products)))
```

The logic is slightly different for the sender, because if it would remove the `sent` fact it would lose the belief that it has already notified the *groceryplanner*, and send the message again. Instead it can use this information to prevent repeatedly sending the same message:

```
if bel(amountLeft(milk, 0), not(sent(groceryplanner, amountLeft(milk, 0))))
   then send(groceryplanner, amountLeft(milk, 0)).
```

The `sendonce` action

Because the above leads to verbose programming, GOAL offers a variant of the `send` action: the `sendonce` action. The syntax is the same as that of `send`, but the semantics are such that the message is sent only if there is no `sent` fact for that message (and receiver(s)) in the mailbox. Writing

```
forall bel(agent(A), fact(P)) then sendonce(A, fact(P)).

% if some machine seems to need a product, tell it we have it
forall bel(agent(Machine), received(Machine, imp(have(P))), have(P))
    then sendonce(Machine, have(P)).
```

is short for

```
forall bel(agent(A), fact(P), not(sent(A, fact(P)))) then send(A, fact(P)).

% if some machine seems to need a product, tell it we have it
forall bel(agent(Machine), received(Machine, imp(have(P))),
    have(P), not(sent(Machine, have(P)))) then send(Machine, have(P)).
```

This means that if the `sent` fact is deleted from the mailbox, the message may henceforth be sent again by the `sendonce` action.

6.4.3 Variables

In GOAL programs, the use of variables is essential to writing effective agents. Variables can be used in messages as expected. For example, a more generic version of the *fridge*'s program rule would be

```
if bel(amountLeft(P, N), N < 2, not(sent(groceryplanner, amountLeft(P, N))))
    then send(groceryplanner, amountLeft(P, N)).
```

Note that here we used an **if**...**then**... rule but that with this rule the agent will still eventually send one message for every value of `N` where `N` < 2. Of course, by using a **forall** rule the groceryplanner would be informed at once of all messages.

Recipients and senders can also be variables in the mental state condition. An example:

```
% This isn't an argument; it's just contradiction!
% - No it isn't.
forall bel(received(X, fact)) then send(X, not(fact)).

% http://en.wikipedia.org/wiki/Marco_Polo_(game)
forall bel(received(X, marco)) then send(X, polo).
```

This is especially useful in situations where you don't know who will send the agent messages, as with the coffee domain example:

```
% answer any question about what this agent can make
forall bel(me(Me), received(A, int(canMake(Me, _))), canMake(Me, Prods))
  then sendonce(A, canMake(Me, Prods)).
```

For any agent A it has received a question from, it will answer its question.

Closed actions

In order for any action in GOAL to be selected for execution, that action must be closed, meaning that all variables in the action must be bound after evaluation of the mental state condition. As a consequence, *messages* must be closed as well, in order to make the action executable.

6.5 Moods

GOAL agents are goal-oriented agents which have their goals specified declaratively. Up until now all examples have shown communication of an agent's *beliefs*. Every message was a statement about the sender's beliefs regarding the content. Given GOAL's goal-orientedness, it would be useful to be able to not only communicate in terms of beliefs but also in terms of *goals*. This way GOAL agents can tell other agents that they have a certain goal.

In natural language communication, such a *speech act* is often performed by *uttering* a *sentence* in a certain *mood*. This mood can be *indicative* ('The time is 2 o'clock'), *expressive* ('Hurray!!'), *declarative* ('I hereby declare the meeting adjourned').

In GOAL, the execution of the `send` action is the *uttering*, the message content is the *sentence*. The *mood* is indicated by prefixing the message content with a *mood operator*. GOAL distinguishes three moods listed in Figure 6.5.

Mood	operator	example	NL meaning
INDICATIVE	:	`:amountLeft(milk, 0)`	"I've run out of milk."
DECLARATIVE	!	`!status(door, closed)`	"I want the door to be closed!"
INTERROGATIVE	?	`?amountLeft(milk, _)`	"How much milk is left?"

Figure 6.4: GOAL message moods

In the case of the indicative mood the mood operator is optional. In other words, in absence of a mood operator, the indicative mood is assumed. That means that all examples in Section 6.4 were implicitly in the indicative mood.

Using these mood operators, GOAL agents can be more GOALish in their communication. For example, if the coffee maker or coffee grinder needs a resource to make something but does not have it, it can inform an agent that it believes *does* have it that it needs it:

```
% if we need a product but don't have it, notify an agent that does have it that we need it.
forall goal(have(P)), bel(requiredFor(P, R), not(have(R)), canMakeIt(Maker, R))
   then send(Maker, !have(P)).
```

Now for the receiving side of the communication. Moods of messages in the mailbox are represented as predicates, allowing for logic programming. An imperative is represented by the `imp` predicate, an interrogative mood by the `int` predicate. There is no predicate for the indicative mood in the mailbox. Using these mood predicates, we can inspect the mailbox for messages of a specific type. For example, to handle a message like the one above from the coffee maker, the coffee grinder can use this action rule:

```
% if some agent needs something we can make, adopt the goal to make it
forall bel(received(_, imp(have(P))), me(Me), canMakeIt(Me, P)) then adopt(have(P)).
```

The coffee grinder will grind beans for whichever agent needs them, so here a *don't care* is used in place of the sender parameter. Another rule will make sure that the correct agent is notified of the availability of the resulting grounds.

The previous section mentioned that messages must be closed. There is one exception, which concerns interrogative type messages. These messages are like open questions, like, for example, "What time is it?" or "What is Ben's age?". These cannot be represented by a closed sentence. Instead, a *don't care* can be used to indicate the unknown component. For example:

```
if not(bel(timeNow(_))) then send(clock, ?timeNow(_)).

if not(bel(age(ben, _))) then send(ben, ?age(ben, _)).

% ask each agent what they can make
forall bel(agent(A), not(me(A)), not(canMake(A, _))) then sendonce(A, ?canMake(A, _)).
```

6.6 Agent Selectors

In many multi-agent systems agents may find themselves communicating with agents whose name they do not know beforehand. For example, the mas may have been created by launching 100 agents using the `agent[100]:file1` syntax. If some of these agents need to communicate with each other, the agent that needs to be addressed needs to be selected somehow. In a multi-agent system it may also be useful to multicast or broadcast a message to multiple receivers. For these cases a flexible way of addressing the receivers of messages is needed.

6.6.1 `send` action syntax

The `send` action allows more dynamic addressing schemes than just the agent name by means of an *agent selector*. The syntax of the `send` action and this agent selector is shown in Figure 6.5.

The first parameter to the `send` action (agent name in the previous sections) is called an *agent selector*. An agent selector specifies which agents are selected for sending a message to. It consists of one or more *agent expression*s, surrounded by square brackets. The square brackets can be omitted if there is only one agent expression.

Some examples of agent selectors:

```
% agent name
send(agent2, theFact).

% variable (Prolog)
send(Agt, theFact).

% message to the agent itself
send(self, theFact).

% multiple recipients
send([agent1, agent2, self, Agt], theFact).

% using quantor
% if we don't know anyone who can make our required resource, broadcast our need
forall goal(have(P)), bel(requiredFor(P, R), not(have(R)), not(canMakeIt(_, R)))
   then send(allother, !have(P)).
```

Agent Name

The agent name is the simplest type of agent expression, which we have already seen in Sections 6.4 and 6.5. It consists of the name of the receiving agent. If the KR language of the agent is Prolog, the agent name must start with a lowercase letter.

Example:

sendaction	::=	**send (** *agentselector* **,** [*moodoperator*] *Poslitconj*) |
		sendonce (*agentselector* **,** [*moodoperator*] *Poslitconj*)
moodoperator	::=	**:** | **!** | **?**
agentselector	::=	*agentexpression* |
		quantor | [*quantor*] |
		[*agentexpression* [**,** *agentexpression*]*]
agentexpression	::=	*agentname* | *variable* | **self**
quantor	::=	**all** | **allother**
agentname	::=	any valid agent name

Figure 6.5: Syntax of the `send` and `sendonce` action

```
send(alice, :hello).

% using the square brackets to address multiple agents for one message
send([alice, bob, charlie], :hello).
```

If the agent name refers to an agent that does not exist in the MAS, or has died, or is otherwise unadressable, the message will silently be sent anyway. There is no feedback confirming or disconfirming that an agent has received the message. Only a reply, the perception of expected behaviour of the receiving agent, or the absence of an expected reply can confirm or disconfirm the reception of the message.

Variables

A variable type agent expression allows a dynamic way of specifying the message recipient. Sometimes the recipient depends on the agent's beliefs or goals or on previous conversations. The variable agent expression consists of a variable in the agent's KR language. If the KR language is Prolog, this means it must start with an uppercase letter. This variable will be resolved when the program rule's mental state condition is evaluated. This means that the mental state condition **must** bind all variables that are used in the agent selector. If an agent selector contains unbound variables at the time of action selection, the action will not be considered for execution.

Example:

```
1  beliefs {
2    agent(john).
3    agent(mary).
4  }
5  goals {
6    informed(john, fact(f)).
7  }
8  main module{
9    program {
10     if bel(agent(X)), goal(hold(gold)), not(bel(sent(_, imp(hold(_))))) then send(X, !hold(gold)).
11
12     goal(informed(Agent, fact(F))) then send(Agent, :fact(F)).
13
14     % This will never be selected:
15     if bel(something) then send(Agent, :something).
16   }
17 }
```

In this example, the program rule on line 9 contains the variable `X`, which has two possible substitutions: `[X/john]` and `[X/mary]`. This results in there being two *options* for the action: `send(john, !hold(gold))` and `send(mary, !hold(gold))`. The agent's action selection engine will select only one option for execution. This means that variables resolve to *one* agent name, and are therefore not suited for multicasting messages.

Quantors

Quantors are a special type of agent expression. They consist of a reserved keyword. There are three possible quantors that can be used in combination with the communicative actions `send` and `sendonce`: `all`, `allother` and `self`. When the `send` action is performed, the quantor is expanded to a set of agent names, in the following way:

- **`all`** will expand to all names of agents currently present in the belief base of the agent (including the name of the sending agent itself).
- **`allother`** will expand to all names of agents currently present in the MAS, with the exception of the sending agent's name.
- **`self`** will resolve to the sending agent's name. So using `self`, an agent can send a message to itself.

Sending a message addressed using a quantor will not result in the quantor being put literally in the mailbox. Rather, the actual agent names that the quantor resolves to are subsituted, and a `sent(..)` fact is inserted for every agent addressed by the quantor. This has consequences for querying the mailbox using quantors. It is possible to test if a message has been sent to `all` agents, for example, by doing

```
forall bel(fact, not(sent(all, fact))) then send(all, fact).
```

This will execute if the message has not been sent to all agents the sending agent believes to exist, so all substitutions of `X` in `bel(agent(X))`. This means that after sending of the original message, if new agents would join the mas, this substitution would change (i.e. `agent(X)` facts would be added). Thus the above mental state condition would again be satisfied, because the message had not been sent to all agents. The semantics of the `all` and `allother` quantors in belief queries reflect the situation *at the time of querying*.

This is illustrated in the following code fragment, in which the `mailbox{...}` section reflects the mailbox contents at this time.

Code listing

```
1  beliefs {
2    agent(maker).
3    agent(grinder).
4    agent(auxilliarygrinder).
5    me(maker).
6
7    % the new agent that just joined the MAS
8    agent(newagent).
9  }
10 mailbox {
11   sent(grinder, imp(have(grounds))).
12   sent(auxilliarygrinder, imp(have(grounds))).
13 }
14 program {
15   % will execute again:
16   forall bel(not(sent(allother, imp(have(grounds))))) then send(allother, !have(grounds)).
17 }
```

6.6.2 The `agent` and `me` facts

In Section 6.6.1 we have seen the use of variables in agent selectors, and how such a variable must be bound in the agent selector. In the example in that section the beliefbase was populated with 2 `agent(..)` facts, holding the names of the agents that agent believes to exist. Using this 'list' of agents, program rules can be constructed that send a message to agents that satisfy some criterium. For example, a way to send a request only to agents that are not busy could be:

```
forall bel(agent(X), not(busy(X))) then send(X, !sweep(floor)) .
```

Note that the `agent(X)` is crucial here, to get a substitution set for `X`, because `not(busy(X))` would not yield a substitution for variable `X`.

The `agent`s and `me`

So the `agent` facts allows us to select a subset of all existing agents dynamically. An advantage of this is that it makes it possible to write 'dynamic' agent programs, meaning we can write *one* GOAL program for a MAS with multiple identical agents.

Let's reiterate the last example snippet:

```
forall bel(agent(X), not(busy(X))) then send(X, !sweep(floor)).
```

This will select all agents which are not busy, and send `!sweep(floor)` to them.

Recall, however, that an `agent` fact is inserted for every existing agent, *including the agent itself*. Consequently, the agent whose program rule is given here, may send `!sweep(floor)` to itself, as it is one of the `agent(X)`s. This may not be the intended behaviour. Suppose the behaviour should be that it only sends this imperative to *other* agents. We cannot use `allother` as agent selector, because, while it excludes the agent itself from the recipient list, it indiscriminately sends the message to *all* other agents, ignoring the selection we made in the mental state condition.

We need another way to distinguish between *other* agents and `this` agent. For this purpose, a special `me(..)` fact is inserted in an agent's belief base upon launch. It specifies the name of the agent. So, taking the example mas from Figure 6.2, after launch of `elevator2`, its belief base consist of the following facts:

```
agent(manager).
agent(elevator1).
agent(elevator2).
agent(elevator3).
me(elevator2).
```

Now the elevator program can include a rule that sends a message to any other elevator agent excluding itself as follows:

```
forall bel(agent(Agt), me(Me), not(Agt=Me), not(Agt=manager)) then send(Agt, !service(somefloor)).
```

The whole point of this is that this program rule works for every elevator agent and so it is not necessary to make a GOAL agent file for each agent in which the agents would be named explicitly.[2]. Also, if the naming scheme or the number of the elevator agents were to be changed, the agent program would not have to be modified; only the mas file would need to be changed.

In the case of the coffee domain agents, it means that the coffee maker and the coffee grinder, which are both machines that can make something out of something, can have very similar programs, sharing action rules for production and capability exploration.

6.7 The Coffee Factory Mas Again

In Section 6.3 the Coffee Factory mas was introduced. In this section the workings of the coffee maker and coffee grinder are analyzed in more detail. We will see how the agents coordinate their actions by communicating in different ways.

[2]In this example the exception is the `manager`, as here we assume this to be a special agent that always has this name. If there were more `manager`s, the belief clause would contain `bel(manager(Mgr), not(Agt=Mgr))`

6.7.1 Capability exploration

The agents know what they can make themselves. This is represented as beliefs in the agent program. For the coffee maker, this look like:

```
beliefs {
  ...
  canMake(maker, [coffee, espresso]).
}
```

To find out what the other agents can make, the following action rules are used in the program:

```
% ask each agent what they can make
forall bel(agent(A), not(me(A)), not(canMake(A, _)), not(sent(A, int(canMake(A, _)))))
   then send(A, ?canMake(A, _)).

% answer any question about what this agent can make
forall bel(me(Me), received(A, int(canMake(Me, _)), canMake(Me, Prods))
  then send(A, :canMake(Me, Prods)) + delete(received(A, int(canMake(Me, _)))).

% process answers from other agents
forall bel(received(Sender, canMake(Sender, Products)))
  then insert(canMake(Sender, Products)) + delete(received(Sender, canMake(Sender, Products))).
```

The first rule checks if there is an agent `A`, other than this agent, for whom this agent does not have any belief of what it can make, and to whom this agent has not already sent an *interrogative* to query it. If this is the case, the rule is applied and an *interrogative* message to ask which products that agent `A` can make is sent. Note that `not(me(A))` prevents `A` being bound to this agent, which would otherwise result in this agent asking itself what it can make. In this situation that would not happen, because `not(canMake(A,_))` has the same effect, since this agent has a belief of what it can make (e.g. `bel(me(Me), canMake(Me, _))` is true). Also recall that after execution of a `send` action, a `sent` fact is inserted in the mailbox.

The second rule handles such incoming interrogatives. It looks in the mailbox for received interrogative messages asking what this agent can make. It replies to the sender with an *indicative* message, indicating what it can make. Also, it removes the received message from the mailbox. This prevents this rule from being triggered repeatedly.

Finally these indicatives are handled in the third rule. The mailbox is queried for received *indicative* messages, containing the information about who makes what. If such a message exists, insert the information as a fact in the belief base. Also, the received message is removed from the mailbox to prevent repeated execution of this program rule for this message.

6.7.2 Production delegation

The coffee maker needs ground beans (grounds) to make coffee, but it cannot grind beans. But once it has found out that the coffee grinder *can* grind beans into coffee grounds, using the above program rules, it can request the grinder to make grounds by sending it an *imperative* message. This is represented more generically in the following action rule:

```
% When we cannot make some product, try to find a maker for it
forall goal(have(P)), bel(requiredFor(P, R), not(have(R))), bel(canMakeIt(Maker, R), not(me(Maker)))
   then send(Maker, !have(R)).
```

When this agent has a goal to make some product `P` for which it needs a requirement `R` which it doesn't have, and it knows of a maker of `R`, it sends an imperative message to that maker. The message content is `!have(R)` (the `R` will be bound to some product at this point), indicating that this agent has a goal to `have R`.

When such an imperative message is received by an agent and it can make the requested product, it can adopt a goal to make it so:

```
forall bel(received(A, imp(have(A, P))), me(Me), canMakeIt(Me, P))
  then adopt(have(A, P)).
```

Note that we did not remove the message from the mailbox. This is because this agent needs a record of who requested what. If we would remove the message, the information that an agent requested a product would have to be persisted by some other means.

Status updates

Once a product has been made for some other agent that requires it, that agent should be informed that the required product is ready. Agents in the Coffee Domain do not 'give' each other products or perceive that products are available, so they rely on communication to inform each other about that.

```
forall bel(received(A, imp(have(P))), have(P))
  then send(A, :have(P)) + delete(received(A, imp(have(P)))).
```

Now we *do* remove the received message, because we have completely handled the case.

On the receiving side of this message, reception of such an indicative message `:have(P)` does not automatically result in the belief by this agent that `have(P)` is true. This insertion of the belief must be done explicitly.[3]

```
% update beliefs with those of others (believe what they believe)
forall bel(received(A, have(P)))
  then insert(have(P)) + delete(received(A, have(P))).
```

Pro-active inform

At any time, it may be the case that an agent sees an opportunity to inform another agent about some fact if it thinks this agent would want to know that, without being asked. This may happen if it believes the other agent has some goal but it believes that this goal has already been achieved. It can then help the other agent by sending an indicative message that the goal state is achieved.

In the Coffee Factory mas this situation can occur if one machine believes that another machine needs some product. If the agent has that product available, then it will inform the other agent of that fact:

```
% if some machine seems to need a product, tell it we have it
forall bel(received(Machine, imp(have(X))), bel(have(X), not(sent(Machine, have(X))))
   then send(Machine, :have(X)).
```

6.8 Notes

As noted in Chapter 1, additional concepts may be introduced to structure and design multi-agent systems. The idea is that by imposing organizational structure on a multi-agent system specific coordination mechanisms can be specified. Imposing an organizational structure onto a multi-agent system is viewed by some as potentially reducing the autonomy of agents based on a perceived tension between individual autonomy and compliance with constraints imposed by organizations. That is, in the view of [14], an organization may restrict the actions permitted, which would have an immediate impact on the autonomy of agents.

The "mailbox semantics" of GOAL is very similar to the communication semantics of 2APL [19]. Providing a formal semantics of communication has received some attention in agent-oriented

[3] This is where we make a leap of faith. The other agent indicated *its* belief in `have(P)`. The only reason we copy this belief is because we trust that other agent.

programming research. Some agent programming languages use middleware infrastructures such as JADE [8], which aims to comply with the communication semantics of FIPA and related standards. The FIPA standard introduced many primitive notions of agent communication called *speech acts*. There are many different speech act types, however, which may vary for different platforms. In practice, this variety of options may actually complicate developing agent programs more than that it facilitates the task of writing good agent programs. We therefore think it makes sense to restrict the set of communication primitives provided by an agent programming language. In this respect we favor the approach taken by *Jason* which limits the set of communication primitives to a core set. In contrast with *Jason*, we have preferred a set of primitives that allows communication of declarative content only, in line with our aim to provide an agent programming language that facilitates declarative programming.

6.9 Exercises

6.9.1 Milk cow

The coffee domain example from Section 6.3 has a coffee maker and a coffee grinder. Suppose we now also want to make lattes. A latte is coffee with milk. To provide the milk, a cow joins the scene. The cow is empathic enough that it makes milk whenever it believes that someone needs it.

1. Expand the list of products the coffee maker can make with `latte`.

2. Add the knowledge that `latte` requires `coffee` and `milk` to that of the coffee maker.

3. Write a new agent called `milkcow.goal` which has the following properties:

   (a) It has no knowledge, beliefs or goals.[4]

   (b) It does not do capability exploration, but it does answer other agent's questions about what it `canMake`.

   (c) When it notices that another agent needs milk, it will make the milk resulting in the cow's belief that `have(milk)`.

   (d) When it notices that another agent needs milk and the cow has milk, it will notify that agent of that fact.

[4] as far as humans can tell.

6.10 Appendix

```
% This agent represents the coffee machine. It's function is to supply a user
% with nice steaming fresh cups of coffee. It knows how to make coffee and
% espresso. It will communicate to find out who can make what. Notice that the
% program and perceptrules sections contain no literals.

% In fact, the program,perceptrules and actionspec implement a machine capable
% of making certain products, if it has all required ingredients, and finding
% producers of ingredients it cannot make itself.

main: coffeeMaker {
    knowledge{
        requiredFor(coffee, water). requiredFor(coffee, grounds).
        requiredFor(espresso, coffee). requiredFor(grounds, beans).
        canMakeIt(M, P) :- canMake(M, Prods), member(P, Prods).
    }
    beliefs{
        have(water). have(beans).
        canMake(maker, [coffee, espresso]).
    }
    goals{
        have(latte).
    }
    main module{
      program {
        % if we want to make something, then make it (the action's precondition
        % checks if we have what it takes, literally)
        if goal(have(P)) then make(P).
      }
    }
    event module{
      program{
        % ask each agent what they can make
        forall bel(agent(A), not(me(A)), not(canMake(A, _))) then sendonce(A, ?canMake(A, _)).
        % answer any question about what this agent can make
        forall bel(me(Me), received(A, int(canMake(Me, _))), canMake(Me, Prod))
           then sendonce(A, :canMake(Me, Prod)).
        % process answers from other agents
        forall bel(received(Sender, canMake(Sender, Products)))
           then insert(canMake(Sender, Products)) + delete(received(Sender, canMake(Sender, Products))) .

        % update beliefs with those of others (believe what they believe)
        forall bel(agent(A), received(A, have(X))), not(bel(have(X))) then insert(have(X)).

        % If we need some ingredient, see if we can make it ourselves
        forall goal(have(P)), bel(requiredFor(P, R), not(have(R))),
           bel(canMakeIt(Me, R), me(Me)) then adopt(have(R)).
        % else try to find a maker for it
        forall goal(have(P)), bel(requiredFor(P, R), not(have(R))),
           bel(canMakeIt(Maker, R), not(me(Maker))) then sendonce(Maker, !have(R)).
        % else just broadcast our need
        forall goal(have(P)), bel(requiredFor(P, R), not(have(R)), me(Me), not(canMakeIt(Me, P)))
           then sendonce(allother, !have(R)).

        % if some machine seems to need a product, tell it we have it
        forall bel(agent(Machine), received(Machine, imp(have(X))), have(X))
            then sendonce(Machine, :have(X)).
      }
    }
    actionspec {
        make(Prod) {
            pre { forall(requiredFor(Prod, Req), have(Req)) }
            post { have(Prod) }
        }
    }
}
```

Figure 6.6: The Coffee Maker Agent

```
% The Coffee Grinder is an agent capable of grinding coffee beans into grounds.
% For making grounds it needs coffee beans. Whenever it needs beans it will
% announce as much by sending an imperative "!have(beans)" to allother agents.

main: coffeegrinder {
    knowledge{
        requiredFor(grounds, beans).
        canMakeIt(M, P) :- canMake(M, Prods), member(P, Prods).
    }
    beliefs{
        canMake(grinder, [grounds]).
    }
    main module{
      program {
        % if we need to make something, then make it (the action's precondition
        % checks if we have what it takes, literally)
        if goal(have(P)) then make(P).
      }
    }
    event module{
      program{
        % capability exploration:

        % ask each agent what they can make
        forall bel(agent(A), not(me(A)), not(canMake(A, _)))
          then sendonce(A, ?canMake(A, _)).
        % answer any question about what this agent can make
        forall bel(me(Me), received(A, int(canMake(Me, _))), canMake(Me, Prod))
          then sendonce(A, :canMake(Me, Prod)).
        % process answers from other agents
        forall bel(received(Sender, canMake(Sender, Products)))
          then insert(canMake(Sender, Products))
          + delete(received(Sender, canMake(Sender, Products))).

        % update beliefs with those of others (believe what they believe)
        forall bel(agent(A), received(A, have(X))), not(bel(have(X))) then insert(have(X)).

        % if we want to make grounds, but have no beans, announce that we need beans
        forall goal(have(P)), bel(requiredFor(P, R), not(have(R)), me(Me),not(canMakeIt(Me, P)))
           then sendonce(allother, !have(R)).

        % if some agent needs grounds, then adopt the goal to make it
        forall bel(received(_, imp(have(grounds))))
           then adopt(have(grounds)) + delete(received(_, imp(have(grounds)))).

        % if some machine seems to need a product, tell it we have it
        forall bel(agent(Machine), received(Machine, imp(have(X))),
           have(X)) then sendonce(Machine, :have(X)).
      }
    }
    actionspec{
        make(Prod) {
            pre { forall(requiredFor(Prod, Req), have(Req)) }
            post { have(Prod) }
        }
    }
}
```

Figure 6.7: The Coffee Grinder Agent

Chapter 7

Design of Agent Programs

This chapter aims to provide some *guidelines* for writing an agent program in GOAL. Although writing an agent program typically will strongly depend on the application or environment that the agent is expected to operate in, some general guidelines may still be given that help writing correct and elegant agent programs.

We also discuss how to structure and reuse parts of an agent program by means of *importing* module and other files. Using the possibility to distribute agent code over different files facilitates a more structured approach to programming a multi-agent system. A mas developer, however, needs to take care that these different files that make up the multi-agent system do not conflict with each other and we discuss some of the issues here.

7.1 Design Steps: Overview

As writing action rules and action specifications requires that the predicates used to describe an environment are available, it generally is a good idea to start with designing a representation that may be used to describe the environment. This advice is in line with the emphasis put on the analysis of the environment that an agent acts in as discussed in Chapter 5.

Generally speaking, it is important to first understand the environment. An environment provides a good starting point as it determines which actions agents can perform and which percepts agents will receive. In this early phase of development, it is important to create an initial design of how to represent the environment logic, how to keep track of changes in the environment, and how to represent goals the agent should set (see also Section 7.2.2 below). The result of this analysis should be an initial design of an *ontology* for representing the agent's environment.

We first introduce a generic approach for designing a GOAL multi-agent system that consists of a number of high-level steps that should be part of any sound code development plan for a mas.

1. **Ontology Design**
   (a) Identify percepts
   (b) Identify environment actions
   (c) Design an ontology to represent the agent's environment.
   (d) Identify the goals of agents
2. **Strategy Design**
   (a) Write percept rules
   (b) Write action specifications
   (c) Determine action selection strategy
   (d) Write action (selection) rules

Of course, it should be kept in mind that the steps that are part of this plan provide a rough guideline and in practice one may wish to deviate from the order and, most likely, one may need to reiterate several steps.

The first part of this approach is probably the most important to get right. At the same time it is important to realise that it is nearly impossible to get the design of an ontology right the first time. Ontology design means designing the predicates (i.e. labels) that will be used to represent and keep track of the agent's enviroment, to set the agent's goals, and to decide which actions the agent should perform.

It is impossible to create an adequate ontology without a proper understanding of the environment which explains the first two steps that are part of ontology design. More generally, in these steps a programmer should gain proper knowledge of the environment. As a general guideline, it is best to start introducing predicates that will be used to represent the agent's environment and will be part of the knowledge and belief base of the agent to keep track of what is the case in that environment. Although in this phase it is useful to identify the goals an agent may adopt, the actual code for managing goals typically consists of action rules that are written as part of the strategy design phase. The main purpose of identifying goals in the ontology design phase, however, is to *check* whether the ontology supports expressing the goals an agent will adopt. *It should never be the case that special predicates are introduced that are used only for representing goals.*

7.2 Guidelines for Designing an Ontology

A key step in the development of a GOAL agent is the design of the domain knowledge, the concepts needed to represent the agent's environment in its knowledge, beliefs and the goals of the agent.

An important and distinguishing feature of the GOAL language is that it allows for specifying both the beliefs and goals of the agent *declaratively*. That is, both beliefs and goals specify *what* is the case respectively *what* is desired, not *how* to achieve a goal. GOAL agents are goal-directed which is a unique feature of the GOAL language. The main task of a programmer is making sure that GOAL agents are provided with the right domain knowledge required to achieve their goals. More concretely, this means writing the knowledge and goals using some declarative knowledge representation technology and writing action rules that provide the agent with the knowledge when it is reasonable to choose a particular action to achieve a goal.

Although the GOAL language assumes some knowledge representation technology is present, it is not committed to any particular knowledge representation technology. In principle any choice of technology that allows for declaratively specifying an agents beliefs and goals can be used. For example, technologies such as SQL databases, expert systems, Prolog, and PDDL (a declarative language used in planners extending ADL) can all be used. Currently the implementation of the GOAL interpreter however only supports Prolog. *We assume the reader is familiar with Prolog and we refer for more information about Prolog to [10] or [57].* GOAL uses SWI Prolog; for a reference manual of SWI Prolog see [59].

7.2.1 Prolog as a Knowledge Representation Language

There are a number of things that need to be kept in mind when using Prolog to represent an agent's environment in GOAL.

A first guideline is that it is best to *avoid the use of the don't care symbol "_" in mental state conditions.* The don't care symbol can be used without problems elsewhere and can be used without problems within the scope of a **bel** operator, but in particular cannot be used within the scope of a goal-related operator such as the achievement goal operator **a-goal** or the goal achieved operator **goal-a**.[1]

[1] The reason why a don't care symbol _ cannot be used within the scope of the **a-goal** and **goal-a** operators is that these operators are defined as a conjunction of two mental atoms and we need variables to ensure answers for both atoms are related properly. Recall that **a-goal**(φ) is defined by **goal**(φ), **not**(**bel**(φ)). We can illustrate

Second, it is important to understand that only a subset of all Prolog built-in predicates can be used within a GOAL agent program. A list of operators that can be used is provided with the distribution of GOAL.

Third, it is important to realise that GOAL uses a number of special predicates that have a special meaning. These special predicates are reserved by GOAL and should *not* be defined in the agent program in , for example, the **knowledge** section. These predicates include `percept(_,_)`, `me(_)`, `agent(_)`, `sent(_,_)`, and `received(_,_)`.

Finally, we comment on a subtle difference between Prolog itself and Prolog used within the context of a GOAL agent program. The difference concerns duplication of facts within Prolog. Whereas in Prolog it is possible to duplicate facts, and, as a result, obtain the same answer (i.e. substitution) more than once for a specific query, this is not possible within GOAL. The reason is that GOAL assumes an agent uses a *theory* which is a *set* of clauses, and not a database of clauses as in the Prolog ISO sense (which allows for multiple occurrences of a clause in a database).

7.2.2 The Knowledge, Beliefs, and Goals Sections

The design of the **knowledge**, **beliefs**, and **goals** sections in an agent program is best approached by the main *function* of each of these sections. These functions are:

- **knowledge** section: represent the *environment or domain logic*
- **beliefs** section: represent the *current and actual* state of the environment
- **goals** section: represent what the agent wants, i.e. the *desired* state of the environment

Useful concepts to represent and reason about the environment or domain the agent is dealing with usually should be defined in the **knowledge** section. Examples are definitions of the concepts of `tower` in the Blocks World or `wumpusNotAt` in the Wumpus World.

Use the **beliefs** section to keep track of the things that change due to e.g. actions performed or the presence of other agents. A typical example is keeping track of the position of the entity that is controlled using e.g. a predicate *at*. Although logical rules are allowed in the **beliefs** section, it is better practice to include such rules in the **knowledge** section.

It is often tempting to define the logic of some of the goals the agent should pursue in the **knowledge** section instead of the **goals** section. This temptation should be resisted, however. Predicates like `priority` or `needItem` thus should not be used in the **knowledge** section. It is better practice to put goals to have a weapon or kill e.g. the Wumpus in the goal base. One benefit of doing so is that these goals automatically disappear when they have been achieved and no additional code is needed to keep track of goal achievement. Of course, it may be useful to code some of the concepts needed to define a goal in the knowledge base of the agent.

It is, moreover, better practice to insert *declarative* goals that denote a *state* the agent wants to achieve into the goal base of the agent than predicates that start with verbs. For example, instead of `killWumpus` adopt a goal such as `wumpusIsDead`.

To summarize the discussion above, the main guideline for designing a good ontology is:

> Use predicate labels that are *declarative*.

In somewhat other words, this guideline advises to introduce predicates that *describe* a particular state and denote a particular *fact*. Good examples of descriptive predicates include predicates such as `at(_,_,_)` which is used to represent where a particular entity is located and a predicate such as `have(Item)` which is used to represent that an entity has a particular item. Descriptive predicates are particularly useful for representing the facts that hold.

what goes wrong by instantiating φ with e.g. `on(X,_)`. Now suppose that `on(a,b)` is the *only* goal of the agent to put some block on top of another block and the agent believes that `on(a,c)` is the case. Clearly, we would expect to be able to conclude that the agent has an achievement goal to put `a` on top of `b`. And, as expected, the mental state condition **`a-goal`**`(on(X,Y))` has `X=a, Y=b` as answer. The reader is invited to check, however, that the condition **`a-goal`**`(on(X,_)` does not hold!

In contrast, labels that start with verbs such as `getFlag` are better avoided. Using such labels often invites duplication of labels. That is, a label such as `getFlag` is used to represent the activity of getting the flag and another label such as `haveFlag` then might be introduced to represent the end result of the activity. Instead, by using the `have(Item)` predicate, the goal to get the flag can be represented by adopting `have(flag)` and the result of having the flag can be represented by inserting `have(flag)` into the belief base of the agent.

Check for and Remove Redundant Predicates

The GOAL IDE automatically provides information about the use or lack of use of predicates in an agent program when the program is saved. This information is provided in the Parse Info tab, see the User Manual.

It is important to check this feedback and remove any redundant predicates that are never really used by the agent. Cleaning your code in this way increases readability and decreases the risk of introducing bugs.

In particular make sure that you do not introduce *abstract goals* in the agent's goal base like *win* which represent overall goals that are not used or cannot be achieved by executing a specific plan. Instead, describe such more abstract and global goals in comments in the agent file.

7.3 Action Specifications

Actions that the agent can perform in an environment must be specified in an action specification section. Actions that have not been specified cannot be used by the agent.

7.3.1 Put Action Specifications in the init Module

Action specifications should be located in the **actionspec** section in the **init** module of the agent. By putting action specifications in this module they become globally available throughout the agent program in all other modules as well. Action specifications that are not specified in the **init** module are not available in other modules. Another benefit of always putting action specifications within the **init** module is that other programmers know where to look for these specifications.

7.3.2 Action Specifications Should Match with the Environment Action

An action specification consists of a *precondition* of the action and a *postcondition*. The precondition should specify *when the action can be performed in the environment*. These conditions should match with the actual conditions under which the action can be performed in the environment. This means that an action precondition should be set to `true` *only if* the action can *always* be performed. It also means that a precondition should *not* include conditions that are more restrictive than those imposed by the environment. For example, the precondition of the `forward` action in the Wumpus World should not have a condition that there is no pit in front of the agent; even though this is highly undesirable, the environment allows an agent to step into a pit... Conditions *when* to perform an action should be part of the action rule(s) that select the action.

Do *not* specify the `forward` action in the Wumpus World as follows:

```
forward{
  pre{ orientation(Dir), position(Xc, Yc), inFrontOf(Xc, Yc, Dir, X, Y),
       pitNotAt(X, Y), wumpusNotAt(X,Y), not(wall(X, Y))         % ????
  }
  post{ not(position(Xc, Yc)), position(X,Y) }
}
```

Instead, provide e.g. the following specification:

```
forward{
  pre{ orientation(Dir), position(Xc, Yc), inFrontOf(Xc, Yc, Dir, X, Y) }
  post{ not(position(Xc, Yc)), position(X,Y) }
}
```

7.3.3 Do NOT Introduce Own Action Specifications

The main reason to *not* introduce action specifications for actions that are not available in the environment is that *there is no reason to do so*. Any action that can be specified can also be programmed using the built-in **insert** and **delete** actions of GOAL. Moreover, actions that are not built-in actions of GOAL are sent to the environment for execution. If the environment does not recognize an action, it will start throwing exceptions. If you feel the need to introduce action labels for representing a complex action that changes the beliefs of the agent, then as an alternative instead consider to comment your code.

7.4 Readability of Your Code

GOAL is an agent oriented programming languages based on the idea of using common sense notions to structure and design agent programs. It has been designed to support common concepts such as beliefs and goals, which allow a developer to specify a *reason* for performing an action. One motivation for this style of programming stems from the fact that these notions are closer to our own common sense intuitions. As such, a more intuitive programming style may result which is based on these notions.

Even though such common sense concepts are useful for designing and structuring programs, this does not mean that these programs are always easy to understand or are easy to "read" for developers other than the person who wrote the code. Of course, after some time not having looked at a particular piece of code that code may become even difficult to understand for its own developer.

There are various ways, however, that GOAL supports creating programs that are more easy to read and understand. As in any programming language, it is possible to *document* code by means of *comments*. As is well-known documentation is very important to be able to maintain code, identify bugs, etc and this is true for GOAL agent programs as well. A second method for creating more accessible code is provided by *macros*. Macros provide a tool for introducing intuitive labels for mental state conditions and to use these macros instead of the mental state conditions in the code itself. A third method for creating more readable code is to properly structure code and adhere to various patterns that we discuss at other places in this chapter.

7.4.1 Document Your Code: Add Comments!

Code that has not been documented properly is typically very hard to understand by other programmers. You will probably also have a hard time understanding some of your own code when you have not recently looked at it. It therefore is common practice and well-advised to *add comments to your code* to explain the logic. This advice is not specific to GOAL but applies more generically to any programming language. **A comment is created simply by using the `%` symbol at the start of a code line.**

There are some specific guidelines that can be given to add comments to a GOAL program, however. These guidelines consist of typical locations in an agent program where code comments should be inserted. These locations include among others:

- just before a definition of a predicate in the **knowledge** section a comment should be inserted explaining the *meaning of the predicate*,
- just before an action specification a comment may be introduced to explain the informal pre- and postconditions of the action (as specified in a manual for an environment, for example; doing so allows others to check your specifications),
- just before a module a comment should explain the *purpose or role of the module*,
- just before specific groups of rules a comment should explain the *purpose of these rules*.

Of course, additional comments, e.g. at the beginning of an agent file or elsewhere, may be added to clarify the agent code.

7.4.2 Introduce Intuitive Labels: Macros

Action rules are used by an agent to select an action to perform. Writing action rules therefore means providing *good reasons* for performing the action. These reasons are captured or represented by means of the mental state conditions of the action rules. Sometimes this means you need to write quite complicated conditions. Macros can be used to introduce *intuitive labels* for complex mental state conditions and are used in action rules to enhance the readability and understandability of code. A macro thus can be used to replace a mental state condition in an action rule by a more intuitive label.

An example of a macro definition that introduces the label `constructiveMove(_,_)` is:

```
#define constructiveMove(X, Y)
   a-goal( tower([X, Y | T]) ), bel( tower([Y | T]), clear(Y), (clear(X) ; holding(X)) ) .
```

Macros should be placed at the beginning of ***program*** *sections, just before the action rules.* They may also be placed just before modules, but usually it is better to place them at the beginning of a program section.

7.5 Structuring Your Code

GOAL provides various options to structure your code. A key feature for structuring code are *modules* discussed in Chapter **??**. Another approach to structuring code is to *group similar action rules together*. It is often useful to group action rules that belong together in a module. An example design guideline, for example, is to put all action rules which select environment actions in the **main** module. One other important tool to organize code is provided by the possibility to distribute code over different files using the `#import` command.

7.5.1 Group Rules of Similar Type

When writing an agent program in GOAL one typically has to write a number of different types of rules. We have seen various examples of rule types in the previous chapters. We list a number of rule types that often are used in agent programs and provide various guidelines for structuring code that uses these different rules.

These guidelines are designed to improve understandability of agent programs. An important benefit of using these guidelines in practice in a team of agent programmers is that each of the programmers then is able to more easily locate various rules in an agent program. Structuring code according to the guidelines facilitates other agent programmers in understanding the logic of the code.

Action Rules

Although we use the label *action rule* generically, it is sometimes useful to reserve this label for specific rules that select *environment actions* instead of rules that only select built-in actions. As a general design guideline, *action rules should be placed inside the **main** module or in modules called from that module.*

Percept Rules

An action rule is a *percept rule* if its mental state condition inspects the percept base of the agent and *only* updates the mental state of the agent. That is, if the rule has a belief condition that uses the `percept/1` predicate and only selects actions that modify the mental state of the agent, that rule is a percept rule. A percept rule, moreover, should be a **`forall...do...`** rule. Using this type of rule ensures that *all* percepts of a specific form are processed. Of course, it remains up to the agent programmer to make sure that all different percepts that an environment provides are handled somehow by the agent.

As a design guideline, *percept rules should be placed inside the **event** module.* It is also best practice to put percept rules *at the beginning of this module.* It is important to maintain a state that is as up-to-date as possible, which is achieved by first processing any percepts when the **event** module is executed. For this reason, it best to avoid rules that generate environment actions based upon inspecting the agent's percept base. Performing environment actions would introduce new changes in the environment and this may make it hard to update the mental state of the agent such that it matches the environment's state. Of course, percept rules can also be put in other modules that are called from the **event** module again but this usually does not introduce a significant benefit.

Note that there is one other place where it makes sense to put percept rules: in the **init** module. Percept rules placed inside the **init** module are executed only once, when the agent is launched. Percept rules in this module can be used to process percepts that are provided only once when the agent is created.

Communication Rules

There are various types of action rules that can be classified as communication rules. An action rule is a *communication rule* if its mental state condition inspects the mailbox of the agent. That is, if the rule has a belief condition that uses the `received/1` or `sent/1` predicate, that rule is a communication rule. An action rule that selects a communicative action such as `send` also is communication rule.

As a design guideline, *communication rules that only update the mental state of the agent should be placed directly after percept rules.* What is the rationale for this? Percepts usually provide more accurate information than messages. It therefore always makes sense to first process perceptual information in order to be able to check message content against perceptual information.

Goal Management Rules

Rules that modify the goal base by means of the built-in **adopt** and **drop** actions are called *goal management rules.* These rules are best placed at the end of the **program** section in the **event** module, or possibly in a module that is called there.

Goal management rules have two main functions, i.e. they should be used:

- Drop goals that are no longer considered useful or feasible,
- Adopt goals that the agent should pursue given the current circumstances.

Both types of rules are best ordered as above, i.e. first list rules that drop goals and thereafter introduce rules for adopting goals. By clearly separating and ordering these rules this way, it is most transparant for which *reasons* goals are dropped and for which reasons goals are adopted.

An agent program should *not* have rules that check whether a goal has been achieved. The main mechanism for removing goals that have been achieved is based on the beliefs of the agent. GOAL automatically checks whether an agent believes that a goal has been achieved and removes it from the agent's goal base if that is the case. It is up to you of course to make sure the agent will believe it has achieved a goal if that is the case. By defining the ontology used by the agent in the right way this should typically be handled more or less automatically.

Other Rule Types

The rules types that we discussed above assume that rules serve a single purpose. As a rule of thumb, it is good practice to keep rules as simple as possible and use rules that only serve a single purpose as this makes it more easy to understand what the agent will do. It will often, however, also be useful to combine multiple purposes into a single rule. An good example is a rule that informs another agent that a particular action is performed by the agent when performing that action. Such a rule is of the form **if...then** <envaction> + <comaction> that generates options where first an environment action is performed and immediately thereafter a communicative action is performed to inform another agent that an action has been performed. Another example of a similar rule is a goal management rule of the form **if...then** <adoptaction> + <comaction> that informs other agents that the agent has adopted a particular goal (which may imply that other agents can focus their resources on other things). Rules that *only* add such communicative actions to keep other agents up-to-date are best located at the places suggested above; i.e., one should expect to find a goal management rule that informs other agents as well at the end of the **event** module.

How NOT to Use Action Rules

The action rules of GOAL are very generic and can be used to do pretty much anything. Their main purpose, however, is to *select actions and define an action selection strategy to handle events and to control the environment*. Action rules should not be used to code some of the basic *conceptual knowledge* that the agent needs. The knowledge representation language, e.g. Prolog, should be used for this purpose. For example, you should *not* use an action rule to insert into an agent's belief base that the Wumpus is not at a particular position; instead, the agent should use logic to derive such facts.

Do *not* use a rule for inserting that the Wumpus is not at a particular location, e.g. the following code should be avoided:

```
forall bel( (not(percept(stench)); not(wumpusIsAlive)),
        position(X, Y), adjacent(X,Y,Xadj,Yadj),
        not(wumpusNotAt(Xadj,Yadj)) )
do insert( wumpusNotAt(Xadj,Yadj) ).
```

Instead, use a definition of the concept using several logical rules such as:

```
wumpusNotAt(X,Y) :- visited(X,Y).
wumpusNotAt(X,Y) :- ...
```

7.5.2 Importing Code

GOAL provides the `#import` command to import code from other files. Two types of imports are supported: Importing modules and importing knowledge and beliefs.

Importing Knowledge

The use of the `#import` command is straightforward. Use `#import "<filename>.pl"` to import a Prolog file. A Prolog file should be plain Prolog and not include any GOAL constructs.

In particular, the file should not include the **knowledge** or **beliefs** keywords to designate a section of a GOAL agent program. The `#import` command for importing knowledge and or beliefs should be located *inside* either a **knowledge** or **beliefs** section.

Importing Modules

Use `#import "<filename>.mod2g"` to import a GOAL module. In this case, the entire module including the **module** keyword should be part of the imported file. It is possible to include macros as well in a module file at the beginning of the file. These macros are available to the module in the file as well as to any other modules that are placed after the import command.

Using Imported Files

The option to import files facilitates structuring code and distributing coding tasks among team members. Distributing code parts and/or coding tasks over multiple files requires coordination of these tasks in the following sense. First, it is important to specify which predicates are used in the agent code. This can be done by maintaining an explicit ontology specification. Second, it is important to understand which code parts are put in which file. If you are working in a team, that means you need to *communicate among your team members which code parts are distributed over different files.* Third, when working in a team it is important to communicate which person has a lock on a code file. The programmer that has the lock on a file is the only person that should change the file. Of course, using an svn repository supports some of these functions such as locking a file or merging code that has been changed by different team members.

7.6 Notes

The guidelines discussed in this chapter are based in part on research reported in [49, 36].

Bibliography

[1] Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Pres (2003)

[2] Barandiaran, X., Paolo, E.D., Rohde, M.: Defining agency: individuality, normativity, asymmetry and spatio-temporality in action. Adaptive Behavior **17**(5), 367–386 (2009)

[3] Beer, R.D.: A dynamical systems perspective on agent-environment interaction. AI (??)

[4] Behrens, T., Hindriks, K.V., Dix, J.: Towards an environment interface standard for agent platforms. Annals of Mathematics and Artificial Intelligence pp. 1–35 (2010). URL `http://dx.doi.org/10.1007/s10472-010-9215-9`. 10.1007/s10472-010-9215-9

[5] Behrens, T.M., Dix, J., Hindriks, K.V.: Towards an environment interface standard for agent-oriented programming. Tech. Rep. IfI-09-09, Clausthal University (2009)

[6] Bekey, G.A.: Autonomous Robots: From Biological Inspiration to Implementation and Control. Cambridge, Mass.: The MIT Press (2005)

[7] Bekey, G.A., Agah, A.: Software architectures for agents in colonies. In: Lessons Learned from Implemented Software Architectures for Physical Agents: Papers from the 1995 Spring Symposium. Technical Report SS-95-02, pp. 24–28 (1995)

[8] Bellifemine, F., Caire, G., Greenwood, D. (eds.): Developing Multi-Agent Systems with JADE. No. 15 in Agent Technology. John Wiley & Sons, Ltd. (2007)

[9] Ben-Ari, M.: Principles of the Spin Model Checker. Springer (2007)

[10] Blackburn, P., Bos, J., Striegnitz, K.: Learn Prolog Now!, *Texts in Computing*, vol. 7. College Publications (2006)

[11] de Boer, F., Hindriks, K., van der Hoek, W., Meyer, J.J.: A Verification Framework for Agent Programming with Declarative Goals. Journal of Applied Logic **5**(2), 277–302 (2007)

[12] Bordini, R.H., Hbner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak using Jason. John Wiley & Sons (2007)

[13] Boutilier, C.: A unified model of qualitative belief change: A dynamical systems perspective. Artificial Intelligence **1-2**, 281–316 (1998)

[14] Bradshaw, J., Feltovich, P., Jung, H., Kulkarni, S., Taysom, W., Uszok, A.: Dimensions of adjustable autonomy and mixed-initiative interaction. In: Autonomy 2003, *LNAI*, vol. 2969, pp. 17–39 (2004)

[15] Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about datalog (and never dared to ask). IEEE Trans. of KDE **1**(1) (1989)

[16] Chandy, K.M., Misra, J.: Parallel Program Design. Addison-Wesley (1988)

[17] Cohen, P.R., Levesque, H.J.: Intention Is Choice with Commitment. Artificial Intelligence **42**, 213–261 (1990)

[18] Cook, S., Liu, Y.: A Complete Axiomatization for Blocks World. Journal of Logic and Computation **13**(4), 581–594 (2003)

[19] Dastani, M.: 2apl: a practical agent programming language. Journal Autonomous Agents and Multi-Agent Systems **16**(3), 214–248 (2008)

[20] Dastani, M., Hindriks, K.V., Novak, P., Tinnemeier, N.A.: Combining multiple knowledge representation technologies into agent programming languages. In: Proceedings of the International Workshop on Declarative Agent Languages and Theories (DALT'08) (2008). To appear

[21] Dennett, D.C.: The Intentional Stance, 8 edn. The MIT Press (2002)

[22] Doyle, J.: Philosophy and AI: Essays at the Interface, chap. The Foundations of Psychology: A Logico-Computational Inquiry into the Concept of Mind, pp. 39–78. The MIT Press (1991)

[23] Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press (1995)

[24] Forguson, L.: Common Sense. Routledge (1989)

[25] Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann (2004)

[26] Goertzel, B., Pascal, H., Hutter, M. (eds.): Proceedings of the Second Conference on Artificial General Intelligence (2009)

[27] Grice, H.: Meaning. Philosophical Review **66**, 377–88 (1957)

[28] Grice, H.: Utterer's meaning and intentions. Philosophical Review **78**, 147–77 (1969)

[29] Gupta, N., Nau, D.S.: On the Complexity of Blocks-World Planning. Artificial Intelligence **56**(2-3), 223–254 (1992)

[30] Hindriks, K.: Modules as policy-based intentions: Modular agent programming in goal. In: Proceedings of the International Workshop on Programming Multi-Agent Systems (ProMAS'07), vol. 4908 (2008)

[31] Hindriks, K., van der Hoek, W.: Goal agents instantiate intention logic. In: Proceedings of the 11th European Conference on Logics in Artificial Intelligence (JELIA'08), pp. 232–244 (2008)

[32] Hindriks, K., Jonker, C., Pasman, W.: Exploring heuristic action selection in agent programming. In: Proceedings of the International Workshop on Programming Multi-Agent Systems (ProMAS'08) (2008)

[33] Hindriks, K.V., de Boer, F.S., van der Hoek, W., Meyer, J.J.C.: Agent Programming in 3APL. Autonomous Agents and Multi-Agent Systems **2**(4), 357–401 (1999)

[34] Hindriks, K.V., de Boer, F.S., van der Hoek, W., Meyer, J.J.C.: Agent Programming with Declarative Goals. In: Proceedings of the 7th International Workshop on Agent Theories Architectures and Languages, *LNCS*, vol. 1986, pp. 228–243 (2000)

[35] Hindriks, K.V., van Riemsdijk, M.B.: A computational semantics for communicating rational agents based on mental models. In: Proceedings of the 6th International Conference on Programming Agents and Multi-Agent Systems (2009)

[36] Hindriks, K.V., van Riemsdijk, M.B., Jonker, C.M.: An empirical study of patterns in agent programs. In: Proceedings of PRIMA'10 (2011)

[37] van der Hoek, W., van Linder, B., Meyer, J.J.: An Integrated Modal Approach to Rational Agents. In: M. Wooldridge (ed.) Foundations of Rational Agency, Applied Logic Series 14, pp. 133–168. Kluwer, Dordrecht (1999)

[38] Lifschitz, V.: On the semantics of strips. In: M. Georgeff, A. Lansky (eds.) Reasoning about Actions and Plans, pp. 1–9. Morgan Kaufman (1986)

[39] Luck, M., d'Inverno, M.: Motivated behaviour for goal adoption. In: Multi-Agent Systems: Theories, Languages and Applications - 4th Australian Workshop on Distributed Artificial Intelligence, pp. 58–73 (1998)

[40] McCarthy, J.: Programs with common sense. In: Proceedings of the Teddington Conference on the Mechanization of Thought Processes, pp. 75–91. Her Majesty's Stationary Office, London (1959)

[41] McCarthy, J.: Ascribing mental qualities to machines. Tech. rep., Stanford AI Lab, Stanford, CA (1979)

[42] Meyer, J.J.C., van der Hoek, W.: Epistemic Logic for AI and Computer Science. Cambridge: Cambridge University Press (1995)

[43] Newell, A., Simon, H.A.: GPS, a program that simulates human thought. In: E. Feigenbaum, J. Feldman (eds.) Computers and Thought. New York: McGraw-Hill (1963)

[44] Omicini, A., Ricci, A., Viroli, M., Castelfranchi, C., Tummolini, L.: Coordination artifacts: Environment-based coordination for intelligent agents. In: Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'04) (2004)

[45] Padgham, L., Lambrix, P.: Agent capabilities: Extending bdi theory. In: Proc. of the 7th National Conference on Arti cial Intelligence - AAAI2000, pp. 68–73 (2000)

[46] Pearl, J.: Probabilistic Reasoning in Intelligent Systems - Networks of Plausible Inference. Morgan Kaufmann (1988)

[47] Pollock, J.L.: Philosophy and AI: Essays at the Interface, chap. OSCAR: A General Theory of Rationality, pp. 189–214. The MIT Press (1991)

[48] Rao, A.S., Georgeff, M.P.: Intentions and Rational Commitment. Tech. Rep. 8, Australian Artificial Intelligence Institute (1993)

[49] van Riemsdijk, M.B., Hindriks, K.V.: An empirical study of agent programs: A dynamic blocks world case study in goal. In: Proceedings of PRIMA'09 (2009)

[50] Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 2nd edn. Prentice Hall (2003)

[51] Schoppers, M.: Universal plans for reactive robots in unpredictable environments. In: Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI'87) (1987)

[52] Scowen, R.S.: Extended BNF - A generic base standard. `http://www.cl.cam.ac.uk/~mgk25/iso-14977-paper.pdf` (1996)

[53] Seth, A.: Agent-based modelling and the environmental complexity thesis. In: J. Hallam, D. Floreano, B. Hallam, G. Hayes, J.A. Meyer (eds.) From animals to animats 7: Proceedings of the Seventh International Conference on the Simulation of Adaptive Behavior, pp. 13–24. Cambridge, MA, MIT Press (2002)

[54] Shoham, Y.: Implementing the Intentional Stance. In: R. Cummins, J. Pollock (eds.) Philosophy and AI: Essays at the Interface, chap. 11, pp. 261–277. MIT Press (1991)

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

[56] Slaney, J., Thiébaux, S.: Blocks World revisited. Artificial Intelligence **125**, 119–153 (2001)

[57] Sterling, L., Shapiro, E.: The Art of Prolog, 2nd edn. MIT Press (1994)

[58] `http://www.swi-prolog.org/` (2008)

[59] `http://www.swi-prolog.org/pldoc/refman/` (2010)

[60] Wang, P., Goertzel, B., Franklin, S. (eds.): Artificial General Intelligence 2008: Proceedings of the First AGI Conference (2008)

[61] Watt, S.: The naive psychology manifesto. ?? (1995)

[62] Winograd, T.: Understanding Natural Language. Academic Press, New York (1972)

[63] Wooldridge, M., Jennings, N.R.: Intelligent agents: Theory and practice. Knowledge Engineering Review **10**, 115–152 (1995)