Programming Cognitive Agents in GOAL

Draft

March 26, 2014

Contents

Preface

The GOAL language is a high-level programming language for programming *multi-agent systems*. The language has been designed for the programming of *cognitive* and *autonomous decision-making agents*. GOAL is a *rule-based* language that supports *reasoning* about knowledge and the goals an agent pursues. Cognitive agents maintain a mental state and derive their decisions on what to do next from their *beliefs* and *goals*. This programming guide advocates a view of agent-oriented programming as *programming with mental states*.

The language offers a rich and powerful set of programming constructs and features for writing *agent programs*. It also provides support for connecting multi-agent systems to environments such as simulators, games, and robots (by means of the EIS interface standard) and for running multiple cognitive agents on a distributed computing environment. Agents are usually connected to some *environment* where they control entities such as grippers for moving blocks, cars in traffic simulators, or bots in real-time games. The possibilities are endless and the GOAL platform is distributed with a diverse set of environments for which agents can be programmed.

This Programming Guide has been significantly updated compared to the previous version from 2012. The language syntax has been updated throughout and is explained in detail. A "Quick Start" chapter has been added that introduces the language in the first chapter for those who want to immediately start programming in the language. Several features that have been added to the language have also been incorporated into this version. One of the more important conceptual changes in this version of the GOAL Programming Guide has been to move away from the concept of a rational agent to that of a *cognitive* agent. A programmer may be interested in programming optimal or near-optimal agents and, of course, can do so. The programming language, however, does not enforce a programmer to write agent programs that are optimal in this sense. Rather, it provides a tool for programming cognitive agents in the sense that these are agents that maintain a mental state.

GOAL has been extensively used at the Delft University of Technology in education and, besides Delft, has been used in education in many other places at both the Bachelor and Master level. Educational materials are available and can be requested from the author. Several assignments have been developed over time that ask students to program agents for simple environments such as: the classic *Blocks World* environment or a dynamic variant of it where users can interfere, the *Wumpus World* environment as described in [51], and more challenging environments such as an elevator simulator, the Blocks World for Teams [39], and the real-time UNREAL TOURNAMENT 3 gaming environment [32]. This last environment has been used in a large student project with more than 150 bachelor students. Students programmed a multi-agent system to control a team of four bots in a "Capture-the-Flag" scenario which were run against each other in a competition at the end of the project.

The language has evolved over time and we continue to develop more advanced tooling support for engineering and debugging multi-agent systems. As we are continuously developing and improving GOAL we suggest the reader to regularly check the GOAL website for updates:

http://ii.tudelft.nl/trac/goal.

In order to be able to further improve the GOAL platform, we very much appreciate your feedback. We hope to learn more from your experience with working with GOAL. So don't hesitate to contact us at goal@ii.tudelft.nl!

Koen V. Hindriks, Utrecht, January, 2014

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, either by helping to implement the language, by developing the theoretical foundations, or by 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, Vincent Koeman, Rien Korstanje, Nick Kraayenbrink, John-Jules Ch. Meyer, Peter Novak, M. Birna van Riemsdijk, Tijmen Roberti, Dhirendra Singh, Nick Tinnemeier, Wietske Visser, and 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.

Other Agent Programming Languages For those readers that are interested in further exploring the landscape of agent programming languages, we suggest to have a look at the two books [13, 14] on multi-agent programming.

Chapter 1

Getting Started with GOAL

We start this chapter with creating our first multi-agent system (MAS) in Goal that consists of a single agent that says "Hello, world!". Similar to "Hello world" programs written in other languages, the agent outputs "Hello, world!" in the console.

After having seen our very first example of a MAS, we continue with a walkthrough of the most important elements of the Goal language. To this end the basic "Hello World" agent is made slightly more interesting by making it print "Hello, world!" exactly 10 times.

Finally, we rewrite this agent and turn it into a simple script printing agent. At the end of the chapter, you should have a basic understanding of the type of agent programs that you can write in Goal.

1.1 A "Hello World" Agent

All agents are created as members of a multi-agent system. The first step in writing our "Hello World" agent therefore is creating a MAS. We assume you have started the Goal platform and that you can create a new MAS file. Do so now and name it `Hello World`; you may want to check out the Goal User Manual for instructions on how to do this [36]. You should see a template with empty **agentfiles** and **launchpolicy** sections. Complete it to make it look exactly like:

```
agentfiles{
   "helloWorldAgent.goal".
}
launchpolicy{
   launch helloWorldAgent : helloWorldAgent.
}
```

The MAS file references an agent file (with extension `.goal`). Save the MAS file and create the agent file; again, you may want to check out the Goal User Manual on how to do this. If not yet open, open the agent file. You should see a **main** module with an empty **program** section. The template for an agent file automatically inserts this module. Complete it to make it look exactly like:

```
main module[exit = always]{
   program{
      if true then print("Hello, world!") .
   }
}
```

Run the agent by launching the MAS using the Run button (if you don't know how, check out the User Manual). The MAS starts in paused mode. Start running it by pressing the Run button again. You should see the following in the Console area:

```
Hello, world!
GOAL agent helloWorldAgent terminated successfully.
```

You have successfully created your first agent program! If your agent did not terminate, you probably have forgotten to add the **exit** condition after the module's declaration.

1.2 Creating a Multi-Agent System

A MAS file is used to set up a new MAS project and to specify which agent files are needed for creating the multi-agent system. You can view a MAS file as a *recipe for launching a multi-agent system*. Let's create a new, second MAS project that you name "Hello World 10". As a result, you should now have a MAS file named `Hello World 10.mas2g`. The extension `mas2g` is used to indicate that the file is a MAS file.

Any MAS file must have at least two sections: An **agentfiles** section that tells the agent platform *which agent files* it should use to create the MAS and a **launchpolicy** section that tells the platform *when* to create and launch an agent. In our case, the **agentfiles** section will consist of one file reference because for now we only want to create a single agent. We will call our agent `helloWorldAgent10`; add the new agent file name to the MAS file and save the MAS:

```
agentfiles{
   % Simple Hello World agent that prints "Hello, world!" message 10 times.
   "helloWorldAgent10.goal" .
}
```

Agent files must have the extension `goal`. Note that even a single agent file needs to be included in a MAS file. The reason is that only a MAS file includes all the relevant information that is needed to create and launch the agent. Also note that comments can be added using the `%` sign. To complete our MAS file we need to add a *launch policy*. A launch policy specifies when to create an agent. Because we want to create our agent immediately when the MAS is created, in our case the **launchpolicy** section consists of a single `launch` instruction to launch our "Hello World" agent. Make sure the **launchpolicy** in the MAS file looks as follows.

```
launchpolicy{
   launch helloWorldAgent : helloWorldAgent10.
}
```

The name `helloWorldAgent` directly following the `launch` command is the name that will be given to the agent when it is created. The name `helloWorldAgent10` following the colon ':' is the name of the agent file that will be used to create the agent. Agent files referenced in the launch policy must be listed in the **agentfiles** section.

1.3 A Cognitive "Hello World" Agent

We will now extend our basic "Hello World" agent. The goal is to write a second agent that says "Hello, world!" exactly 10 times. To achieve this we will use a distinguishing feature of GOAL agents: cognitive agents can have goals and their main purpose in live is to achieve these goals. We will use this feature to simply *tell the agent it has a goal.* The only thing we need to think about is how to tell the agent that it has the goal of printing "Hello, world!" 10 times. There are many ways to do so. We will represent this goal as a simple counter that represents the number of times the agent has outputted something to the console. We use a simple *fact* and write `nrOfPrintedLines(10)` to represent the agent's goal. We tell the agent it has this goal by putting it in a **goals** section. A **goals** section, like a **program** section, needs to be part of a module. Because we are adding an *initial* goal for the agent, we put the **goals** section into the **init** module of the agent. The function of the **init** module is to initialize the agent before it starts making decisions and performing actions. The goal in the **goals** section will be put into the initial goal base of the agent. Open the agent file and add the following:

```
init module{
   goals{
      nrOfPrintedLines(10).
   }
}
```

We can also provide the agent with an initial *belief.* Instead of the **goals** section we use a **beliefs** section to add a belief to an agent's belief base. Beliefs should represent what is the case. Therefore, because initially the agent has not printed any lines, we put `nrOfPrintedLines(0)` in the **beliefs** section. Add the the **beliefs** section below *before* the **goals** section inside the **init** module to get the following. Do not forget the trailing dot '.' that follows the fact!

```
init module{
   beliefs{
      nrOfPrintedLines(0).
   }

   goals{
      nrOfPrintedLines(10).
   }
}
```

Together, the beliefs and goals of an agent are called its *mental state*. The states of an agent are very different from the states of other programs such as a Java program. Agent states consist of facts, and possibly also logical rules as we will see later, instead of assignments to variables. Our agent maintains two different databases of facts. One database called a *goal base* consists of things the agent wants to achieve. The other database is called a *belief base* and consists of facts that the agent believes are true (now). The fact that GOAL agents have a belief and goal base is the main reason for calling them *cognitive* agents. Our "Hello World" agent can derive decisions on what to do next from its initial goal to print a number of lines to the console and its belief that it did not print any lines yet.

Modules like the **init** module are the basic components that agents are made of. Another way of saying this is that an *agent is a set of modules.* The **init** module is important for initializing agents. But in order to make the agent do something we need other modules that allow the agent to make decisions and perform actions. The most important module that an agent uses to decide *what to do next* is called the **main** module. This time we want our agent to base its decision to print "Hello, world!" on its goal to print this line 10 times. To do so we need to modify the rule in the **main** module above that our previous agent used.

Rules for making decisions are more or less straightforward translations of rules we would think of when we would tell or explain someone what to do. Our "Hello World" agent, for example, should print "Hello, world!" if it wants to. That is, *if* the agent has a goal to print a number of lines, *then* it should print "Hello, world!". Rules of a GOAL agent have exactly the same **if** ... **then** ... form. Copy and modify the earlier rule of our first agent into the **main** module of our new agent and make sure it now looks like:

```
main module {
   program {
      if goal( nrOfPrintedLines(10) ) then print("Hello, world!") .
   }
}
```

The **if** is followed by a condition that states that the agent has a goal `nrOfPrintedLine(10)`, which is exactly our initial goal that we specified above. **goal** is an operator to check whether the agent has a particular goal. In the next section we will use another operator that can be used to inspect the beliefs of an agent. The second part of the rule from **then** on tells the agent to perform **print**`("Hello, world!")` if the condition holds.

Let's see what happens if we run this version of our agent. Launch the MAS, press the Run button, and pause it after a second or so. What you will see is not exactly what we wanted our agent to do. Instead of performing the **print** action exactly 10 times it keeps on printing. That the agent does this should not really come as a surprise. The agent did not do anything to keep track of how many times it performed the action! Inspect the agent's beliefs and open its Introspector by double clicking on the agent's name. You will see that its initial belief did not change and it still believes that it printed 0 lines.

Exercise 1.3.1

What will happen if we replace the belief `nrOfPrintedLines(0)` in the agent's **beliefs** section with `nrOfPrintedLines(10)`? Will the agent still perform any **print** actions? Check your answer by running the modified agent and inspecting its mental state using the Introspector. (After you did so undo your changes before you continue.)

1.4 Adding an Event Module

As we saw, the agent did not update its beliefs every time that it performed the **print** action. We can make the agent do so by making it respond to the *event* that it performed an action. As we will see later, an agent can also respond to other events such as the *event of receiving a percept* or *receiving a message*. An agent reacts to the *event that it performed an action* by calling a special module called the **event** module. Every time that an agent performs an action it will call this module. The **event** module's main function is to update the agent's mental state after performing an action. Usually, performing an action will change an agent's environment and the beliefs of the agent should be updated to reflect this. As a rule you should remember that *rules for processing events should be put in the **event** module.* Add the following **event** module to our agent program.

```
event module{
   program{
      if bel( nrOfPrintedLines(Count), NewCount is Count + 1 )
         then delete( nrOfPrintedLines(Count) ) + insert( nrOfPrintedLines(NewCount) ).
   }
}
```

As before, a **program** section is used to add rules to a module. The **bel** operator in the condition of the rule is used to inspect, or *query* in database terms, the current beliefs of the agent to find out what we need to remove from that base. The comma '`,`' is Prolog notation for "and" and is used to inspect the current belief `nrOfPrintedLines(Count)` of the agent *and* to increase the current `Count` with 1 to obtain `NewCount`. The **delete** action removes the old information and, thereafter, the **insert** action is performed to insert the new information into the belief base. The **+** operator is used for combining one or more actions; actions combined by **+** are performed in the order they appear.

Also note the use of `NewCount` and `Count` in the rule. These are Prolog variables. You can tell so because they start with a capital letter as usual in Prolog (any identifier starting with a capital is a variable in Prolog). Variables are used to retrieve concrete instances from facts from the agent's belief base (when inspecting that base). For example, given that the belief base of the agent contains the following fact:

```
nrOfPrintedLines(1246).
```

performing a query with the condition of the rule would associate the variable `Count` with 1246 and the variable `NewCount` with 1247. As a result all occurrences of these variables in the rule will be instantiated and we would get the instantiated rule:

```
  if bel( nrOfPrintedLines(1246), 1247 is 1246 + 1 )
    then delete( nrOfPrintedLines(1246) ) + insert( nrOfPrintedLines(1247) ) .
```

The fact that the rule condition could be instantiated to something that holds also tells us that the rule is applicable. Because the **delete** and **insert** actions can always be performed, applying the rule would update the belief base as desired.

We will make one more change to our "Hello World" agent before we run it again. By default a **main** module *never terminates* but in our case we want the agent to quit when its goal has been achieved. Therefore, we add an **exit** condition to the **main** module to quit the module when the agent has no more goals. Add the **exit** condition and make sure that the declaration of your **main** module looks like:

```
main module [exit = nogoals] {
```

We are ready to run our agent again. Do so now to see what happens.

Some magic has happened. You should have seen that the agent exactly prints "Hello, world!" ten times and then terminates. Open the Introspector and inspect the agent's goal base this time. You should see that the initial goal of the agent has disappeared! The reason is that the agent now believes it has achieved the goal (check this by inspecting the beliefs). As soon as an agent starts believing that one of its goals has been completely achieved that goal is removed from the agent's goal base. There is no point in keeping a goal that has been achieved. In our "Hello World" agent example this means that the agent repeatedly will apply the rule for printing text until it comes to believe it printed 10 lines of text. Because having the goal is a condition for applying the rule, the agent stops printing text when the goal has been completed.

Exercise 1.4.1

What would happen if we change the number of lines that the agent wants to print to the console in the **goals** section? For example, replace `nrOfPrintedLines(10)` with `nrOfPrintedLines(5)` and run the MAS again. As a result, you should see now that the agent does nothing at all anymore. The reason is that the rule in the **main** module is not applicable anymore. Fix this by using a *variable* in the condition of the rule instead of the hardcoded number 10. Run the MAS again to verify that the problem has been fixed.

1.5 Adding an Environment

So far we have been using the built-in **print** action to output text to the console. We will now replace this action and use an environment called *HelloWorldEnvironment* instead that opens a window and offers a service called `printText` to print text in this window. Our "Hello World" agent can make use of this service by *connecting the agent to the environment.*

An environment can be added to a multi-agent system by adding an **environment** section to the MAS file. The main thing we need to do is to indicate where the environment we want to use can be found. In our case we want to add an environment called *HelloWorldEnvironment*. GOAL supports loading environments automatically if they implement a well-defined environment interface called EIS [5, 6]. As an agent developer, it is sufficient to know that an environment implements this interface but we do not need to know more about it. All we need to do now is to create an **environment** section *at the start of the MAS file* and add a `jar` file that is distributed with GOAL called `HelloWorldEnvironment.jar` to this section. One way of making sure that the environment file can be found is to make sure that the `jar` file is located in the same folder as the MAS file that you created.

```
environment{
   env = "HelloWorldEnvironment.jar".
}
```

The next step is to make sure that our "Hello World" agent is connected to this environment. An environment makes available one or more *controllable entities* and agents can be connected to these entities. Once connected to an entity the agent controls the actions that the entity will perform. When the *HelloWorldEnvironment* is launched it makes available one entity. When this happens the agent platform is informed that an entity has been created. An agent then can be connected to that entity using a *launch rule* in the **launchpolicy** section. To connect our agent to the entity created by the *HelloWorldEnvironment* we need to only slightly change our earlier launch policy. Do this now and replace the launch instruction in the **launchpolicy** with the following launch rule:[1]

```
launchpolicy{
   when entity@env do launch helloWorldAgent : helloWorldAgent10.
}
```

The launch rule above adds a condition in front of the launch instruction. The rule is triggered whenever an entity becomes available. When triggered the agent `helloWorldAgent` is created and connected to the entity using the agent file `helloWorldAgent10`.

The next thing we need to do is *tell the agent which services the environment offers*. The *HelloWorldEnvironment* offers one service, or action, called `printText`. This action has one argument that can be filled with a string. For example, our agent can print "Hello, world!" by performing the action `printText("Hello, world!")`. We tell the agent it can perform this action by specifying it in an **actionspec** section. Open the agent file and add the **actionspec** section below *after* the **goals** section. The **init** module should now look like:

```
init module{
   beliefs{
      nrOfPrintedLines(0) .
   }
   goals{
      nrOfPrintedLines(10) .
   }
   actionspec{
      % The action printText expects a string of the form "..." as argument. It can
      % always be performed and has no other effect than printing Text to a window.
      printText(Text) {
         pre{ true }
         post{ true }
      }
   }
}
```

The **actionspec** section consists of action specifications. An action specification consists of a *declaration* of the action, e.g., `printText` with its argument `Text`, a *precondition* (**pre**) and a *postcondition* (**post**). A precondition is a condition that can be evaluated to true or false. Here we used **true** as precondition which always evaluates to true, meaning that the action can always be performed. A postcondition also evaluates to true or false but more importantly it can be used to let the agent know what the effect of performing the action is. For our "Hello World" agent we did not specify any effect and used **true** as postcondition. (We could have left the postcondition empty but by not doing so we indicate that we at least gave it some thought.)

[1] We will modify the `helloWorldAgent10` agent here but you might consider creating a new MAS called, e.g., `Hello World Environment.mas2g` and agent file called `helloWorldAgentEnvironment.goal` instead.

The only thing that remains is to start using the new service. To do so, replace the **print** action in the rule in the **main** module with `printText` to obtain (if you completed Exercise 1.4 instead of 10 below your rule may have a variable):

```
main module [exit = nogoals] {
   program {
      if goal( nrOfPrintedLines(10) ) then printText("Hello, world!") .
   }
}
```

Launch the MAS again and run it. Adding the environment and connecting the agent to the entity did not only make the agent print something to a window. The agent also received *percepts* from the environment. Open the Introspector of the "Hello World" agent and inspect the Percept tab. You should see the following:

```
percept(lastPrintedText('Hello, world!')).
percept(printedText(9)).
```

As you can see, the environment keeps track itself of how many times something has been printed, i.e., the `printText` action was performed, and informs the agent of this by means of the second percept `printedText` above.

If the environment's counting is correct (you may assume it is), then something is also going wrong. Instead of printing "Hello, world!" 10 times the environment informed the agent it printed it only 9 times... What happened? Recall that the event module is triggered by events and therefore also by percepts the agent receives. Because the agent received percepts before the agent performed an action, the **event** module incorrectly inserted `nrOfPrintedLines(1)` into the agent's belief base!

We can learn an important lesson from this: *whenever possible use percept information from the environment to update an agent's beliefs*. To follow up on this insight we will use percepts to update the agent's beliefs instead of the rule in the **event** module we used before. The basic idea is to add a rule to the agent program that tells us that *if* a percept is received that informs us we printed *NewCount* lines and we believe that we printed *Count* lines, *then* we should remove the old belief and add the new information to the belief base of the agent.

In order to create this rule for processing the percept, we need to find out how to inspect the percepts that an agent receives. As percepts are special kinds of beliefs, inspecting percepts can be done using the **bel** operator and the reserved keyword `percept`. Replace the old rule and add the following rule to the **event** module:

```
   if bel( percept( printedText(NewCount) ), nrOfPrintedLines(Count) )
      then delete( nrOfPrintedLines(Count) ) + insert( nrOfPrintedLines(NewCount) ).
```

Run the MAS and check the belief and percept base of the agent to verify that the agent printed "Hello, world!" 10 times. (Of course, you can also count the printed lines to be sure.)

Exercise 1.5.1

The *HelloWorldEnvironment* provides an initial percept `printedText(0)`. In this exercise we will use this percept instead of providing the agent with an initial belief `nrOfPrintedLines(0)` in the **beliefs** section in the **init** module. To this end do the following:

- Remove the **beliefs** section from the **init** module.
- Add a **program** section after the **goals** section in the **init** module.
- Add a rule for processing the initial percept. Copy the rule from the **event** module and modify it as needed. (As there is no initial belief anymore, remove the corresponding **bel** condition from the rule and also remove the **delete** action.)

1.6 A Simple Script Printing Agent

Using the language elements that we have seen in the previous sections, we will now extend the simple "Hello World" agent and turn it into an agent that prints a script that consists of a sequence of sentences. We want different script sentences to be printed on different lines so the only thing we need to do is make sure that the `printText` action each time prints the next sentence in our script. We also want the agent to store the script in one of its databases.

We introduce a new predicate `script(LineNr, Text)` to store the different sentences of our script. Introducing new predicates is up to us and we can freely choose predicate names with the exception that we should not use built-in predicates. The idea is that the first argument `LineNr` of the `script` predicate is used to specify the order and position of the sentence in the script. We will use the second argument `Text` to specify the string that needs to be printed in that position. We choose to use the belief base to store our script. Add the following **beliefs** section to the "Hello World" agent that we created so far.

```
beliefs{
   script(1, "Hello World") .
   script(2, "I am a rule-based, cognitive agent.") .
   script(3, "I have a simple purpose in life:") .
   script(4, "Print text that is part of my script.") .
   script(5, "For each sentence that is part of my script") .
   script(6, "I print text using a 'printText' action.") .
   script(7, "I keep track of the number of lines") .
   script(8, "that I have printed so far by means of") .
   script(9, "a percept that is provided by the printing") .
   script(10, "environment that I am using.") .
   script(11, "Bye now, see you next time!") .

   nrOfPrintedLines(0) .
}
```

The additional modification that we still need before we can use the script stored in the belief base is a change to the rule for printing text in the **main** module. We repeat the rule here for convenience.

```
   if goal( nrOfPrintedLines(10) ) then printText("Hello World") .
```

In order to figure out which line in the script we need to print, the agent should inspect its belief base. The idea is that the agent should retrieve the number of the last line printed, add 1 to that number to obtain the next line number, and retrieve the corresponding text from the belief base using the `script` predicate. Of course, we also need to make sure that the text we retrieved from the script is printed by the `printText` action. As before, we use the `,` operator to combine these different queries and use the **bel** operator to inspect the belief base. But this time we also use the `,` operator to combine the **goal** condition that is already present with the new **bel** condition for inspecting the belief base. A variable `Text` is used to retrieve the correct script line from the belief base and for instantiating the `printText` action with the corresponding text for this line. Now modify the rule in the **main** module's **program** section as follows.

```
   if goal( nrOfPrintedLines(10) ),
      bel( nrOfPrintedLines(LineNr), NextLine is LineNr + 1, script(NextLine, Text) )
      then printText(Text) .
```

Run the script agent to verify that the script is printed. You should see now that the script is printed except for the last line. This is because the agent has a goal to only print 10 lines but the script consists of 11 lines! Fix this issue by changing the `10` into `11`. You need to change this only in the **goals** section if you completed the first exercise in Section 1.4; otherwise you will also need to modify the `10` that occurs in the rule in the **main** module. The next exercise asks you to resolve the issue in a more principled manner.

Exercise 1.6.1

The goal in our script agent explicitly mentions the number of lines the agent wants to print. This is a form of hard-coding that we would like to avoid. We will do so by changing our script agent in this exercise and provide it with a goal to print the entire script.

- As a first step, simply remove the **goals** section in the **init** module. The idea is to compute a goal that will make the agent print the entire script and to adopt that goal.
- Add a **program** section after the **beliefs** section in the **init** module.
- Add the rule template **if bel**(. . .) **then adopt** `( nrOfPrintedLines(Max) ).` to the new **program** section. (Don't forget the trailing '.'!) The **adopt** action adds a goal to an agent's goal base.
- The idea now is to compute the `Max` number of lines that need to be printed in the condition of the rule to fill in the . . . . We assume you are familiar with Prolog here. Use the built-in Prolog predicates `findall/3` and `max_list/2` to compute the number of lines that need to be printed and make sure the result is returned in the variable `Max`. (See, e.g., the SWI Prolog manual [60] for additional explanation.)

You can check whether your agent has been modified correctly by comparing your agent with the script agent that is distributed with GOAL.

Chapter 2

Cognitive Agents

The main aim of this programming guide is to introduce you to the programming language GOAL. This language provides a *toolbox* that is useful for developing *cognitive agents*. The tools in this toolbox have been designed after the idea that our *folk psychology* can be put to good use for engineering such agents. The main tool in our toolbox is the *agent programming language* itself. This language, as we have seen in the previous chapter, uses the core concepts of *beliefs* and *goals* that are part of our folk psychology for programming agents. We humans use these concepts to explain our own behaviour and that of others. It is very appealing to also be able to use these very same concepts for writing programs. In the remainder of this guide we will explore how the agent programming language GOAL allows us to do exactly that.

The cognitive agents that we will be developing are also *autonomous decision-making agents.* This is a natural consequence of taking our folk psychology of beliefs and goals as a starting point for developing agents. Cognitive agents choose what to do next by making decisions based on their current beliefs and goals. GOAL agents use *rules* in combination with their beliefs and goals for making decisions. Rules provide agents with a basic *practical reasoning* capability. Practical reasoning is the capability of an agent to derive its choice of action from its beliefs and goals.

Apart from a basic capability for making decisions, it will also be useful to integrate various well-known Artificial Intelligence (AI) techniques into these agents. This will enlarge our toolbox for programming agents and will make it possible to develop more sophisticated agents for a range of problems that require more than basic reasoning and decision-making intelligence. A second aim of this guide therefore is to explain how we can engineer agents that integrate techniques from Artificial Intelligence such as *knowledge representation*, machine learning, and, for example, *planning* to create more intelligent agents.

Before we explore the toolbox for engineering cognitive agents in more detail in the remainder of this programming guide, in this chapter we first provide some background and explain and discuss some of the core concepts that our toolbox is based on. The reader that is more interested in starting to write cognitive agents itself right away can first skip this chapter and revisit it at some later time to gain a better understanding of the origins of the ideas behind our toolbox for programming agents.

2.1 What is an Agent?

If we want to use the notion of an agent to develop software, it is useful 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* [51]. 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. It emphasizes that 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 from its environment which inform it about the *current* *state* of this environment. It also highlights that it is important to identify which *actions* an agent may perform to *change* its environment and *how* these actions affect the environment. An environment does not need to be a *physical* environment. Although an agent may be a physical *robot* acting in a physical environment it may just as well be an entity that is part of a virtual environment such as a bot that searches the World Wide Web. This concept of agent suggests that an agent should have some basic abilities for processing percepts and for selecting actions. In other words, we need a solution to address the fundamental problem of Artificial Intelligence: *How does an agent decide what to do next*?

The definition of an agent from [51] is very general and does not tell us much about engineering cognitive agents. The concept highlights the importance of percepts and actions for developing agents but agents themselves remain a black box. Various other characteristics have been identified to differentiate software agents from other types of software programs. Agents, according to a well-known definition of [64], are:

- *situated* in an environment,
- *reactive* because they are able to *perceive their environment* and *respond in a timely fashion* to perceived changes in their environment,
- *proactive* because they are *goal-directed* and *take the initiative* by performing actions that achieve their goals, and,
- *social* because they *communicate and cooperate* with other agents which are also part of the same multi-agent system.

These characteristics of agents taken together define what has been called a *weak* notion of agency. Any software system that is situated, reactive, proactive and social in this sense counts as an agent. Compared to the agent concept of [51] weak agency adds that agents should be *goal-directed* and *social*. Of course, to be useful for engineering agents, we need to make these aspects of agents more concrete. It is one of our aims to do this and clarify and demonstrate what it means to respond in a timely fashion, to take the initiative, cooperate and to explain how to implement these characteristics in a cognitive agent. Our toolbox will provide various tools to do so. For example, the Environment Interface Standard provides a tool for implementing the interaction of an agent with its environment [4, 5]. We have already seen that cognitive agents derive their decisions a.o. from their goals and may initiate action by pursuing goals. GOAL also provides support for developing multi-agent systems. In Chapter 7 we will see how agents can interact with each other by communicating and exchanging messages. That chapter will also introduce *mental models* which allow agents to maintain a shared situational awareness that facilitates communication and cooperation between agents.

One notion that has consistently been used by many to differentiate agents from other software entities is the notion of *autonomy*. According to [64] agents are autonomous if they *operate without the intervention of others*, and *have control over their actions and internal state*. This definition of autonomy seems inspired by a key difference between the agent-oriented and object-oriented programming paradigm: whereas objects do not have control over their own state and the methods made available by an object may be called by any other object that has access to it, this is different for agents. Autonomous agents are decision-makers that have *complete control* over their own choice of action. Agents therefore also fit particularly well into an important trend where more and more tasks and responsibilities are delegated to machines. The notion of autonomy itself, however, does not tell us too much about how to engineer agents either. In particular, we should not use the concept of an autonomous agent to enforce all kinds of restrictions on the design of a single or a multi-agent system. One of the main implications for design that follows from the concept of autonomy is that in the design of agents it is particularly important to identify who and when has the *responsibility to see to it that a goal is achieved*.

Interestingly, definitions of agency have also been discussed in the context of dynamical systems theory, an area that is typically more oriented towards physical and biological systems. There are clear relations between this type of work and, for example, robotics. 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. [2] assumes that an agent always is coupled with an environment. The view of an agent as a distinct individual sets an agent apart from its environment and other agents. A key aspect of agency identified in this work is the *asymmetry between an agent and its environment*: An agent intentionally initiates changes in its environment whereas an environment only responds to what an agent does.

2.2 The Intentional Stance

Our notion of a cognitive agent is based on the idea that the core folk psychological notions can be used for engineering agents. The usefulness of common sense folk psychological concepts can be traced back to the origins of Artificial Intelligence. Interestingly, the title of one of the first papers on AI written by McCarthy is *Programs with Common Sense* [41]. In AI, several of the core folk psychological concepts are used to formulate key research problems. For example, the area of planning 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. The intuitions that guide this research thus are clearly derived from the everyday, common sense meanings of these concepts. The techniques and languages developed within AI, however, have produced more precise, *formal* counterparts of these concepts that perhaps are not as rich in meaning but have otherwise proven very useful for engineering systems (cf. [55]).

For this reason it appears quite natural to use the core folk psychological concepts of beliefs, knowledge, and goals not only for *describing* but also for *engineering* agents. It has been argued that engineering agents in terms of these so-called intentional notions is advantageous for several reasons. As [42] 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)

Shoham, who was one of the first to propose a new programming paradigm that he called *agent-oriented programming*, also cites McCarthy about the usefulness of ascribing intentional notions to machines [42, 56]. It has been realized since long that in order to have machines compute with such notions it is imperative to more precisely specify their meaning [56]. To this end, various logical accounts have been proposed to clarify the core meaning of a range of common sense notions [19, 38, 47]. These accounts have aimed to capture the essence of several common sense notions that are useful for specifying cognitive agent programs. These logical accounts did not provide a computational approach for developing cognitive agents, however. The GOAL agent programming language introduced here provides both: It is based on a precise and well-defined operational semantics that unambiguously specifies the beliefs, goals, actions and decision-making of a cognitive agent and it also provides a computational framework for programming agents [12, 33, 35].

A key source of inspiration for viewing objects as intentional systems has been the work of philosopher Dennett [23] on the *intentional stance*. The idea to view agents as *intentional systems* has inspired many others; some of the more prominent examples in the area of multi-agent systems are [48, 55, 56, 64]. What is the *intentional stance*? According to [23], it is a strategy that *treats the object* whose behaviour you want to predict *as a cognitive agent with beliefs and desires*:

> first you decide to treat the object whose behaviour 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.

By now it should be clear that taking an intentional stance towards objects provides us with a powerful tool for understanding and predicting the behaviour of that object. It provides just as powerful a tool for engineering systems. Because our main aim is to engineer MAS, we are less interested in the *normative* stance that Dennett requires to be able to view the agent as *rational*. It is sufficient for our purpose that cognitive agent programs are designed as believers that take the initiative to achieve their goals, even if they do not behave optimally. The idea that an agent acts so as to realize its goals (rather than doing the opposite as an irrational agent perhaps would), however, also is a useful intuition for programming cognitive agents. The important point is that, as Dennett puts it, by ascribing beliefs and desires (or goals, as we will do), we only need *a little practical reasoning to figure out what the agent should do.* That is exactly what we need to figure out when engineering cognitive agents!

We thus promote the intentional stance here as an *stance!engineering* or *design stance*. The basic idea is to explicitly label certain parts of our agent programs as *knowledge*, *beliefs*, and *goals*. The main task of a programmer then remains to provide a cognitive agent with the practical reasoning that is needed to figure out what to do. To this end, the agent programming language offers rules as the main language element for programming the decision-making of an agent and offers the programming construct of a module for providing additional structure on the agent's decision-making process.

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 [41] and [56]). Core folk psychological concepts are integrated into the agent programming language as useful programming constructs for engineering cognitive and autonomous decision-making agents. This addresses the challenge of providing a tool that supports using these concepts for engineering cognitive agents. An additional challenge that we need to address as a result of integrating these concepts into a programming language is related to the programmer. We need to ensure that the programming language is a tool 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. Special attention has been paid to these aspects and over the years the agent programming language Goal has evolved into a tool that facilitates the programmer in engineering cognitive agents [37, 50].

2.3 What is Folk Psychology?

If we want to use the core of folk psychology for designing and engineering agents, we should first make explicit what folk psychology is. 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 for developing agents and which elements are less useful. Even though our folk psychology must be shared it turns out that addressing this question is harder than expected. To a certain extend an analogy perhaps can 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 *explain* (or even 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 the core folk psychological concepts such as beliefs, goals, etc. Interestingly, our understanding of these concepts evolves, and, 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.

We take a pragmatic stance here as our primary aim is to apply folk psychology for engineering multi-agent systems. The core of folk psychology for us consists of core concepts such as beliefs, knowledge, desires, goals, intentions, and plans. The most basic concepts in this list are the beliefs and goals of an agent that we use to explain what must have gone on in the mind of that agent when it decided to do something. We are well aware that agents might be attributed a far more diverse set of mental attitudes, including subtle variations of our main list of attitudes such as expecting, fearing, etc. To keep things simple, however, we will only use the most basic concepts of belief and goal. It turns out, moreover, that these basic concepts already are sufficient for engineering quite sophisticated cognitive agents. It is also more difficult to explain the meaning of the more complex concepts that are part of folk psychology, which makes these concepts more difficult to use for engineering agents (recall the ease of use aspect discussed above).

2.3.1 Rational Agents?

According to [23] the use of intentional notions such as beliefs and goals presupposes *rationality* in the agent that is so described. In a very general sense it is true that folk psychology assumes agents to be *rational*. Basically, this means that agents perform actions to further their goals, while taking into account their beliefs about the current state of the world. Even though we think the idea of an agent that acts towards achieving its goal is important and can be usefully exploited for engineering agents, we do not want to emphasize this aspect of agents too much. Agents that have beliefs and goals can also be irrational and it is perfectly possible to program such agents in the agent programming language Goal. In fact, it is generally much harder to develop perfectly rational agents! That being said it is still useful to explore the notion of rationality in somewhat more detail and discuss what it means to have beliefs and goals that are rational.

The beliefs of an agent, if they are to count as rational, should satisfy a number of conditions. First, the beliefs that a *rational* agent has should most of the time 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.[1] As has been argued by several philosophers, most of our common sense beliefs must be true (see e.g. [23, 25]), which is not to say that agents may not maintain exotic beliefs that have little justification. For example, an agent might have a belief that pyramids cannot have been built without the help of extraterrestrial life. The point is that most of the beliefs that we have must be true because otherwise our explanations of each other's behaviour would make little or no sense at all!

There is also 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 on context, but an agent that acts without taking relevant aspects of its environment into account may be said to be irrational (assuming the agent has some kind of perceptual access to it). For example, it would be strange intuitively 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 although different ones than those that apply to beliefs. Desires nor goals need to be true or complete in the

[1] 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. [24, 43]). 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 Plato's definition as well.

sense that beliefs should be. Agents are to a large extent free 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, it is generally agreed that the goals that an agent adopts must conform with certain *feasibility conditions*. These conditions include that goals should be *consistent* in the sense that one goal of an agent should 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 if 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).

2.3.2 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 [23]. 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:

$$\mathbf{bel}(p) \quad : \quad \text{the agent } \textit{believes} \text{ that } p \tag{2.1}$$

$$\mathbf{goal}(p) \quad : \quad \text{the agent } \textit{has a goal} \text{ (or } \textit{wants}\text{) that } p \tag{2.2}$$

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:

$$\mathbf{bel}(a, \mathbf{bel}(b, p)) \quad : \quad a \textit{ believes} \text{ that } b \textit{ believes} \text{ that } p \tag{2.3}$$

$$\mathbf{bel}(a, \mathbf{goal}(b, p)) \quad : \quad a \textit{ believes} \text{ that } b \textit{ wants} \text{ that } p \tag{2.4}$$

$$\mathbf{goal}(a, \mathbf{bel}(b, p)) \quad : \quad a \textit{ wants} \text{ that } b \textit{ believes} \text{ that } p \tag{2.5}$$

$$\mathbf{goal}(a, \mathbf{goal}(b, p)) \quad : \quad a \textit{ wants} \text{ that } b \textit{ wants} \text{ that } p \tag{2.6}$$

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 [42]. 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 really 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]. GOAL supports the engineering of first- and second-order intentional systems. A GOAL agent, for example, can have beliefs 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* [23], 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* [25]. 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.[2] 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 [62]. 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.[3]

2.4 Notes

The idea of using the intentional stance for developing cognitive agents has been proposed in various papers by Shoham. Shoham's view is that agents are "formal versions of human agents", see for example, [55].[4]

Central in the early work on agent-oriented programming (AOP) in [55] has been *speech act theory*. According to [55], "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 7 for a more detailed discussion.

Our view on designing and programming agents as one particular form of *implementing the intentional stance* is related to [55]. The *intentional stance* has been first discussed by Daniel

[2]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.

[3][42] 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".

[4]The title of this book chapter, *Implementing the Intentional Stance*, refers to a project that aims at a reorientation of the intentional stance to a design stance for developing rational agents. Shoham's view, however, is different from what we are after. He argues that artificial agents should have "formal versions of knowledge and belief, abilities and choices, and possibly a few other mentalistic-sounding qualities". Our view is that the intentional stance is useful for engineering agents but that does not mean, as Shoham seems to suggest, that we should aim for agents that are themselves able to adopt the intentional stance. As discussed in the main text, that would require third- or higher-order intentionality that is not supported in GOAL [25].

Dennett in [23]. The usefulness of ascribing mental attitudes to machines, however, was already discussed by McCarthy [42].

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. [64] defines autonomy as the ability of an agent to control its actions and internal state. In [16] 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, [7] 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. If adaptivity is a requirement for autonomy, definitions and discussions of autonomy should at least reference the ability to sense an environment.

2.5 Exercises

2.5.1

An agent in the weak sense of agency is an entity that is situated, reactive, proactive and social. Choose two of these characteristics and discuss whether they provide support for an agent designer to engineer an agent system. Explain and motivate why you think they are or are not helpful for writing an agent program.

2.5.2

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.

2.5.3

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 what makes them autonomous or not according to you.

2.5.4

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.

2.5.5

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 $\mathbf{bel}(\neg\mathbf{bel}(p))$.

2.5.6

It has been argued that higher-order beliefs about opponents in games are crucial for good game play, i.e. winning (see e.g. [25]).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

2.5.7

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

2.5.8

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 are not always rational. What do you think? Do people act rational most of the time or not. If you think so, explain why, or otherwise explain why you think people are often irrational.

Chapter 3

Mental States

A cognitive agent maintains a *mental state* to represent the current state of its environment and the state it wants the environment to be in. Mental states are used by agents to assess the current situation and determine what to do next. Agent programming is *programming with mental states*. It is very important for an agent to make sure that its mental state is up to date. But the first thing to get right is to make sure that a mental state contains the information that the agent needs to make the right decision. More precisely, we need to make sure that the agent is able to represent the information it needs about its environment. In this chapter we will discuss the representation of environment information and the reasoning an agent can perform using its mental state. The representation of the current state determines the *informational state* of the agent, and consists of the *knowledge* and *beliefs* of the 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 of an agent is made up of its knowledge, beliefs and goals. We look in more detail at how an agent's mental state can be represented. We also discuss how an agent can reason with its knowledge, beliefs, and goals by inspecting its own mental state. The main concept that we introduce in this chapter is that of a *mental state condition*. These conditions allow an agent to reason with and inspect its mental state and can be used by the agent to make decisions.

3.1 Representing Knowledge, Beliefs and Goals

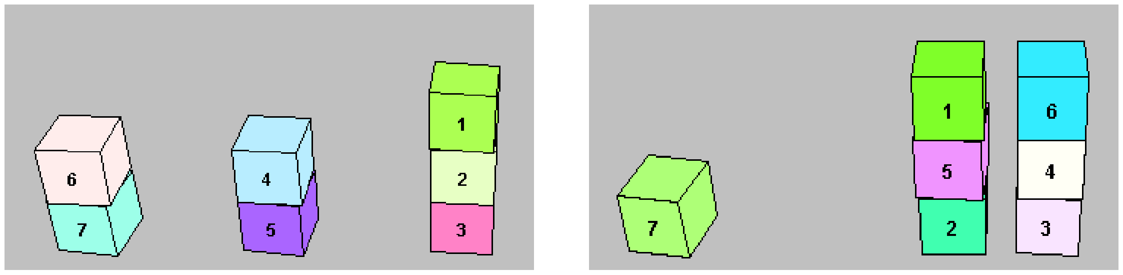
One of the first steps in developing and writing a cognitive agent is to design and write the knowledge, beliefs and goals that an agent needs to make the right decisions. This means that we need to think about which knowledge, which beliefs, and which goals the agent should have to behave correctly (do what we as programmers want the agent to do). We call this task *designing the agent's mental state*. Specifying the content of an agent's mental state is not something that is finished in one go but typically is done in several iterations while writing an agent program. It is particularly important to get the representation of the agent's knowledge, beliefs and goals right because the choice of action of the agent depends on it. The action specifications and action rules discussed in Chapters 4 and 5 also depend on it.

For representing an agent's mental state we use a *knowledge representation language*. We will use one and the same language for the content of each of the databases for storing knowledge, beliefs, and goals. Using one language will, for example, allow an agent to easily check whether it believes that a goal it has has been achieved. In principle, we can choose any knowledge representation language for representing an agent's mental state. The programming language Goal is not married to any particular knowledge representation language and we are free to choose one. Here we have chosen to use Prolog [58, 59, 60]. Prolog is a very well-known and

[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.

powerful logic language. We will use Prolog for representing mental states and show how we can do so by writing an example GOAL agent for the classic Blocks World environment in this chapter.



(a) Initial State (b) Goal State

Figure 3.1: Example Blocks World problem taken from [57].

3.1.1 Example: The Blocks World

As a running example in this and the following chapters we will use one of the most well-known problem domains in Artificial Intelligence known as the *Blocks World* [51, 57, 63]. In its most simple and well-known form, in the Blocks World an agent can move and stack cube-shaped blocks on a table by controlling a robot gripper. An important fact about the simple Blocks World 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 and on how to represent and reason with configurations of blocks; we will discuss how an agent can control the gripper in Chapter 4 and which blocks it should move in Chapter 5.

When representing information about an agent's environment we need to be precise. We need to think about what exactly needs to be represented and which possibilities that are available in the environment need to be modelled. It is a good idea when starting to write an agent program to first investigate the problem domain or environment that the agent needs to deal with in more detail (cf. also Chapter 8). For the Blocks World, the following restrictions that apply are relevant. First, the Blocks World contains one or more but at most a finite number of *blocks*, a *table* where blocks can sit on, and a *gripper* for moving a block; there are no other things present in the Blocks World. To make the Blocks World interesting we need more than two blocks but in principle we could do with less if we would like to. A block can be directly on top of at most one other block and at most one block sits directly on top of another block. That is, a block is part of a stack and either is located on top of a single other block, or it is sitting directly on the table. These are some of the basic "laws" of the Blocks World (cf. [20]).[3] See Figure 3.1 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 large enough to be able to place all blocks directly on the table (i.e., without any block sitting on top of another). This is another basic "law" of our version of the Blocks World.[4]

An agent that is tasked with achieving a goal state in the Blocks World needs to be able to reason about block configurations to make the right decision. It needs to be able to compare the current and the goal configuration. To do so, it needs to *represent* arbitrary configurations of blocks. It needs to be able to represent such configurations mainly for two reasons: to keep

[3]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 modelled as an instantaneous move.

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

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 Prolog, we need to introduce specific predicates to be able to specify a configuration of blocks. Typically, it is a good idea to introduce predicates that correspond with the most *basic concepts* that are relevant in the problem domain. More complicated concepts then may be *defined* in terms of the more basic concepts. We will use this strategy here as well.[5]

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 the simple binary predicate `on`:

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

The predicate `on(X, Y)` means that *block* `X` *is (directly) on top of* `Y`. For example, `on(b1,b2)` is used to represent the fact that block `b1` is on block `b2` and `on(b3,table)` means that block `b3` is on the table. This is our informal definition of the predicate `on(X, Y)`. In order to use this predicate correctly we need to agree on some basic rules for using the predicate. First, only *blocks* can be on top of something else in the Blocks World. That is, the first argument `X` should only be filled or instantiated with the name of a block. Second, `X` must be *directly* on top of `Y` meaning that there can be no other block in between block `X` and `Y`. Third, the second argument `Y` may refer *either* to a block *or* to the table. And, fourth and finally, there can be *at most one* block on top of another block. If this rule is violated, we would get an *inconsistent* state. We would also get inconsistent states if a block would be represented as being both on the table as well as on another block. Note that it is up to us programmers who write the agent to stick to these rules and to use the predicate `on` correctly. If we would not follow these rules, the `on` predicate would mean something different than we intend it to mean!

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 facts that represent the goal state correspond with the goal of an agent that has the task to solve the Blocks World problem.

There is one other thing that we should keep in the back of our minds. We have implicitly assumed so far that all blocks have *unique names* `b1`, `b2`, `b3`, `...` that we can use to refer to each individual block. This *Unique Names assumption* is very useful because if we make this assumption we can uniquely identify each block and we can distinguish one block from another. Using different names for each block greatly simplifies our task as programmer when writing an agent for the Blocks World. It means that we never have to worry about the possibility that block `b1` is the same block as, for example, block `b12`; because the blocks have different names they also must be different blocks.

[5]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 a predicate `above` [20]. Here we follow tradition and introduce the predicates `on` and `clear` to represent blocks configurations. See the Exercises 3.4 for more on using `above`.

We will also assume that we have names for *all* blocks in the Blocks World; that is, there are no blocks in the Blocks World that do not have a name and that we cannot talk about. To be complete, we also explicitly add the identifier `table` to the list of names here. We already used this label above to refer to the table in the Blocks World. Because we have names for everything now we have in effect implemented a *Domain Closure assumption* for the Blocks World. This assumption simply means that everything there is in our problem domain has a unique name. In other words, this means that we assume that *there are no other (unnamed) blocks*.[6]

Using the `on` predicate we can represent the configuration of blocks in a Blocks World state. For example, the configuration of blocks in the initial state of Figure 3.1 can be represented by:

```
on(b1,b2). on(b2,b3). on(b3,table). on(b4,b5). on(b5,table). on(b6,b7). on(b7,table).
```

We can now also *define* the concept of a block. Even if strictly speaking we do not need an explicit definition of a block (it is after all implicitly defined by our `on` predicate), it is still useful to introduce a unary `block` predicate for various reasons. One reason is readability: a program that uses the `block` predicate is more easy to read and understand than one that does not. Defining the `block` predicate also allows us to illustrate how we can benefit from the rules we agreed upon for using the `on` predicate. Recall that we agreed above to fill the first argument of the `on` predicate with names of blocks only. But this convention means that something that sits on top of something else must be a block. In other words, we can derive that something is a block from an `on` fact. We thus can define the `block` predicate by a Prolog rule as follows:

```
block(X) :- on(X, _).
```

Because we agreed to never instantiate the first argument with `table` we know that the query `block(table)` will fail. The second argument has been filled with Prolog's *don't care variable* `_` as we are not interested in the value of this argument when evaluating the `block` predicate.

It is useful to introduce one more predicate for representing some additional facts about the Blocks World. First, we want to represent the fact that the table is always considered to be clear. Second, we want to be able to conclude from facts about which blocks sit on top of each other whether a block is clear or not. In order to deal with the limitations of the gripper in our Blocks World it is important to be able to conclude that a block is clear. For this purpose, we introduce the predicate `clear(X)` to mean that object `X` is clear.

For the case that `X` is a block, `clear(X)` means there is no other block on top of it. If an agent has access to all information about all blocks, the agent can infer from this information that a block is clear. It then follows that a block is clear if there is no fact that another block sits on top of it. We will assume that the agent has access to the complete state or configuration of blocks throughout this chapter. Another way of saying this is that we assume that the Blocks World environment is *fully observable* (cf. [51]). We then can use the following rule to define the `clear(X)` predicate with `X` a block in terms of the `block` and `on` predicates:

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

Prolog's *negation as failure* is used here to conclude that there is no fact that another block sits on top of `X`. Because we don't care this time which block might sit on top of `X` we have used the don't care variable in the first argument of the `on` predicate. It is easy to see from Figure 3.1 that blocks `b1`, `b4`, and `b6` are clear in the initial state. Use the representation of the initial state above, and the definitions of the `block` and `clear` predicates to show that this also follows from our definitions.

[6]Note that we cannot explicitly state this using Prolog (we would need a quantifier to do so) but this assumption is implicitly built into Prolog (because Prolog implements a version of the *Closed World assumption*, see also the box discussing this assumption below).

Floundering
As a side note, we want to remark here that it is important to take the order of Prolog conjuncts into account when defining a predicate. The `clear` predicate provides a simple example of the so-called *floundering* problem in Prolog. We would run into this problem if we accidentally would reverse the conjuncts in the body of the rule for `clear` and would have used the following rule instead:

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

The floundering problem has to do with the instantiation of variables. Although the rule above would still work fine when `X` is instantiated, it would give the wrong answer when the variable `X` is *not* instantiated. The query `clear(a)` in the initial state of Figure 3.1, for example, correctly yields true. But the query `clear(X)` would yield false and return no answers! The problem is that the variable `X` needs to be instantiated first *before* the negation `not( on(_, X) )` is evaluated because even with only a single block present this conjunct would fail otherwise.
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. In our definition of `clear` in the main text we have correctly put the positive literal `block(X)` first to ensure that the variable `X` is bound before the variable is used within the second, negative literal `not(on(_, X))`.

The `clear` predicate has been introduced to allow an agent to conclude that a block `X` can be placed on top of another block `Y` or on the table. The gripper controlled by the agent can only move a block if there is no other block on top of it or there is room on the table to put the block on. Because we assumed that the table always has room to place a block, the agent should also be able to infer this fact. The rule for `clear(X)` above, however, does not allow an agent to conclude this (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)`). To also cover the case that the table is clear we add the following fact about the table to complete our definition of the `clear` predicate:

```
clear(table)
```

We have introduced a new language for representing facts and for reasoning about the Blocks World. The language is very basic and consists only of a few concepts but it is already useful for our purpose: solving Blocks World problems. That is not to say that there are no useful extensions that we can introduce for representing and reasoning about the Blocks World. In fact, below we provide a definition of the concept of a *tower* that will be particularly useful for solving Blocks World problems. And there are still other useful concepts that may be introduced (see the Section 3.4 where you are asked to use the binary predicate `above`).

The concepts that are part of a language are also called an *ontology*. It is usually a very good idea to explicitly document the ontology used by an agent (see also Chapter 8). Doing so facilitates working in a team of programmers that work on the same multi-agent system.

3.1.2 Mental States

Now we have a language for talking about Blocks World problems, we are ready to specify the mental state of an agent that controls the robot gripper in the Blocks World. A mental state of a Goal agent consists of *three* different types of databases: a *knowledge base*, a *belief base*, and a *goal base*. The knowledge and beliefs of an agent keep track of and are used to reason about the current state of the agent's environment and goals represent the state that the agent wants to realize in the environment.

The information that an agent has about its environment consists of the *knowledge* and of the *beliefs* that 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 *can change* at runtime. There is another very important difference that we will discuss below.

The rules that we have introduced above to define `block` and `clear` are particularly good examples of knowledge. As we will learn in Chapter 4, Prolog rules cannot be modified. In general, therefore, it is a good idea to put rules and definitions into the knowledge base. Such rules typically consist of either *conceptual definitions* or *domain knowledge* that represents the basic "laws" or the "logic" of the problem domain. The rules for `block` and `clear` are examples of conceptual definitions. The knowledge base may also contain facts that do not change. A good example of a fact that is always true and therefore can be added to the knowledge base is `clear(table)`.

Knowledge of an agent is specified in a **knowledge** section of an agent program. The **knowledge** keyword is used to indicate that what follows is the knowledge of the agent. A **knowledge** section is used to create the *knowledge base* of the agent. Usually, a **knowledge** section is included in the **init** module of an agent if it consists of general domain knowledge but each module can have its own **knowledge** section.

```
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 code listed in the **knowledge** section is a Prolog program. In fact, if Prolog is used as the knowledge representation language used by the agent, the content of a **knowledge** section must be a valid Prolog program and respect the usual syntax of Prolog. It is also possible to use many of the built-in operators of the Prolog system that is used (in our case, SWI Prolog [59]).

Because facts in the knowledge base cannot change this base cannot be used to keep track of facts about the environment that change over time. Facts that can change must be put in the *belief base* of the agent. The initial beliefs that an agent should have can be specified in a **beliefs** section. We can, for example, represent the initial state of Figure 3.1 in a **beliefs** section of an agent program. This section may then be used to create the initial belief base of the agent. Like a **knowledge** section, a **beliefs** section is usually included in the **init** module of an agent. In principle, though, any module can have a **beliefs** section.

```
beliefs{
  on(b1,b2). on(b2,b3). on(b3,table). on(b4,b5). on(b5,table). on(b6,b7). on(b7,table).
}
```

The facts included in the **beliefs** section may change when, for example, an agent performs an action. A similar remark as for the **knowledge** section applies to the **beliefs** section: the code within this section must be a valid Prolog program. Our example **beliefs** section above consists of facts only. Although this is not required, as a general rule it is best practice to use the **beliefs** section for facts (that can change) and the **knowledge** section for rules (that cannot change).

The knowledge base introduced above by itself is not sufficient to derive the information about the Blocks World that is needed for making the right decisions. This is true more generally and applies to most domains and environments. Even though we can conclude by only inspecting the knowledge base that the table is clear, we cannot, for example, derive that block `b1` is also clear. Usually, only the combination of the agent's knowledge and beliefs allows an agent to make the inferences it needs to be able to make about the current state of its environment. It thus is much more useful to combine knowledge and beliefs. This is also what is done when a GOAL agent evaluates what it believes as we will see below in more detail. That is, a GOAL agent draws

Perception
One downside of specifying beliefs in a **beliefs** section as we did in the example for the Blocks World is that we are *hard-coding* information that is *specific* for one particular environment configuration into the agent program. It is often better to not include a **beliefs** section for this reason. When you are writing an agent program, the initial state that the agent needs to handle is usually unknown. Typically, the initial beliefs of an agent need to be derived from percepts received from its environment. In other words, an agent needs to collect information about the state of its environment by looking at it (by means of sensors that are available to the entity that the agent controls).

conclusions about what it believes by reasoning with both its knowledge and its beliefs. Note that the combination of knowledge and beliefs in the example knowledge and belief bases above allows us to conclude that block `b1` is clear.

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 as failure* of Prolog. As a result of using this operator, the rule for `clear(X)` will succeed if information is *missing* that there is a block sitting on top of `X`. In that case, `block(X), not(on(_, X))` succeeds simply because there is *no* information whether `on(_, block)` holds.
Another way to make the same point is 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 belief base of the agent needs to represent the state of the environment completely. More often than not an agent can only keep track of part of its environment. An agent can only keep track of the complete state of its environment if that state is *fully observable* for the agent [51].
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.

A final remark on the **beliefs** and **knowledge** sections is in order. It is not possible to introduce one and the same predicate in the **knowledge** as well as in the **beliefs** section of an agent program. That is, one and the same predicate cannot occur in the head of a Prolog rule that appears in the **knowledge** section as well as in the head of a 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 close together.

Finally, we will represent the goal state of our Blocks World agent. The goal state in Figure 3.1 specifies a single goal that the agent should achieve. Initial goals that an agent should have can be specified in a **goals** section. A **goals** section that is part of the **init** module is used to create the initial *goal base* of an agent. In principle, though, any module can have a **goals** section. For our particular example illustrated in Fig. 3.1 we can represent the agent's initial goal as follows:

```
goals{
  on(b1,b5), on(b2,table), on(b3,table), on(b4,b3), on(b5,b2), on(b6,b4), on(b7,table).
}
```

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 3.1 is represented as a single *conjunction* in the **goals** section. That is, the facts that describe the goal state are separated by means of the comma `,` operator and not by the dot `.` operator. A dot is only used to indicate the end of a goal and within a **goals** section separates different goals from each other. In other words, the **goals** section below that separates each of the facts by a dot consists of seven goals instead of one.

```
goals{
  on(b1,b5). on(b2,table). on(c3,table). on(b4,b3). on(b5,b2). on(b6,b4). on(b7,table).
}
```

The fact that the conjunctive goal counts as a *single* goal and the facts separated by the dot count as *seven* distinct goals makes a big difference for how the agent can achieve its goals. The seven distinct goals are completely independent from each other and can be achieved in any order that the agent would like. The sixth goal `on(b6,b4)` could be achieved immediately in the initial state by moving block `b6` on top of `b4`. Of course, if that would be the first move of the agent, in order to achieve the goal `on(b4,b3)` the agent would have to remove `b6` again from `b4` at some later point in time. The point is, however, that moving `b6` onto `b4` would *not* get us any closer to achieving the conjunctive goal. The conjunctive goal requires that the agent creates a tower of blocks where `b6` sits on top of `b4` *and* `b4` sits op top of `b3` *and* `b3` sits on the table. That is, a conjunctive goal requires that all of its *sub-goals* are achieved *simultaneously*. A conjunctive goal is only said to be *completely achieved* if all of its sub-goals have been achieved in one and the same state (at one and the same time). Thus if we would move `b6` onto `b4` we would have to remove it first again to be able to move `b4` onto `b3` some time (recall that blocks that are not clear cannot be moved in the Blocks World).

A conjunctive goal puts *more constraints* on how that goal can be achieved than when we treat the separate sub-goals as independent goals. Suppose we start out in an initial state, for example, with three blocks where we have `on(b1,table)`, `on(b2,table)`, and `on(b3,b1)`. We will compare the options that an agent has to achieve the conjunctive goal `on(b1,b2), on(b2,b3), on(b3,table)` with the options it has to achieve the three independent goals `on(b1,b2)`, `on(b2,b3)`, and `on(b3,table)`. The key observation to make is that in order to achieve the first, conjunctive goal we *must* realize an intermediate state where block `b2` sits on block `b3` which sits on the table.[7] This is not necessary, however, for achieving the three independent goals. One way of achieving these goals is to first move `b2` on top of `b3` (which achieves the second goal `on(b2,b3)`), then move `b2` to the table again, then move `b3` to the table (which achieves the third goal `on(b3,table)`), and, finally, move `b1` on top of `b2` (which achieves the first goal `on(b1,b2)`). Of course, it would have been more efficient to first move `b3` to the table as doing so would only have required three moves to achieve the three goals. But the point is that for achieving the three independent goals it is not necessary to do so.

From a logical point of view, the dot `.` operator in the **beliefs** and **knowledge** sections has the same meaning as the conjunction `,` operator. That is, the individual facts in a belief base can be combined in one big conjunction which would represent the same information. (It should be noted, though, that the grammar does not allow the use of the comma operator to separate simple facts in a **beliefs** or **knowledge** section as that is not ok in a Prolog program.) The meaning of the dot and comma operators, however, is very different in a **goals** section. In the **goals** section

[7]This observation is related to the famous *Sussman anomaly.* Early planners were not able to solve simple Blocks World problems because they constructed plans for sub-goals that could not be combined into a coherent 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. [26].

the conjunction operator is used to indicate that facts are part of a *single* goal which need to be achieved simultaneously whereas the dot operator is used to indicate that an agent has *multiple different* goals that need not be achieved simultaneously.

Goals in a **goals** section are best viewed as restricted Prolog queries rather than Prolog programs. Different from Prolog queries which can contain negative literals, negative literals cannot be (part of) a goal when Prolog is used as knowledge representation language. The reason is that Prolog databases do not support storing *negative* literals (which makes sense as `not` is a negation as failure operator). As a result, a goal to achieve a state where a condition does *not* hold cannot be explicitly specified. An additional restriction on goals as queries is that goals must be *ground*. That is, it is not allowed to use goals that contain free variables. The main reason for not allowing this is that it is not clear what it means to achieve a goal with a free variable.

Goals with Free Variables?
What would a goal with a free variable such as `on(X,table)` mean? In Prolog, this depends on whether `on(X,table)` is considered to be a fact or to be a query:

- Facts with free variables are implicitly *universally* quantified. This would suggest that the goal `on(X,table)` should be taken to mean that *everything* should be put on the table. (One issue that would arise with this reading is that the agent would need to be able to compute what *everything* is but it is not clear how an agent can do that.)

- Queries with free variables are implicitly *existentially* quantified. This would suggest that the goal `on(X,table)` should be taken to mean that *something* should be put on the table.

Because we need to be able to store a goal in a database (and we can only store facts but not queries in a Prolog database) *and* we want to be able to check (i.e., query) whether a goal is believed by an agent (to check whether the goal has been achieved), we are in trouble. A goal with a free variable would need to be treated as a universal fact as well as a query but clearly we cannot have it both ways. If we use Prolog as knowledge representation language, it is not possible to support goals with free variables.
Even though we cannot use free variables in a goal, we can make the agent want to move *all* blocks to the table, for example, by the rule:

```
forall bel(block(X)) do adopt(on(X,table))
```

When this rule is applied, the agent will adopt a goal for each block to move it to the table. Note, however, that this does not mean that the agent wants all of the blocks to be on the table *simultaneously*. Whether the agent will actually move all blocks on the table depends on the other rules in the agent's program.
We can also make the agent adopt a single goal for moving *some* block to the table by:

```
if bel(block(X)) then adopt(on(X,table))
```

When this rule is applied, the agent will adopt a single goal to move some arbitrary block to the table.

Whereas the sub-goals of a single goal need to be realized simultaneously to achieve the complete goal, two distinct goals may (or even need) be achieved at different times. Two distinct goals need to be achieved at different times when they cannot be realized simultaneously. For example, a Blocks World agent might have two goals `on(b1,b2)` and `on(b2,b1)`. Obviously, these goals cannot be achieved simultaneously but they can be achieved one after the other. In other words, it is perfectly fine if two or more goals of an agent are *inconsistent*, assuming that it is possible to realize each of the goals one after the other. This is another key difference between goals and beliefs as it does not make a lot of sense to have inconsistent beliefs in a belief base of an agent.

Goals that need to be realized at some future point in time are also called *achievement goals*. The goals of a GOAL agent are achievement goals that are removed as soon as they have been achieved (see also Chapter 4). In order to achieve these type of goals, an agent needs to perform actions to *change* the current state of its environment to ensure that a *desired* state is achieved, i.e., a state that realizes the goal. The goal (state) of a Blocks World problem provides a typical example of an achievement goal. Apart from achievement goals there are many other types of goals. At the opposite of the goal spectrum we find so-called *maintenance goals*. In contrast with achievement goals, maintenance goals without deadlines should never be removed from an agent's goal base.[8] In order to achieve a maintenance goal, an agent needs to perform actions, or, possibly, refrain from performing actions, to continuously *maintain* a particular condition in its environment, i.e., to ensure that this condition does *not* change. A typical example of a maintenance goal is a goal of a robot to maintain a minimum level of battery power (which may require the robot to return to a charging station). Other types of goals are, for example, *deadline goals* that require an agent to achieve a particular condition before some other condition becomes true. A robot may want, for example, to return to its home base before the night falls.

3.2 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 which determine the actions that an agent will perform (see Chapter 5).

3.2.1 Mental Atoms

Mental atoms are used for inspecting different databases of the agent. There are two basic mental atoms. Mental atoms of the form **bel**(φ) are used for inspecting what follows from the agent's beliefs. Mental atoms of the form **goal**(φ) are used for inspecting what follows from the agent's goals. The condition φ in both cases is a query specified in line with the rules of the knowledge representation language that is used (e.g., when using Prolog, φ must be a valid Prolog query).

Belief Atoms A belief atom **bel**(φ) means that the the agent believes that φ. For example, given the initial beliefs specified above, we can infer that **bel**(on(b1,b2)) holds; that is, the agent believes that block b1 sits on top of b2. An agent not only believes what follows from its belief base but also what follows from its knowledge base. For example, given the knowledge base we specified above, it also follows that **bel**(clear(table)) holds; that is, the agent believes that the table is clear. Even stronger, an agent believes everything that follows from its beliefs *in combination with* its knowledge. This allows an agent to infer its beliefs by also using the rules present in the **knowledge** section. More precisely, **bel**(φ) holds whenever φ can be derived from the content of the agent's belief base combined with the content of its knowledge base. Continuing the example of Figure 3.1, by combining the initial beliefs and the knowledge we specified above, it follows that in the initial state **bel**(clear(b1)) holds; that is, the agent believes that block b1 is clear. This can be inferred by using the rules clear(X) :- block(X), not(on(_, X)) and block(X) :- on(X, _) which are present in the agent's knowledge base and from the absence of any information in the belief base that suggests that there is a block on top of b1.

Goal Atoms A goal atom **goal**(φ) means that the agent has a goal that φ. For example, given the initial goal specified above, we can infer that **goal**(on(b1,b5)); that is, the agent wants block b1 to sit on top of b5. Simply put, an agent has a goal if it follows from its goal base. But inferring which goals an agent has works somewhat differently from inferring which beliefs

[8]One of the main differences between an achievement and a maintenance goal is that the former is removed when achieved whereas the latter should remain in the database when the goal condition holds. It is possible to prevent automatic removal of a goal from a goal base by adding a predicate to the goal that is never achieved, e.g., maintain. This trick allows to implement a kind of maintenance goal in GOAL.

an agent has. We can illustrate this by checking whether **goal**(on(b1,b5), on(b2,table)) follows from the goals specified in the two different **goals** sections that we discussed above. For convenience, we repeat both sections here. The first section contains a *single conjunctive goal*. This is the goal that represents the goal state of our Blocks World problem in Figure 3.1.

```
goals{
  on(b1,b5), on(b2,table), on(b3,table), on(b4,b3), on(b5,b2), on(b6,b4), on(b7,table).
}
```

As you probably would expect, **goal**(on(b1,b5), on(b2,table)) follows from the conjunctive goal above. Clearly, if the agent wants to achieve that b1 sits on top of b5 *and* that b2 sits on the table *and* that b3 sits on the table, etc., it follows that the agent wants to achieve *simultaneously* that b1 sits on top of b5 *and* that b2 sits on the table. In fact, given this goal, we have that **goal**(on(b1,b5),on(b2,table),on(b3,table),on(b4,b3),on(b5,b2),on(b6,b4), on(b7,table)) holds.

The second section contains *seven individual goals*.

```
goals{
  on(b1,b5). on(b2,table). on(c3,table). on(b4,b3). on(b5,b2). on(b6,b4). on(b7,table).
}
```

In contrast with the conjunctive goal it does *not* follow from these seven goals that we have **goal**(on(b1,b5), on(b2,table)). The point is that we cannot derive from these individual goals that the agent wants to *simultaneously* achieve that b1 sits on top of b5 *and* that b2 sits on the table. Clearly, both **goal**(on(b1,b5)) and **goal**(on(b2,table)) hold given the first two of the seven goals above. But this does not mean that the agent wants to achieve these two goals *at the same time*; specifying these goals as separate goals leaves room for quite the opposite, i.e., that the agent might prefer achieving these goals at different times. As we have seen above, there is nothing that prevents the agent from doing so whereas the conjunctive goal imposes more constraints on what an agent can do.

How does an agent then derive which goals it has? It evaluates **goal**(φ) by checking whether φ follows from *one* of the goals in its goal base. That is, **goal**(φ) holds whenever φ can be derived from *a single goal* in the goal base of the agent. Clearly, on(b1,b5), on(b2,table) follows from the conjunctive goal above but not from any one of the seven goals specified in the second **goals** section.

Another question that still remains to be answered is whether we can infer more from an agent's goals than what directly follows from the goal itself. To illustrate what we mean, consider the notion of a *tower* (or stack) of blocks. Given the conjunctive goal above, it seems clear that the agent wants to construct two towers of three blocks, i.e., a tower with b1 on b5 on b2 on the table and a tower with b6 on b4 on b3 on the table, and a single block b7 on the table. It is not too difficult to provide a definition of a tower predicate in Prolog. If we consider a single block to be a tower as well (admittedly a borderline case but useful to keep our definition simple) and use lists such as [b1,b5,b2] to represent a tower, then the following clauses recursively define our concept of tower:

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

These two rules specify when a list of blocks [X| T] is considered to be a tower. The first rule requires that the basis of a tower is grounded on the table and says that a single block on the table is a tower. The second rule says that whenever [Y| T] is a tower, we have another tower [X, Y| T] if we can find a block X that sits on top of Y. Note that the rules do not require that block X in a tower [X| T] is clear. This means that any stack of blocks that is part of a larger tower is also considered to be a tower. For example, if we have `tower([b1,b5,b2])`, then we also have `tower([b5,b2])` (but not `tower([b1,b5])`).

We would now like to use these rules to conclude that our Blocks World agent wants to build a tower `[b1,b5,b2]`; that is, we would like to show that **goal**`(tower([b1,b5,b2]))` holds. As goals are queries, we cannot include these rules into the goal base directly. But we should not do so either. As we discussed above, rules that define concepts such as that of a tower should be included in the knowledge base of an agent by making them part of a **knowledge** section. One benefit of including the rules into the **knowledge** section is that we can also use the `tower` predicate in combination with beliefs. As we will show below, adding the tower concept as knowledge for reasoning about the Blocks World is also very useful for defining when a block is *in position* or *misplaced*.

Similar to how is verified that **bel**(φ) holds, **goal**(φ) holds whenever φ can be derived from the agent's goal base in combination with the content of its knowledge base. As we saw above, however, only one goal at a time is used to infer the goals of an agent. To be precise, **goal**(φ) holds if *there is a goal* in the agent's goal base that *in combination with* the content of its knowledge base allows to derive φ. That is, in order to show that `goal`(φ) holds we need to find *a single goal* in the agent's goal base from which in combination with the content of its knowledge base we can derive φ.[9]

Before we discuss some examples of goals that follow from the agent's initial mental state, we first combine the various sections that we specified above and that are used to create the initial mental state. By putting everything together we obtain part of an agent program that specifies the initial mental state of our Blocks World agent. Below we also include the `tower` predicate in the **knowledge** section; Table 3.1 combines the resulting **knowledge**, **beliefs**, and **goals** sections.

```
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).
   % a tower is any non-empty 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(b1,b2). on(b2,b3). on(b3,table). on(b4,b5). on(b5,table). on(b6,b7). on(b7,table).
}
goals{
  % a single goal to achieve a particular Blocks World configuration.
  % assumes that these blocks are available in the Blocks World.
  on(b1,b5), on(b2,table), on(b3,table), on(b4,b3), on(b5,b2), on(b6,b4), on(b7,table).
}
```

Table 3.1: Mental State of a Blocks World Agent That Represents Figure 3.1

We start by revisiting the question whether we can show that **goal**`(tower([b1,b5,b2]))` holds. In order to show this, we need to find a goal from which, together with the agent's knowledge, we can derive `tower([b1,b5,b2])`. There is only a single goal present in the mental state specified in Table 3.1 that we can use. But this goal in combination with the rules for `tower` are sufficient to derive the goal: we can first show that we can derive `tower([b2])` because we have

[9]This reading differs from that provided in [12] where the **goal** operator denotes *achievement goals*. This means that **goal**(φ) implies `not(`**bel**(φ)`)` because a goal is an achievement goal only if the agent *does not believe that* φ. The **goal** operator introduced here is a more basic operator that is unrelated to the agent's beliefs and that does not imply `not(`**bel**(φ)`)`. Our **goal** operator can be used in combination with the **bel** operator to define achievement goals as we will see below.

`on(b2,table)`; then we can show `tower([b5,b2])` by using `on(b5,b2)` and finally we show that `tower([b1,b5,b2])` follows. Similarly, we can show that **goal**`(tower([b6,b4,b3]))` and **goal**`(tower([b7]))` hold. Of course, we can also use other parts of the knowledge base. For example, we can use the rule for `clear` to show that **goal**`(clear(b1))` holds.

3.2.2 Mental State Conditions

Mental state conditions are composed from mental atoms. The mental atoms **bel**(φ) and **goal**(φ) are the most basic mental state conditions. Mental atoms can be negated using the **not** operator. The negated atoms **not**(**bel**(φ)) and **not**(**goal**(φ)) are also mental state conditions. Plain and negated mental atoms are also called *mental literals*. A mental state condition then simply is a conjunction of one or more mental literals. Example mental state conditions are:

1. **bel**`(clear(b4))`
2. **goal**`(clear(b1))`
3. **not**(**bel**`(tower([X,Y,Z,b4])))`
4. **goal**`(on(b2,table)),` **not**(**bel**`(on(b2,table))`
5. **goal**`(on(b7,table)),` **bel**`(on(b7,table))`

The first item says that the agent believes that block `b4` is clear; the second that the agent wants `b1` to be clear; the third that the agent does not believe there is a tower of four blocks with `b4` as base; the fourth that the agent wants `b2` to be on the table and does not believe this already to be the case; and, the fifth means that the agent wants `b7` to be on the table and believes that this is already the case.

Disjunctive Mental State Conditions?
It is not possible to use disjunction in a mental state condition. Mental state conditions are used by an agent to make decisions and provide the reasons for selecting a particular action. Intuitively, therefore, a disjunctive mental state condition of the form $\psi_1 \vee \psi_2$ would provide two different reasons for selecting an action: whenever ψ_1 is the case or whenever ψ_2 is the case that would be a reason for selecting the action. In cases like these, an alternative would be to use two or more action rules for selecting an action where each rule checks one of the possible reasons for selecting the action (see Chapter 5 for more on action rules). Also note that negation can be distributed over conjunction and disjunction: $\neg(\psi_1 \wedge \psi_2)$ is equivalent to $\neg\psi_1 \vee \neg\psi_2$. It follows from this transformation and the fact that instead of disjunction we can equivalently use multiple action rules that we do not need disjunction for our purposes.

Goals as in the fourth example item above that still need to be achieved because they have not yet been realized are also called *achievement goals*. Achievement goals provide an important reason for choosing to perform an action: what the agent wants requires change. We introduce a new operator **a-goal**(φ) for representing achievement goals. The operator is defined as follows:[10]

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

The **a-goal** operator can be used to express several important concepts in the Blocks World. The operator is exactly what we need to express that a block is *misplaced*, for example. It is important to be able to conclude that a block is misplaced because if a block is misplaced, then the agent will have to move the block to the place it should be in. That is, the block is not right

[10]See [30] for a discussion of this definition.

now but needs to be brought *in position*. Informally, a block is misplaced if the agent believes that the block's current position is different from the position the agent wants it to be in.[11] Note that we would only say that a block is misplaced if an agent wants the block to be somewhere else than it is, i.e., has a goal that involves the block which has not been realized yet. How do we express where we want a block to be in a goal state? The `tower` predicate is useful for this purpose as it allows us to express that a block needs to be in the right position in a particular tower in the goal state. (Recall that the exact position of a tower is irrelevant in our version of the Blocks World.) The goal part of expressing that a block `X` is misplaced then can be expressed by **`goal`**`(tower([X| T]))`. This expresses that for block `X` to be in position it should be on top of a tower `T`. That a block is misplaced if we do not believe it to be where it should be can be expressed by **`not`**`(`**`bel`**`(tower([X| T])))`. Combining both parts and using the definition for **a-goal**, we find that **`a-goal`**`(tower([X| T]))` expresses that a block `X` is misplaced. The concept of a misplaced block is important because - assuming goal states are complete - only misplaced blocks need to be moved. We will use the mental state condition that expresses that a block is misplaced in Chapter 5 for defining a strategy for solving Blocks World problems.

The condition **`a-goal`**`(tower([X, Y| T])),` **`bel`**`(tower([Y| T])` provides another very useful example of a mental state condition for solving Blocks World problems. 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 its goals. Such a move therefore is also called a *constructive move*. Note that after making a constructive move with a block, that block will not have to be moved again. The mental state condition will be used in Chapter 5 to specify a rule for solving Blocks World problems.

Another useful type of mental state condition are conditions of the form **`goal`**`(`φ`),` **`bel`**`(`φ`)`. These express that a sub-goal has been achieved. The operator **`goal-a`**`(`φ`)` is an abbreviation that can be used to state that a sub-goal has been achieved.

$$\mathbf{goal\text{-}a}(\varphi) \overset{df}{=} \mathbf{goal}(\varphi),\ \mathbf{bel}(\varphi)$$

When we instantiate φ with `tower([X| T])`, we obtain **`goal-a`**`(tower([X| T]))` which expresses that the current position of a block `X` corresponds with the position it has in the goal state. Here, φ must be a sub-goal that is achieved and that is part of a larger goal that has not been completely achieved. We call such a sub-goal a *goal achieved*. The **goal-a** operator cannot be used to express that a complete goal in an agent's goal base has been achieved because upon completion a goal is immediately removed from the agent's goal base.[12]

To conclude, we mention the use of the special mental literal **`not`**`(`**`goal`**`(`**`true`**`))` for checking that an agent has *no* goals. We can see that an agent cannot have any goals if **`not`**`(`**`goal`**`(`**`true`**`))` holds by realizing that **`goal`**`(`**`true`**`)` only holds if an agent has at least one goal. That is, **`goal`**`(`**`true`**`)` holds if *there is a goal* from which **`true`** follows (in combination with the agent's knowledge but that is not important here). In other words, **`goal`**`(`**`true`**`)` fails if there is no goal and **`not`**`(`**`goal`**`(`**`true`**`))` succeeds if the goal base of the agent is empty.

[11] In ordinary language, the difference between *knowledge* and *belief* makes a difference here. We normally would only say that something is misplaced if we *know* that the object is in a position different from where we want it to be. For an item to be misplaced it is necessary that an agent's beliefs about the item's position correspond with the actual position of the item, i.e., the agent knows the item's position. If the item would be 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. Because we will assume our agent's beliefs to match with what is the case, our agent knows when a block is misplaced.

[12] Another important concept in the Blocks World, 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 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 important for analysing Blocks World problems as 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.

3.3 Notes

The mental states discussed in this chapter consist of the knowledge, beliefs and goals of an agent about its environment. An agent whose mental state only represents its environment is a first-order intentional system (cf. Chapter 2). Such systems do not allow for, e.g., second-order beliefs about beliefs or goals. Agents that use a first-order intentional system maintain representations of their environment but not about their own or other agent's beliefs. In a multi-agent system where other agents interact with the agent and its environment, it may be useful to also represent in the agent's mental state which other agents are present and what they do, believe, and want. This would require the ability to represent other agent's beliefs and goals. To this end, an agent that uses second- or higher-order intentional systems is needed. Agents would also require some kind of Theory of Mind, to be able to reason with their beliefs about other agents. In Chapter 7, we will introduce the tools to represent other agent's mental states.

The introduction of a third database for storing static *knowledge* is a distinguishing feature of GOAL. Conceptually, the idea is that static domain and conceptual knowledge can be used in combination with both the agent's beliefs and its goals (of course, we cannot combine beliefs and goals if we want to avoid inconsistency). In early versions of the language we found ourselves having to duplicate rules into both the belief and goal base. Duplication of code is undesirable as it increases the risk of introducing bugs (e.g., by changing code at one place but forgetting to change it also in another place).

In the current implementation of GOAL the knowledge, beliefs, and goals are represented in Prolog [58, 59]. 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 [17], the Planning Domain Definition Language (PDDL; [26]), or other, similar such languages, or possibly even Bayesian Networks [46]. The assumption that we will make throughout this guide is that an agent uses a *single* knowledge representation technology. For some preliminary work on lifting this assumption, we refer the reader to [22].

3.4 Exercises

```
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).
  % a tower is any non-empty stack of blocks that sits on the table.
  tower([X]) :- on(X, table).
  tower([X, Y| T]) :- on(X, Y), tower([Y| T]).
  % a block is above another block Y if it sits on top of it or
  % sits on top of another block that is above Y
  above(X, Y) :- on(X, Y).
  above(X ,Y) :- on(X, Z), above(Z, Y).
}
beliefs{
  on(b1,b4). on(b2,table). on(b3,table). on(b4,b2). on(b5,b1).
}
goals{
  % a single goal to achieve a particular Blocks World configuration.
  % assumes that these blocks are available in the Blocks World.
  on(b1,b2), on(b2,b3), on(b3,b4), on(b4,table).
}
```

Table 3.2: Mental State

3.4.1

Consider the mental state in Table 3.2. Will the following mental state conditions succeed? Provide all possible instantiations of variables if the condition succeeds, or else provide 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]))`

3.4.2

As explained in the chapter, we can use **`a-goal`**`(tower([X| T))` to express that block `X` is *misplaced*. It follows from this that a block can only be misplaced if it is part of the goal state. This is not the case for block `b5` in Table 3.2. In order to reach the agent's goal, however, we nevertheless need to move block `b5` because it is *in the way* of moving other blocks. In the example, block `b5` sits on top of the misplaced block `b1` and we first need to move `b5` before we can move `b1`.
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. (Hint: Keep in mind that the block below `X` may also not be part of the goal state!) Do not introduce new predicates but only use predicates available in Table 3.2.

3.4.3

We already discussed that we can do without the `block` predicate. We can simply use the body of the rule we introduced for defining the predicate itself. The predicate `clear` can also be eliminated and has been introduced mainly to facilitate reasoning about the Blocks World and to enhance readability of the program. In this exercise, we will modify the agent program `stackbuilder.goal` such that it no longer uses the `block` or `clear` predicates:

1. Start by removing the definitions of both predicates from the **knowledge** section in the program. What parsing error is generated by this change?
2. We need to fix the precondition of the `move` action in the **actionspec** section. Replace the occurrences of the `clear` predicate in the precondition with another query that means the same. (Hint: Observe that it is not necessary to add a check whether `X` is a block in the precondition because this already follows from the conjunct `on(X, Z)`.)
3. Which version of the `stackbuilder.goal` program do you like better? Why?

Chapter 4

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: [15]

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. An agent needs to be informed about the *precondition* of an action, i.e., when an action can be performed, and about the *postcondition* of an action, i.e., the effects of performing the action. Agents need to know this so they can take this information into account when selecting an action. An agent should only select an action that can be performed in the current state and should select actions that will contribute to achieving its goals. The main purpose of this chapter is to explain how an agent can be provided with knowledge about the actions it is able to perform by providing the agent with *action specifications*. An action specification is used to capture the conditions that need to be true to successfully execute an action as well as the conditions that will change in the environment. In order to provide the agent with knowledge about the actions it can perform, a programmer needs to write action specifications for all actions that the agent can perform to change its environment. In this chapter we focus on the preconditions and effects of actions. Chapter 6 discusses other aspects of performing actions in an environment such as the fact that an action may take time to complete.

4.1 Action Specifications

Each action that can be performed by an agent to change its environment needs to be specified in the agent's program. Actions that can change the agent's environment are also called *environment actions*. If an agent is connected to an environment, that environment will make actions available that the agent can perform for changing it (see also Chapter 6). An agent needs to be informed about environment actions in order to be able to understand when it can perform these actions and what the effects of these actions are. Only environment actions must be specified. An agent knows what built-in actions that are part of the agent programming language do (see Section 4.2). Other *user-defined actions* that are not available in the environment but that are introduced for convenience may be specified in an agent program as well. We will see an example of a user-specified action below.

Action Specification An action specification specifies the *name* of the action and its *parameters*, the *precondition* of the action, and its *postcondition*. The pre- and post-condition are specified in the knowledge representation language that is used in the agent program. The precondition is a query for checking whether the action can be performed. The postcondition is a condition that specifies what has changed after the action has been performed. In other words, the postcondition captures the effects of the action. An action specification has the following structure:

```
<actionname>(<parameters>){
   pre{ <precondition> }
   post{ <postcondition> }
}
```

An action specification starts with a declaration of the action: It specifies the `<actionname>` of the action and the number of parameters the action has. Parameters of an action are *formal* parameters and must be specified between brackets as a comma-separated list of variables of the knowledge representation language. For example, when using Prolog, capital letters must be used to specify action parameters. Parameters are optional, i.e., an action does not need to have parameters. In that case, only the action name (without brackets) should be specified.

It is important that the name and the number of parameters of an environment action exactly match with the actual name and number of parameters of the action that is made available by the environment. An environment action is sent to the environment when an agent chooses to perform it. But if the name or parameters do not match, the environment will not recognize the action and may behave in unexpected ways.

We will continue to use the Blocks World environment that was introduced in Chapter 3 as an example and provide an action specification for the action that is made available by this environment for controlling the gripper. The gripper allows an agent to move blocks. The name of the action in the Blocks World for moving a block is `move`. This action has two parameters: The first parameter indicates which block should be moved and the second parameter indicates where the block should be moved to (either another block or the table). `move(X,Y)` thus would be a correct declaration of this action, assuming that we use Prolog for specifying variables.

The gripper in the Blocks World can pick up at most one block at a time. In other words, it is not possible to move a stack of blocks and the block that is moved should be clear. The place where the block is moved to should also be clear. This means that if a block is moved onto another block or the table, that block or the table should be clear. These conditions are preconditions of the `move` action which we can specify using the `clear` predicate introduced in Chapter 3.

The effect of moving a block is that the block is no longer in its old position and is now sitting on top of the object it has been moved to. The old position needs to be removed and the new position needs to be added to the beliefs of the agent after completing the move action. Recall that we represented the position of a block by an `on` fact in Chapter 3. In order to be able to remove the old position, we need to retrieve the block's initial position before the `move` action is performed. We do so by including a query for this position in the precondition of the `move` action. By negating this fact in the postcondition, we indicate that it should be removed. The new fact that should be inserted into the agent's beliefs is included as a positive literal in the postcondition.

Putting all the pieces together results in the following action specification for the `move` action:

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

The query `on(X,Z)` in the precondition retrieves the old position of block `X` from the agent's belief base. The condition **`not`**`(on(X,Y))` has been added to prevent the agent from trying to move a block that is already on the table to the table again (note that a block on the table may be clear and the table itself is assumed to be always clear in our variant of the Blocks World).

As is common in our version of the Blocks World, we abstract away pretty much all physical aspects of real robot grippers. The gripper in the Blocks World is modelled as a robot arm that can perform a single action of picking up and putting down one block to a new destination in one go. We also abstract away time. That is, the picking up and putting down are taken as a single action that does not take any time and can be performed *instantaneously*. Actions that take time before they are completed in an environment are discussed in Chapter 6.

Action Specification Section Actions are specified in the action specification section of a module. The **actionspec** keyword is used to indicate the beginning of an action specification section. An **actionspec** section consists of one or more action specifications. As a rule of thumb, environment actions are best specified in the **actionspec** section of the **init** module. This module is the first module that is executed after launching the agent and is used for initializing an agent. Actions that have been specified in the **actionspec** section of the **init** module can be used in all modules that are part of the agent program. In other words, the declarations of actions that have been specified in the **init** module have *global* scope. Actions that are specified in other modules can only be used within the scope of that module. An action specification section with a single action specification for the `move` action looks like:

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

Internal Actions If an agent is connected to an environment, by default, actions specified in an **actionspec** section are sent to that environment. It sometimes can be convenient, however, to specify actions that are not sent to the environment that an agent is connected to. To differentiate these user-defined actions from environment actions, you can use the **`@int`** keyword. By adding this keyword directly after the declaration of the action, the agent is instructed to not sent the action to its environment. For example, the following action specification introduces a new action `skip` that is declared to be an internal action:

```
skip@int{
  { true }
  { true }
}
```

As usual, a `skip` action should always be enabled and should have no effect. This is achieved by using **true** as a precondition which indicates that `skip` can always be performed. The post-condition **true** indicates that the action has no effect. It would also have been possible to use an empty postcondition {}. We prefer to use **true** as a postcondition, however, to make explicit that we did not forget to specify the postcondition.

Multiple Action Specifications for A Single Action It is possible to provide more than one action specification for a single action. It is useful to include multiple action specifications for one and the same action in order to *distinguish cases*. For example, in the Wumpus World [51], an agent can perform an action called `turn`. The turn action allows the agent to either turn left or right. Of course, the effects of turning left or right are different. After turning left the orientation of the agent has changed 90 degrees leftwards and after turning right 90 degrees rightwards.

We can only differentiate between turning left and right and specify the right postconditions for each case, however, by providing two action specifications for the action. Assuming that we use `orientation` to store the agent's current orientation and `left(Old,New)` to compute the `New` orientation after turning left from the `Old` orientation and `right(Old,New)` to do the same for a right turn, we can specify the `turn` action as follows:

```
actionspec{
   ...
   turn(X) {
      pre{ X=left, orientation(Old), left(Old,New) }
      post{ not(orientation(Old)), orientation(New) }
   }
   turn(X) {
      pre{ X=right, orientation(Old), right(Old,New) }
      post{ not(orientation(Old)), orientation(Old,New) }
   }
}
```

Note that even though the `turn` action can always be performed, the preconditions in the action specifications for `turn` are not empty. In order to be able to update the orientation of the agent after performing the `turn` action, we need to retrieve the old and new orientation in the precondition. The precondition is used to compute the new orientation from the old orientation by means of the `left` and `right` predicates (these predicates should be specified in the **knowledge** section of the agent's **init** module).

4.1.1 Preconditions

As we have seen above, the keyword **pre** is used for specifying a precondition of an action. Preconditions have two main uses: Checking whether an action is *enabled* and retrieving information that needs to be *updated* after performing the action. Here we discuss in somewhat more detail how preconditions are evaluated and used.

Preconditions are queries of the knowledge representation language that is used in the agent program. For example, when using Prolog, a precondition should be a Prolog query. A precondition φ is evaluated by checking whether φ can be derived from the agent's belief base combined with its knowledge base. Free variables in a precondition may be instantiated when evaluating the condition exactly like a Prolog program may return instantiations of variables.

Enabled An action is said to be *enabled* whenever its precondition is believed to be the case by the agent.[1] For example, the precondition of the `move` action above specifies that the action `move(X,Y)` is enabled when both `X` and `Y` are clear (`clear(X), clear(Y)`) and `X` does not already sit on top of `Y` (**not**`(on(X,Y))`; this can only be the case if `Y=table`).

If we would have liked, we could also have added **not**`(X=Y)` to the precondition to make clear that a block cannot be moved on top of itself and prevent the agent from ever trying to do so. Performing, e.g., the action `move(b1,b1)` will either make the environment behave in unexpected ways or make the agent believe facts that are not true in its environment. This condition is redundant, however, if we make sure that whenever the agent decides to perform a `move` action it will never select `move(X,X)` for some `X`. As a rule, however, it is good to try to provide action specifications that are as *complete* as possible and include all conditions that are needed for checking that an action is enabled.

Although it is best practice to include all conditions that are required to verify that an action is enabled, a precondition should not include conditions that are not really needed to check that action will execute successfully. In particular, conditions that are used to identify when it would

[1] Note that because an agent may have false beliefs an environment 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.

be a good idea to perform an action should not be included in a precondition. Such conditions should instead be part of the condition of an action rule in a **program** section. For example, it is a bad idea to include a condition **not**(on(X,table)) that checks whether a block X is not on the table in the precondition of move. Such a condition can be useful for specifying a strategy that first moves all blocks to the table but it does not indicate when the action is enabled at all. In fact, including a condition like this would prevent the agent from performing move actions that can actually be performed in the environment.

Actions Must be Closed In principle, actions are enabled when their preconditions hold. But in addition the action must also be closed. That is, all variables that occur in the parameters as well as in the postcondition of the action must have been fully instantiated before the action is executed. For example, the variables X, Y, and Z all need to have been fully instantiated before the move action can be performed. The reason is that it is not clear what it means to execute an action with a a variable as parameter or a postcondition that is not closed. If a parameter is not closed, it is not clear *which* action should be performed. And if a postcondition is not closed, it is not clear *what* effects performing the action has. All actions, including environment, user-defined, and built-in actions must be closed before they are executed.

Completely Instantiated Actions
The reason that the variables that occur in an action's parameters or in its postcondition must be fully instantiated is that it is not clear what it means to execute the action otherwise. 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 often it is not even clear from which items an agent can select. Moreover, it is often a bad idea or just plain right impossible to apply an action to all items. For example, it is clear that we should exclude any attempts to move the table in the Blocks World. But an agent should clearly also not try to move natural numbers (one option is to instantiate a variable with an integer). But how should we make sure that the agent knows this?
For similar reasons, it is unclear what it means to apply a postcondition with free variables. Postconditions are used for updating the agent's beliefs. But what, for example, does it mean to update an agent's beliefs with the condition on(X,b1)? Should we add all instances of the fact? Or only one, selected at random? And how should we prevent the beliefs of an agent to become inconsistent in either of these cases?
One way to prevent that pre- or post-conditions are not closed even when all action parameters are, is to require that all variables in the action specification of an action also appear as parameters of the action (as is done in STRIPS, cf. [51]). This requires, for example, that instead of the move(Block,To) action specified in the main text we need to introduce an action moveFromTo(Block,From,To) because we then also need to include the From variable as an action parameter. The downside of this restriction is that we can no longer use the precondition of an action to retrieve the current position of a block but need to perform such computations elsewhere in the agent program. Moreover, if an environment only makes a move action available with two parameters as in our version of the Blocks World, there would be no way of specifying this action. (Recall that the number of parameters of an action must match those of the action available in the environment.)

Retrieving Information Observe that the condition on(X,Z) in the precondition of the move action does *not* specify a condition that needs to be true in order to be able to perform the move action. Assuming that the agent always knows the position of a block X, i.e., has a fact on(X,Y) for some Y in its belief base, this condition will always succeed and does not tell us anything about whether the action is enabled or not. This condition has been included instead to retrieve information from the agent's belief base that is needed for updating the agent's beliefs after performing a `move` action. A new variable `Z` is used for retrieving the block or table that block `X` is sitting on *before* the action is performed. This information can be retrieved by means of the precondition because it is evaluated before performing an action. It is clear that the position of the block will have changed after performing the action and the old position then must be removed to keep the agent's beliefs up to date.

4.1.2 Postconditions

The keyword **post** is used for specifying a postcondition of an action. The main use of an action's postcondition is that it allows the agent to *update* its beliefs to keep track of the changes that result from performing the action.

Postconditions like preconditions are specified in the knowledge representation language that is used in the agent program. A postcondition must be a conjunction of literals, i.e., positive and negated facts. For example, when using Prolog, a postcondition should be a conjunction of Prolog literals. Rather than being evaluated a postcondition is applied. In short, applying a postcondition means to update an agent's mental state.

Updating an Agent's Beliefs A postcondition of an environment action is used for specifying the effects that an action has on its environment. A postcondition of a user-defined action that is not sent to an environment specifies how the agent's mental state should be updated when the action is performed. In both cases, the postcondition is used for updating the agent's mental state. The effects on an agent's goal base are discussed in Section 4.1.3; here we discuss how the beliefs of an agent are updated by a postcondition. The aim of updating the agent's beliefs with a postcondition is to make sure that the agent believes the postcondition. This is achieved by *first* removing all facts from the agent's belief base that are negated in the postcondition and *thereafter* adding all positive facts in the postcondition to the belief base. This ensures that all positive facts in a postcondition are believed by the agent, and, if facts occur only once, that all negated facts in the postcondition are no longer believed after applying the postcondition. In addition, goals that have been *completely* achieved are removed from the agent's goal base (see Section 4.1.3 below).

Closed World assumption
The approach used for updating the mental state of an agent after performing an action is related to the Closed World assumption that we discussed in Chapter 3. It assumes that the database of beliefs of the agent consists of *positive* facts only. Negative information is not inserted into this database but instead can only be inferred by making the Closed World assumption (or, more precisely, negation as failure if 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 a Closed World assumption and allows to add negative literals to an agent's belief base [51].

For example, suppose that the agent executes the action `move(b1,b2)` and `b1` sits on top of `b3` initially. The instantiated postcondition for the `move` action in that case would be **`not`**`(on(b1,b3)), on(b1,b2).` Applying this postcondition then means that the fact `on(b1,b3)` is removed from the agent's belief base because the fact occurs in the negative literal **`not`**`(on(b1,b3))` and the fact `on(b1,b2)` is added to the belief base because it occurs in the positive literal `on(b1,b2)`. The agent thus will no longer believe `on(b1,b3)` but instead will believe `on(b1,b2)` after applying the postcondition.

Instantaneous Actions The intuition of updating the agent's beliefs by applying the postcondition is that the agent should believe the effects after performing an action. It is important to realize, however, that this intuition only holds for actions that *take no time to complete*, i.e., only for *instantaneous* actions. The point is that upon selecting an environment action for execution, an agent *immediately* does two things: it sends the action to the environment *and* it updates its mental state by applying the postcondition. That means that an agent's beliefs will only match what is true in the agent's environment if the effects of an action are immediately realized in that environment. See Chapter 6 for more on how to specify actions that take time before they are completed.

Variables in Postconditions Recall that an action can only be performed if it is closed. Because a postcondition is not evaluated but applied, this also means that all variables in the postcondition must have been instantiated before it is applied. Variables in a postcondition can only be instantiated either by action parameters or by bindings that result from evaluating conditions in the precondition. It therefore is required that all variables that occur in a postcondition must also occur either in the action's parameters or in the precondition.

Inconsistent Postconditions It is possible that postconditions are instantiated in such a way that the postcondition becomes *inconsistent*. Ideally, we would like to prevent this from happening but inconsistent postconditions may easily arise. For example, by making only some small changes to the action specification for `move` above we can get an inconsistent instantiated postcondition. All that we need to do is to replace the condition **not**`(on(X,Y))` with the condition **not**`(X=Y)`. (Recall that we said above that **not**`(on(X,Y))` was mainly introduced for pragmatic reasons and we argued that in order to obtain a complete action specification we would need to add **not**`(X=Y)`.) In that case, the action specification for `move` would look like:

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

Given this action specification, there is nothing that prevents us from instantiating it such that `Y=Z`. If we do so, however, the postcondition would become **not**`(on(X,Y)), on(X,Y)` by substituting `Y` for `Z`. Obviously, this postcondition is a contradiction and, from a logical point of view, can never be true. However, though logically inconsistent, no big problems will be introduced when a `move` action would be executed with this postcondition. (Note that because `X` cannot be equal to `Y`, the only instantiation that would result in an inconsistent postcondition requires `Y=table, Z=table`).) The reason is that a postcondition is only applied but never evaluated. A postcondition can be viewed as a recipe for updating an agent's mental state and that recipe can always be applied. As we have seen, the procedure of applying a postcondition proceeds in two steps. In our example, we first need to remove the fact `on(X,Y)` from the belief base, and thereafter have to add the same fact `on(X,Y)`. As a result, either nothing will have changed if the fact initially was present already, or the fact `on(X,Y)` will have been added to the belief base.

Although this *procedural* semantics for applying a postcondition prevents any acute problems that might have arisen, it is considered better practice to try and prevent contradictions from arising in the first place. This provides a less pragmatic reason for including the condition `not(on(X,Y))` in the precondition of the `move` action as it will prevent the postcondition of this action from ever becoming inconsistent.

4.1.3 Updating an Agent's Goals

Intuitively, what an agent wants to achieve should not be the case yet. In other words, there is no reason for an agent to have an achievement goal that has already been completely realized. A good reason why an agent should not have such goals is that an agent that is only minimally rational should not spent any of its resources on trying to realize something that is already the case. Also see the Exercise in Section 1.3.

If we want to make sure that an agent never has an achievement goal which it believes has already been completely realized, it should be clear that by performing an action the goal base of an agent might also be affected. Because the beliefs of an agent are updated by applying the postcondition of an action, an agent may thus come to believe that one or more of its goals have been completely realized. To ensure that an agent does not have such goals, we need to remove any goals that the agent believes have been achieved from the agent's goal base. In other words, goals that have been achieved as a result of performing an action are removed from the agent's goal base.

For example, suppose an agent's current mental state consists of a belief and goal base that contain the following:

```
beliefs{
   on(b2,b3). on(b3, table).
}
goals{
   on(b1,b2), on(b2,b3), on(b3, table).
}
```

Now also suppose that the agent successfully performs the action `move(b1,b2)` in this mental state. After only applying the postcondition to the agent's beliefs, the following updated mental state would result:

```
beliefs{
   on(b1,b2). on(b2,b3). on(b3, table).
}
goals{
   on(b1,b2), on(b2,b3), on(b3, table).
}
```

But in this state the agent's only goal is implied by the beliefs that it has. We argue that there is no point in keeping this achievement goal and in this state the goal should be removed from the agent's goal base. The reason for removing the goal is that the agent does not need to and should not invest any resources in it any more. The removal of goals that have been completely achieved has been built into GOAL and whenever a goal is believed to have been realized by an agent that goal automatically is removed. Continuing our example, this means that the mental state that results after performing the `move(b1,b2)` action will look like:

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

}
```

Only Completely Achieved Goals Are Removed Goals are removed from an agent's goal base *only if* they have been *completely* achieved. An achievement goal φ is only achieved when all of its sub-goals are achieved at one and the same time. For this reason, an agent should not drop a sub-goal that may have been achieved before the goal it is a part of has been achieved. If an agent would do that, it would become impossible for the agent to ensure that all sub-goals are achieved simultaneously. For those readers familiar with the problem, the Sussman Anomaly provides a famous example of the problems that an agent runs into if all it aims for is realizing the sub-goals of a larger goal one by one [51].

The fact that a goal is only removed when it has been achieved implements a so-called *blind commitment strategy* [47]. 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: a goal is only removed when it has been completely achieved. This default strategy can be adapted by the programmer for particular goals by using the built-in **drop** action.

4.2 Built-in Actions

We have seen that action specifications need to be provided for environment and user-defined actions above. GOAL provides several *built-in actions* that do not need to be specified before an agent can use them. Here we discuss all built-in actions except for the communication actions which are discussed in Chapter 7. These actions include actions for:

- printing text to a console: **print**,
- changing an agent's beliefs: **insert**, **delete**,
- changing an agent's goals: **adopt**, **drop**, and
- creating a log file: **log**.

Printing Text We start by discussing the simplest action: **print**. The **print** action has a single parameter that should be instantiated with a string (some text between double quotes `"..."`). **print** can always be performed and all it does is outputting a string to the console. For example,

```
print("Hello, world!")
```

outputs the text `Hello, world!` to the console.

Mental Actions There are four built-in actions that can be used to modify an agent's mental state. These actions are also called mental actions. We first discuss the actions for changing the agent's goal base. The **adopt** action:

```
adopt(φ)
```

is used for adopting a new goal. The goal φ must be a valid expression of the knowledge representation language that is used by the agent that can be inserted into a database of that language. For example, when using Prolog, φ must be a conjunction of one or more positive facts. The **adopt** action is enabled if the following preconditions hold:

- the agent does not believe that φ, i.e., we must have `not(bel(φ))`.
- the agent does not already have a goal φ, i.e., we must have `not(goal(φ))`.

The first precondition ensures that an agent does not adopt a goal that it believes has already been completely achieved. The second condition prevents the agent from (again) adopting a goal that it already is pursuing. Note that this condition also prevents an agent from adopting a goal that is a sub-goal of a goal that the agent already has. For example, an agent cannot adopt `on(b2,table)` as a goal if it already has a goal `on(b1,b2), on(b2,table)`. The effect of executing `adopt(φ)` is that φ is added as a single, new goal to the goal base of the agent.

The **drop** action:

```
drop(φ)
```

is used for dropping one or more goals of an agent. The goal φ must be a valid query in the knowledge representation language that is used. A **drop** action can always be performed. The effect of the action is that *any* goal present in the goal base of an agent that implies φ is removed from the goal base. More precisely, the condition `goal(φ)` is evaluated and each individual (complete) goal ψ in the goal base of the agent for which this condition succeeds is removed. As usual, the knowledge base in combination with an individual goal are used to evaluate `goal(φ)`. Suppose, for example, that an agent has the following knowledge and goal base:

```
knowledge{
   clear(X) :- not(on(_,X)).
}
goals{
   on(b1,b2), on(b2,table).
   on(b2,b1), on(b1, table).
   on(b1,b2), on(b2,b3), on(b3,table).
}
```

If the agent executes the action **drop**`(clear(b1))`, a goal in the goal base is removed if from that goal in combination with the knowledge base `clear(b1)` can be derived. In this example, the agent will remove the first goal `on(b1,b2), on(b2,table)` and the third goal `on(b1,b2), on(b2,b3), on(b3,table)`.

We now turn to actions for changing the belief base of an agent. The **insert** action:

```
insert(φ)
```

is used for inserting φ into the agent's belief base. The belief φ must be a valid expression in the knowledge representation language that can be used for updating a database. For example, when using Prolog, φ must be a conjunction of literals. An **insert** action can always be performed. The effects of executing **insert**(φ) are the same as applying a postcondition φ of an action. The **insert** action first removes all negative literals that occur in φ from the belief base and then adds all positive literals that occur in φ to the belief base of the agent. For example, **insert**`(`**not**`(on(b1,b2)), on(b1,table))` removes the fact `on(b1,b2)` from the belief base and adds the fact `on(b1,table)`. An **insert**(φ) action modifies the belief base of an agent such that φ follows from the agent's belief base after performing the **insert** action.

It is important to realize that just like other actions that modify the belief base of an agent it is possible that an **insert** action may also modify the goal base. The goal base will be updated if by performing an **insert** action the belief base is changed in such a way that the agent comes to believe that one of its goals has been completely achieved. In that case, the goal that the agent believes has now been realized is removed from the goal base.

The **delete** action:

```
delete(φ)
```

is used for removing φ from the agent's belief base. The **delete** action can be viewed as 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. In other words, **delete**`(p)` has the same effect as **insert**`(`**not**`(p))`.

Minimal Change
One thing to note about the **delete**(φ) action is that it removes *all* positive literals that occur in φ. The **delete** action thus *cannot* be used to ensure that the agent's beliefs are changed *minimally* such that the agent does not believe φ any more. Such an action would remove, for example, only `p` or `q` when the conjunction `p, q` should be deleted. Note that it is sufficient to only remove either `p` or `q` to make sure that the agent no longer believes `p, q`. The **delete** action discussed in the main text, however, removes both `p` and `q` and thus does not support minimal change.

Even though each **delete** action could be replaced in this way with an **insert** action, using **delete** for removing facts instead of **insert** enhances the readability of an agent program. It is best practice to only use **insert** for adding facts and to use **delete** for removing facts.

For example, instead of a single action **insert**(**not**(on(b1,b2)), on(b1,table)) which removes on(b1,b2)) and adds on(b1,table), the two actions **delete**(on(b1,b2)) and **insert**(on(b1,table)) can be used to obtain the same effect.

Combo Actions and the + Operator Actions can be combined into a new action using the **+** operator. Two or more actions that are combined by the **+** operator are called a *composed action* or *combo action*. We illustrate the use of this operator for combining **delete** and **insert** actions to create a new composed action that has the same effect as an **insert** action that "inserts" negative facts **not**(p), i.e., removes p. Using the **+** operator we can write, for example, the following composed action:

```
delete(on(b1,b2)) + insert(on(b1,table))
```

Actions combined by **+** are executed in the order that they appear. In the example, first the **delete** action and then the **insert** action will be executed. By first deleting and thereafter adding facts we simulate what would happen when **insert**(**not**(on(b1,b2)), on(b1,table)) is executed and we get the same end result.

The **+** operator can be used to combine as many actions as needed into a single, composed action (see also Section 4.2). It can be used to combine environment, user-defined, as well as built-in actions. Although there are no constraints on combining actions using **+**, it is not always a good idea to combine multiple environment actions into a single composed actions. This would be fine when environment actions are instantaneous but if environment actions are combined that take time to complete, unexpected behaviour can result (see also Chapter 6).

It is important to realize that the preconditions of actions in a combo action are also evaluated in order. All actions that are part of a combo action will be executed only if the precondition of each action holds when it is evaluated. For example, the **adopt** in the combo action:

```
insert(on(b1,table)) + adopt(on(b1,table))
```

will never be performed because it is one of the preconditions of **adopt** that the agent does not believe the goal that should be adopted. Clearly, in this case, after executing the **insert** action this precondition is violated. Moreover, if an action in a combo action is not enabled, i.e., its precondition fails, actions that appear later in the combo action are also not executed any more. For example, when after performing the initial **insert** action the precondition of the move action is evaluated and fails in the combo action:

```
insert(b1, table) + move(b1, table) + insert(just_performed_move)
```

both the move action and the **insert** action that follows the move action are not executed.

Logging Mental States to File It sometimes is useful to write the contents of part of an agent's mental state at certain points while the agent is running to a file. To this end, an agent can perform the **log**. The **log** has a single parameter that can be used to indicate which part of the mental state should be written to file: kb refers to the knowledge base, bb to the belief base, gb to the goal base, pb to the percept base, and mb to the message box of the agent. Any other argument is treated as a string and written as such to a file. For example,

```
log(bb)+log(just_performed_skip)
```

writes the contents of the current belief base of the agent and a string just_performed_skip to a file. By using **+** it is also possible to write multiple databases to file; e.g., **log**(bb)+ **log**(pb) will write the belief and percept base to a file. The file names used are a combination of the agent's name and a date-time-stamp. Note that it is only possible to write to a file if the MAS is executed with sufficient rights to do so. Also take into account that writing to a file may have significant impact on the performance of an agent.

Finally, like all other actions, built-in actions as well as composed actions combined by + must be closed when they are executed. That is, all variables must have been instantiated.

4.3 Notes

The action specifications discussed in this chapter are STRIPS-style specifications [40]. Action specifications in STRIPS consist of a pre- and post-condition which both are conjunctions of literals. The positive literals in the postcondition are part of the *add list* of facts that should be added to the beliefs of an agent after performing the action. The negative literals in the postcondition are part of the *delete list* of facts that should be removed after performing the action. STRIPS action specifications are used by many planners that automatically generate a plan, i.e., a sequence of actions, for achieving a goal state. One difference between action specifications in Goal and STRIPS is that we allow variables in the pre- and post-condition that do not occur in the parameter list of the action whereas this is not allowed in STRIPS. Another difference is that Goal allows multiple action specifications for a single action.

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 that its goal has been achieved. In [19, 47] 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 that it is no longer possible to achieve the goal. For example, an agent perhaps also should drop a goal to get to a meeting before 3PM if it misses the train and cannot get there in time any more. 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 instead provides the built-in **drop** action which allows an agent to remove goals whenever the agent thinks that is best.

4.4 Exercises

4.4.1

We revisit the "Hello World" agent that was developed in Chapter 1 in this exercise. The aim is to provide an action specification for the `printText` action that can be used to keep track of the number of lines that have been printed instead of the code in the **event** module that was used in Chapter 1. In other words, you should ensure that the agent's beliefs remain up to date by using another precondition and postcondition than **true** for the action `printText`.

- Start by removing the **event** module.
- Specify a new precondition for `printText` that retrieves the number of lines printed until now and add 1 to this number to compute the updated number of lines that will have been printed after performing the action.
- Specify a new postcondition that removes the old fact about the number of lines that were printed and that inserts the new updated fact.

Test your agent and verify that it is still operating correctly.

4.4.2

Consider the incomplete agent program for a Blocks World agent in Figure 4.1. 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 action specifications for the `pickup(X)` and `putdown(X,Y)` actions. Only use predicates that already occur in the agent program listing of Figure 4.1.

```
init module{
  knowledge{
    block(X) :- on(X,_).
    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]).
  }
  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).
  }
  actionspec{
     pickup(X){
        pre{                 }
        post{                }
     }
     putdown(X,Y){
        pre{                 }
        post{                }
     }
  }
}

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).
   }
}
```

Figure 4.1: Incomplete Blocks World Agent Program

Chapter 5

Agent Programs

In this chapter, we write a complete Blocks World agent program that is able to solve the Blocks World problems introduced in Chapter 3. We will use the mental state conditions for the Blocks World that we wrote in that chapter and the action specification for the `move` action discussed in Chapter 4 for writing our Blocks World agent program. Apart from the ingredients that we need and discussed in the previous two chapters, the main element that we still need to introduce is how an agent chooses the actions that it will perform. We will fill this gap here and explain how an agent decides what to do next. The focus of this chapter is on writing *strategies for selecting the right action* and on *the basic structure of an agent program*. The main language element introduced in this chapter are *action rules* for writing action selection strategies.

5.1 Programming with Mental States

An agent derives its choice of action from its knowledge, beliefs and goals. The knowledge, beliefs and goals of an agent make up its mental state. An 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 be viewed as *programming with mental states*. Because mental states and ways to change these states play such an important role in the decision making process of an agent, we briefly take stock of what we have seen so far in the previous two chapters before we introduce action rules that are used by an agent to make decisions.

In the Blocks World, the decision on what to do next concerns where to move a block, in a robotics domain it can be where to go to or whether to pick up something with a gripper or not. A decision to act typically depends on the beliefs about the current state of the agent's environment as well as on more general knowledge about the environment the agent acts in. In the Blocks World, for example, 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 the right decision. In Chapter 3 we discussed that the knowledge of an agent consists of conceptual and domain knowledge and is static whereas the beliefs are used to keep track of the current state of the agent's environment and change over time. In Chapter 4 we discussed how an agent's mental state can be updated to keep track of the current state of the environment.

A decision to act also depends on the goals of the agent. In the Blocks World, for example, an agent may decide to move a block on top of an existing tower of blocks if it is a goal of the agent to have that block sit on top of that tower. In a robotics domain, a robot may have a goal to bring a package somewhere and therefore picks it up. 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 we have seen in Chapter 4, goals are automatically removed when the agent comes to believe that the goal has been completely achieved. It is the only mechanism for automatically removing goals that has been built into Goal agents and we therefore said it is a *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 3 in order to verify whether the appropriate conditions for selecting an action are met.

When an agent has chosen an action to perform an action, an agent needs to actually *perform* the action. Part of performing an action, as we saw in Chapter 4, means changing the agent's mental state. An action to move a block, for example, will not only move the block but by applying the associated postcondition also updates the agent's beliefs about the current position of the block. Whether or not a block is also really moved in the agent's environment (simulated or not) depends on whether the agent is connected to this environment. Both the precondition and the effects of an action on the mental state of an agent need to be explicitly specified in an agent program by the programmer. There is no need to specify the actions that are part of the programming language itself.

5.2 On Deciding What to Do Next in the Blocks World

We continue the Blocks World example we used in previous chapters and will write an agent program that is able to effectively solve Blocks World problems here. This agent program will provide a simple example that allows us to explain how a basic agent decides what to do next.

In Chapter 3 we already introduced a simple language including the predicates `block`, `on`, `clear`, and `tower` for talking about the Blocks World and in Chapter 4 we specified the action `move` for changing a configuration of blocks. See Section 3.1.1 for an explanation of the Blocks World environment. The next step is to design and implement a *strategy* for choosing the right actions for moving blocks from an initial state to a goal state.

To be precise, what we are looking for is an *action selection strategy* that performs a sequence of moves that transforms an initial configuration of blocks in a Blocks World problem to a configuration of blocks that matches the goal state of the problem. We only need to make sure that each block sits of top of the right block and we do not need to worry about the exact position of a stack of blocks on the table. See Figure 3.1 for an example problem.

In addition to simply solving a Blocks World program, we want to write an agent that also performs reasonably. The performance of a Blocks World agent can be measured by means of the number of moves it makes to transform an initial state into a goal state. An agent performs *optimally* if it is not possible to improve on the number of moves that the agent makes to reach a goal state.[1] Although we are not after writing an agent program that performs optimally, we do not want our agent to make too many redundant moves either. We therefore briefly discuss some basic insights that will help us write a good agent program for solving a Blocks World problems.

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 3.1, all blocks except block `b3` and `b7` are misplaced. A first observation that we can make is that *only misplaced blocks have to be moved* in order to solve a Blocks World problem. Of course, if a block is not where we want it to be, that block also *needs* to be moved. We call the action of moving a block into position a *constructive move*. An important observation is that a constructive move always decreases the number of misplaced blocks.[2] Because only misplaced blocks need to be moved, it is best to make

[1]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 [31] for an approach to define such heuristics in GOAL.

[2]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*. 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 `b1` is a self-deadlock in Figure 3.1. The concept of self-deadlocks, also called singleton deadlocks, is important because on average nearly 40% of the blocks are self-deadlocks, see also [57].

```
init module{
   knowledge{
      block(X) :- on(X, _).
      clear(X) :- block(X), not( on(_, X) ).
      clear(table).
      tower([X]) :- on(X, table).
      tower([X, Y| T]) :- on(X, Y), tower([Y| T]).
   }
   beliefs{
      on(b1,b2). on(b2,b3). on(b3,table). on(b4,b5). on(b5,table). on(b6,b7). on(b7,table).
   }
   goals{
      on(b1,b5), on(b2,table), on(b3,table), on(b4,b3), on(b5,b2), on(b6,b4), on(b7,table).
   }
   actionspec{
      move(X, Y) {
         pre{ clear(X), clear(Y), on(X, Z), not( on(X, Y) ) }
         post{ not( on(X, Z) ), on(X, Y) }
      }
   }
}

main module [exit=nogoals] {
   program [order=linear] {
      if a-goal(tower([X, Y| T])), bel(tower([Y| T])) then move(X, Y).
      if a-goal(tower([X| T])) then move(X, table).
   }
}
```

Table 5.1: Agent Program for Solving the Blocks World Problem of Figure 3.1.

a constructive move whenever possible. We thus obtain the first rule that we want to implement as part of our strategy:

if a constructive move with block X *can be made, then move block* X in position.

Because we cannot always make a constructive move, we need another rule that tells us what to do in such a situation. In Figure 3.1, for example, we cannot make any constructive move. A third observation tells us not to move a block on top of another one in that case: A block that is put on top of another block but not moved into position needs to be moved again later. Because we assumed that there is always room on the table, it is better to move the block onto the table. That has the benefit that we can still move all other blocks that we could move before and are not preventing an agent from moving a block by putting another on top of it. We thus obtain our second rule:

if a block is misplaced, then move it to the table.

The condition of this rule will make sure that an agent only moves blocks to the table that need to be moved anyway. By first applying this rule until it is no longer applicable, an agent would first move all blocks (that are not already in position) to the table. By then applying our first rule, the agent would build the goal configuration. Note that a block in this case would never be moved more than twice. This means that an agent that implements these rules will never make more than $2 \times n$ moves where n is the number of blocks. By reversing the order of the rules and giving priority to the first rule, an agent might do even better.

5.3 Action Rules: Deciding Which Action to Perform Next

Action rules allow an agent to choose an action that it might perform next. In its most basic form, an action rule consists of a condition and the action that is selected. As cognitive agents derive their choice of action from their beliefs and goals, the condition of a rule is a mental state condition that allows an agent to inspect its mental state. The action of a rule can be any action including an environment, user-defined, built-in, or composed action (see Chapter 4). One important type of action rule that we will use here is of the form:

```
if <mental_state_condition> then <action>.
```

This rule basically tells an agent that if the mental state condition holds and the action can be performed, then the agent may consider performing the action. An action rule thus provides a *reason* for performing the action (by means of the mental state condition). We also say that an action rule is *applicable* if both the mental state condition of the rule and the precondition of the action hold. That a rule is applicable does not mean, however, that the action will be selected by the agent; there may be other applicable rules that the agent may prefer to apply instead.

Using the mental state conditions that we introduced in Chapter 3 and the `move` action specified in Chapter 4, it is rather straightforward to translate the informal rules of our strategy in the previous section into action rules. For convenience, we repeat these mental state conditions:

```
a-goal(tower([X| T]))                         % block X is misplaced
a-goal(tower([X, Y| T])), bel(tower([Y| T]))  % X can be moved constructively
```

The first condition expresses that block `X` is misplaced; we need this condition to implement the second rule of our strategy. The second condition expresses that a constructive move can be made with block `X`; this is exactly what we need to implement the first rule of our strategy. The two rules in the order of their priority are:

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

These rules correspond with lines 25-26 of the Blocks World agent program listed in Table 5.1. Together these rules implement our informal strategy for solving a Blocks World problem. Because these are the only rules available, the agent will only choose to perform a constructive move or to move a misplaced block to the table. (Check for yourself that the agent will never consider moving a block that is in position.) Also 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.[3] In other words, whenever the first rule is applicable and a constructive move `move(X,Y)` can be made, the second rule also is applicable and it also is an option to perform the action `move(X,table)`.

As we have seen in, for example, Figure 3.1, it is not always possible to perform a constructive move and in that case the first rule is not applicable. In that same example, however, it is possible to move several misplaced blocks to the table. To be precise, we can move block `b1`, `b4`, and `b6` to the table. We can check this by verifying that the instantiations **`a-goal`**`(tower([b1| T]))`, **`a-goal`**`(tower([b4| T]))`, and **`a-goal`**`(tower([b6| T]))` of the condition of the second rule hold and that the preconditions for the corresponding move actions hold. We also say that the rule generates three *action options*. Which of these options will the agent perform? The agent will simply select one *randomly*.

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

5.4 Rule Evaluation Order

Action rules are the main ingredients for specifying a strategy for choosing which action to perform next. An *action selection strategy* simply is a set of such rules. The rules that together define the action selection strategy of an agent are put into a **program** section in the agent's **main** module. See lines 23-28 of the Blocks World agent program listed in Table 5.1.

As we discussed above, whenever the rule for constructive moves is applicable the other rule is applicable as well. In general, more than one action rule may be applicable and each of these rules in turn may generate one or more action options. The *rule evaluation order* of a **program** section determines which rule will be applied.

Linear Order In line 24 of Table 5.1 the order has been explicitly specified to be **linear**. This means that the rules are evaluated *in linear order*, that is, from top to bottom as they appear in the **program** section. In linear order style rule evaluation, the first rule that is applicable is the only rule that is applied. That is, a **program** section with linear rule evaluation order is executed by evaluating the rules that appear in that section in order and by applying the first rule that is applicable; after applying the first rule, execution of the section is completed.

Changing the Rule Evaluation Order The order of rule evaluation can be changed by specifying an **order** option for the **program** section (see for an example that uses this option line 24 in Table 5.1). Four different styles of evaluating rules are supported: **linear**, **linearall**, **random**, and **randomall**.

Random Order The **random** rule evaluation order selects a rule at random until an applicable rule is found. When an applicable rule is found, that rule is applied. As a result, the agent will select an arbitrary action option. An agent that selects actions this way performs non-deterministically and may perform different actions in a different order each time it runs. Suppose, for example, that instead of **linear** order in the Blocks World agent program of Table 5.1 **random** order is used. Also suppose that the agent already has moved blocks `b1` and `b2` to the table starting from Figure 3.1. In that case, block `b4` can either be moved onto block `b3` and into position using the first rule and be moved to the table using the second rule. In random rule order evaluation either one of these actions can be performed by the agent, whereas with linear order only the constructive move can be chosen.

Linear All Order The **linearall** order evaluates *all* rules that appear in a **program** section in the order that they appear and applies *all* rules that are applicable. To be precise, when using this order, the first rule is evaluated and applied if applicable, then the second rule is evaluated and if applied if applicable, etc. Suppose again that the Blocks World agent has moved `b1` and `b2` already to the table. Choosing the **linearall** option in the Blocks World agent program then will result in the agent moving `b4` on top of block `b3` and thereafter moving `b6` to the table (and not moving it onto `b4` which would be a constructive move!).

Random All Order The **randomall** order is similar to the **linearall** order but evaluates rules in random order. To be precise, it evaluates *all* rules that appear in a **program** section once but in *random* order and applies a rule if it is found to be applicable.

Module	**Rule Evaluation Order**
init	**linearall**
event	**linearall**
main	**linear**
user-defined	**linear**

Table 5.2: Default Rule Evaluation Order for **program** Section

Default Order It is not necessary to specify the rule evaluation order of a **program** section. For example, in the agent program in Table 5.1 there is no need to do so because, by default, action rules that appear in the **program** section of a **main** module are evaluated in linear order. This is also true for **program** sections that occur in user-defined modules (see also Chapter **??**). The default order for **program** sections in the **init** and **event** modules, however, is **linearall**. Table 5.2 summarizes the default rule evaluation order for different types of modules.

5.5 Tracing the Blocks World Agent Program

As we observed 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 linear rule evaluation order for specifying an action selection strategy as it allows to prioritize rule evaluation. In the Blocks World example this means that the first rule for making a constructive move is preferred over the second rule for moving misplaced blocks to the table and will be applied whenever possible. In this example, this also means that this style of rule evaluation will perform better than any of the other styles.

As we also observed, however, even if the first rule is not preferred over the second, the agent will never perform more than $2 \times n$ moves, with n being the number of blocks. It thus is interesting to find out how much better the linear style actually performs than the random style. By tracing the behaviour of the Blocks World agent program that uses the **random** option we will also gain a better understanding of the execution of an agent program.

We will trace one particular run of the Blocks World agent program of Table 5.1 in more detail here, where we assume that the **linear** order has been replaced by **random**. We also assume that the **init** module has already been executed and the mental state of the agent has been initialized.

In the initial state, depicted also in Figure 3.1, the agent then can move one of three blocks `b1`, `b4`, and `b6` that are clear to the table. In order to verify that moving the blocks `b1`, `b4`, and `b6` to the table are options, we need to verify the mental state conditions of action rules that may generate these actions. We will do so for block `b1` here; the other cases are similar. Only the second rule in the **program** section of Table 5.1 is applicable. It is also clear that this is the only rule that can be instantiated such that the action of the rule matches with `move(b1,table)`. The mental state condition of the second rule expresses that a block `X` is misplaced. As block `b1` clearly is misplaced, this rule is applicable. The reader is invited to verify this by checking that **`a-goal`**`([b1,b5,b2])` holds in the initial mental state of the agent.

Next we need to verify that the precondition of the `move` action for moving the blocks `b1`, `b4`, and `b6` holds by instantiating the precondition of the `move` action and by inspecting the knowledge and belief base of the agent. For example, instantiating `move(X,Y)` with block `b1` for variable `X` and `table` for variable `Y` gives the corresponding precondition `clear(b1), clear(table), on(b1,Z),` **`not`**`(on(b1,table))`. By inspection of the knowledge and belief bases, it immediately follows that `clear(table)`, and we find that `on(a,Z)` can be derived by instantiating variable `Z` with `b2`. Using the rule for `clear` it also follows that `clear(b1)` and we conclude that action `move(b1,table)` is enabled as we do not have `on(b1,table)` yet. Similar reasoning shows that the actions `move(b4,table)` and `move(b6,table)` are enabled as well. The reader is invited to check that no other actions are enabled.

Assuming that `move(b1,table)` is selected from the available options, the action is executed by updating the belief base by applying the instantiated postcondition **`not`**`(on(b1,b2)), on(b1,table)`. This means that the fact `on(b1,b2)` is removed from the belief base and `on(b1,table)` is added. The goal base may need to be updated also when one of the goals has been completely achieved but this is not the case here.

As all blocks except blocks `b3` and `b7` are misplaced, similar reasoning would result in a possible trace where consecutively `move(b2,table)`, `move(b4,table)`, and `move(b6,table)` are executed. Observe that performing `move(b4,table)` only is possible because we assumed **random** order. After performing these actions, all blocks are on the table, and the first rule of the program can be applied to build the goal configuration. One possible continuation, e.g., is to execute `move(b5,b2)`, `move(b1,b5)`, `move(b4,b3)`, and `move(b6,b4)`. In this particular trace, the goal state would be reached after performing 8 actions.

How does this compare to other traces? Starting in the initial configuration of Figure 3.1, there are 3 shortest traces - each including 6 actions - that can be generated by the Blocks World agent to reach the goal state. Each of these traces starts by first performing `move(b1,table)`, `move(b2,table)`, `move(b4,b3)`, and then continues:

Trace 1:	`move(b6,b4)`, `move(b5,b2)`, `move(b1,b5)`.
Trace 2:	`move(b5,b2)`, `move(b6,b4)`, `move(b1,b5)`.
Trace 3:	`move(b5,b2)`, `move(b1,b5)`, `move(b6,b4)`.

Table 5.3: Three Optimal Traces

There are many more possible traces, e.g., by starting with moving block `b6` to the table, all of which consist of more than 6 actions.

To conclude the discussion of the example Blocks World agent, we return to our initial question how good or bad the **random** rule evaluation order would perform. In Figure 5.1, the line labelled RSG (for "Random Select GOAL ") shows the average performance of the agent of Table 5.1 that uses **random** rule evaluation order. This agent is compared with one that always first moves all blocks to the table (simply reverse the order of the rules in the agent program) and only then starts building towers. The performance of this agent is plotted by the line labelled US (for "nstack Strategy"). Performance is shown in terms of the number of moves relative to the number of blocks present in the Blocks World problem. As can be seen, the performance of the agent that randomly evaluates rules is somewhat better than the performance of the agent that first moves all blocks to the table. This is what is to be expected as the agent that randomly evaluates rules may sometimes perform constructive moves instead of moving a block first to the table and then moving it into position.

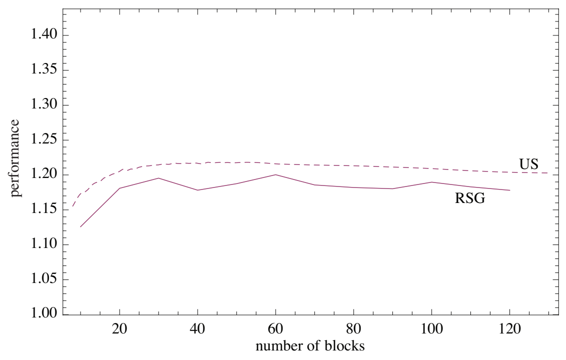


Figure 5.1: Random Rule Evaluation (RSG) Compared to an Unstack Strategy (US)

5.6 The Structure of an Agent Program

An agent program *is a set of modules.* Table 5.1 provides one example of an agent program. The agent program consists of two modules: an **init** module and a **main** module. These modules are two of the three modules that are built into the programming language. In Chapter 6, the third module called **event** module will be introduced. The only requirement that an agent program must satisfy is that it contains either a **main** or an **event** module. Chapter **??** will also introduce user-defined modules.

Sections of a Module
A module can have the following 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*, typically used to represent an initial state of affairs. These are added to the *belief base* of the agent.

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

program: a set of action rules that define a strategy or policy for action selection.

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

Module A module in turn consists of different sections. These sections include a **knowledge** section for specifying an agent's knowledge, a **beliefs** section for keeping track of the current environment state, a **goals** section to specify what an agent wants, a **program** section for specifying an action selection strategy, and an **actionspec** section for specifying actions. Most of these sections are *optional* and do not need to be present. The absence of a **knowledge** section in a module, for example, simply means that no knowledge has been provided to fill the knowledge base of the agent. Knowledge may already have been or still is to be provided by another module.

A module is executed by first handling all sections other than the **program** section. These sections are used to initialize or update the agent's mental state and to declare actions that the agent can perform. Thereafter the action rules in the **program** section of the module are applied.

The init module When an agent is first created and launched, its **init** module is executed first. The main function of the **init** module is to initialize the mental state of the agent and to declare the actions that an agent can perform. The **program** section of an **init** module may also be used to handle percepts that are send only once by an environment. After initializing an agent by means of the **init** module, that module is never (automatically) executed again.

If an **init** module is absent or does not contain a **knowledge**, **beliefs**, or **goals** section this simply means that the initial knowledge, belief and goal bases of the agent will be empty. The absence of an action specification section means that the agent cannot perform any other actions than those provided as built-in actions by GOAL, or can only provide the actions that are specified in the module that is being executed at that moment.

The Main Decision Cycle After initializing the agent by means of an **init** module, the agent enters its main decision cycle that it will continue to execute until it the agent is terminated. Each decision cycle, first, the **event** module is entered and executed. The action rules in an **event** module are also called *event* or *percept rules*. These rules can be used for specifying how percepts received from the environment and other events should be used to update the agent's mental state. The function of the **event** module is to handle events that are received from an environment (e.g., percepts) or other agents (e.g., messages). The idea is to make sure that the state of an agent is updated just before it will decide which action to perform next.

After executing the **event** module and updating the agent's mental state, in each decision cycle, second, the **main** module is entered and executed. The main purpose of the **main** module is to execute the strategy of the agent for selecting actions in order to achieve the agent's goals.

Executing an Agent Program As an agent program is a set of modules, executing an agent program means to execute the modules that make up the agent. When an agent program is launched, the agent program is used to create an agent with an initially empty mental state. After executing the **init** module, other modules then are executed while executing the main decision cycle discussed above. This decision cycle is continuously executed and generates a run of the agent program.

5.7 Notes

The GOAL programming language was first introduced in [35]. Previous work on the programming language 3APL [34] had made clear that the concept of a declarative goal, typically associated with cognitive agents, was not supported adequately in that language. Declarative goals at that time were part of the "missing link" needed to bridge the gap between agent programming languages and agent logics [19, 47, 38]. The programming language GOAL was introduced as a step towards bridging this gap. The design of GOAL has been influenced by the abstract language UNITY [18]. A temporal verification framework for GOAL was provided as a formal specification framework for the agent programming language. In [30], it has been shown that GOAL agents instantiate Intention Logic [19], 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 [10]. The *Tropism System Cognitive Architecture* designed for robot control also uses a random generator in its action selection mechanism [8]. 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 means of a biased roulette wheel. Each option is allocated a space (slot) on the wheel, proportional to the associated tropism value. Subsequently, a random selection is made on the roulette wheel, determining the robot's action.

A set of action rules can be viewed as a *policy*. There are two differences with standard definitions of a policy in the planning literature, however [26]. First, action rules do not always 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.[4]

5.8 Exercises

5.8.1

Consider again the program of Figure 4.1. By default, the rules are evaluated in linear order. Suppose we change this to **random** order by adding `[order=random]` to the **program** section. After this change, because of the third action rule, an agent can now also put a block that it is holding down on the table even if the block could also be put in position. Change the rule to ensure that a block is only put down on the table if it cannot be put in position directly. (Although it is clear that the linear rule evaluation order is useful to prevent this problem, the aim of this exercise it to show that in principle we can do without this option by specifying the right rule conditions.)

[4] "Universal" in the sense of [52], where a *universal plan* (or policy) specifies the appropriate action for *every* possible state.

Chapter 6

Environments: Actions & Sensing

We connected the agent that we wrote in Chapter 5 to an environment called the Blocks World. This environment is rather simple. Blocks in this environment, for example, are moved instantaneously and moving blocks with the gripper does not take any time. By making some additional simplifying assumptions, moreover, we could focus all of our attention on designing a strategy for selecting `move` actions. For example, we assumed that the agent controlling the gripper is the only agent that is present and that only that agent can move blocks. Because of this assumption we did not need to think about changes made to the block configuration by, for example, another agent. In other words, we assumed that the agent controlling the gripper has *full control*.

Environments in which changes can happen that are not controlled by the agent 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 5 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 that we wrote in Chapter 5. The presence of other agents is one reason why agents need to be able to perceive their environment.

We are also going to look at a slightly more complicated version of the Blocks World where another agent (you!) may interfere with the actions of the agent that is controlling the gripper. In this version of the Blocks World, dynamic events can occur and the agent has to adapt its behaviour in response to such events. We call this dynamic version of the Blocks World the *Tower World*. One important difference between the Blocks World and the Tower World is that several actions are available for controlling the gripper and that each of these actions *take time*. Such actions are also called *durative actions*. As we want our agent to act in real time, this introduces some additional complexity for writing an agent that can achieve its goals in the Tower World.

6.1 Environments, Entities, 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* that is made available by an environment. Each environment makes available different entities. An entity can be any object that is present in an environment and that can perceive and act to change that environment. In the Blocks World, for example, an agent can be connected to a gripper. The gripper allows the agent to "see" the blocks that are present in the environment and to move blocks. The gripper thus acts as the sensors and actuators of the agent. In a gaming environment such as Unreal Tournament, for example, an agent can be connected to a bot. The bot allows the agent to see weapons and other bots, for example, and to move the bot around in the game. An agent may also be connected to physical entities such as a robot in the real world. The agent in all of these cases can be viewed as the *mind* of the entity and the entity itself is best viewed as the *body* of the agent. By connecting to an entity an agent gets control over the capabilities of the entity and is able to perceive what the entity is able to see in the environment.

```
environment{
   env = "blocksworld/blocksworld.jar".

   init = [start = "bwconfigEx1.txt"].
}

agentfiles{
   "stackBuilder.goal".
}

launchpolicy{
   when entity@env do launch stackbuilder : stackBuilder.
}
```

Table 6.1: A MAS file that connects an agent to the gripper in the Blocks World

An agent connects to an entity in an environment by means of an *environment interface*. Goal supports and requires the Environment Interface Standard (EIS) to connect to an environment. EIS provides a toolbox for engineering environment interfaces that has been developed to facilitate the connection and interaction of agent platforms such as Goal with environments [5, 6]. A connection of an agent with an entity in an environment is established by means of a MAS file. A MAS file is a recipe for launching a multi-agent system including an environment and consists of three sections. See Table 6.1 for an example MAS file for the Blocks World. The first section is the **environment** section. This section is optional; agents can also be run without connecting them to an environment. In the **environment** section a reference to a jar file must be provided that enables Goal to load the interface to an environment. 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. Look for additional documentation that is distributed with an environment for information on the parameters that can be set for that environment. Initialization parameters are specified as a list of key-value pars between square brackets. In Table 6.1, for example, the initial configuration of blocks is loaded from a text file by specifying the name of a file between double quotes as value for the key named `start`.

Agents are constructed from Goal agent files that must be listed in an **agentfiles** section. The files mentioned in this section are used for creating agents and all agent files that are needed should be listed here by including the file name (or a path to the file) between double quotes ended with a dot. 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 a launch rule is provided in Table 6.1. This rule is of the form

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

This rule is a *conditional* launch rule that is triggered whenever an entity is made available by the environment (indicated by the condition **entity@env**). If the agent platform is notified that an entity is available, this rule is triggered. In that case, an agent is with name `agentName` is created using the referenced agent file `agentFilename` and the agent is launched and connected to the entity. The rule in Table 6.1, for example, creates an agent called `stackbuilder` using the agent file `stackbuilder.goal` and connects it to the gripper in the Blocks World. Examples of other types of launch rules are provided in Chapter 7.

We say that agents are connected to an environment and to an entity rather than that they *are part* of an environment. We use this terminology to emphasize the *interface* that is used to connect the agent to an entity in its environment. This interface can be viewed as a channel between the agent and its environment. It is used by the agent to request the environment to provide a *service*. Or, to phrase it somewhat differently, the agent uses the interface for sending commands to the entity it is connected to in order to make the entity perform *actions* in the environment. It is also used by the agent to request the environment to provide *information* about the current state of the environment. Put differently, the agent uses the interface for receiving *percepts* from the environment. Percepts allow an agent to observe (part of) its environment.

We cannot emphasize enough how important it is while writing 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. An agent program must use these percepts and actions and therefore can only be written after careful analysis of the percepts and actions the interface provides. As we will discuss below, it is therefore 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. A MAS file therefore does not need to have an **environment** section. An agent that is not connected to an environment may perform useful computations like any other type of program. The main difference with other types of programs, as noted in Chapter 5, is that agent programming is *programming with mental states*. In a multi-agent system, moreover, not all agents need to be connected to an environment. Agents that are not connected to an environment and that are part of a MAS in which some agents are connected to an environment can be used, for example, to coordinate and manage those agents. By communicating with agents that are connected to an environment such agents can, moreover, also obtain information about the state of the environment. An agent, for example, can 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 on the environment can be used, for example, to instruct other agents what to do.

6.2 The Blocks World with Two Agents

It is possible to connect more than one agent to a single entity and we will exploit this possibility to introduce the need for sensing an environment. We will connect two agents to the gripper in the Blocks World. The two agents that we connect to the gripper are the Blocks World agent `stackBuilder` of Chapter 5 and a new agent that we introduce here called `tableAgent`.

```
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 6.1: A MAS file that connects two agents to the gripper in the Blocks World

In order to connect two (or more) agents to an entity we need to add a new agent file that we call `tableAgent.goal` to the **agentfiles** section; see line 5 in Figure 6.1. We also need to change the launch rule of the MAS file in Table 6.1. There is no need to change the **environment** section; we still want to connect to the Blocks World environment. The change we need to make in order to connect two agents to the gripper requires a simple modification of the launch rule: We add the `tableAgent` to the list of agents that need to be connected to the gripper as in line 9 of Figure 6.1.

```
 1 init module {
 2    knowledge{
 3       block(X) :- on(X, _).
 4       clear(X) :- block(X), not( on(_, X) ).
 5       clear(table).
 6 
 7       % allClear succeeds if all blocks are clear
 8       % note that allClear also succeeds if there are no blocks
 9       allClear :- forall(block(X), clear(X)).
10    }
11    program{
12       % first insert all percepts into the agent's belief base
13       forall bel( percept( on(X, Y) ) ) do insert( on(X, Y) ).
14       % then adopt a goal to clear all blocks (in this order, to make sure that the agent
15       % does not believe that allClear has been achieved just because there are no blocks)
16       if true then adopt(allClear).
17    }
18    actionspec{
19       move(X,Y) {
20          pre{ clear(X), clear(Y), on(X,Z), not(on(X,Y)) }
21          post{ not(on(X,Z)), on(X,Y) }
22       }
23       % @int declares action as internal and prevents that it is send to the environment
24       skip@int {
25          pre{ true }
26          post{ true }
27       }
28    }
29 }
30 
31 main module[exit=nogoals]{
32   % randomly select to move a block to the table (if possible) or to perform skip.
33    program[order=random]{
34       if bel( on(X,Y) ) then move(X,table).
35       if true then skip.
36    }
37 }
38 
39 event module{
40    program{
41       forall bel( percept( on(X, Y) ), not( on(X, Y) ) ) do insert( on(X, Y) ).
42       forall bel( on(X, Y), not( percept( on(X, Y) ) ) ) do delete( on(X, Y) ).
43    }
44 }
```

Figure 6.2: Agent That Moves Blocks to the Table

The code of the `tableAgent` is listed in Figure 6.2. The main goal of this agent, different from the `stackBuilder` agent, is to put all blocks on the table. The agent program contains only some of the knowledge of the `stackBuilder` agent (it does not need to know what a tower is) and it has no initial beliefs. In the **program** section of the **init** module (lines 11-17 in Figure 6.2) the agent first processes the initial percepts it receives and thereafter adopts a goal `allClear` to clear all blocks. The **knowledge** section contains a rule that specifies when this goal is achieved. Because an agent can only adopt a goal that it not believes to have been achieved already, before adopting the goal we first need to make sure that the agent has some beliefs about blocks (otherwise the goal would be vacuously true). The agent can perform `move` actions like the `stackbuilder` agent but its **actionspec** section also contains another `skip` action. This action is declared as internal action by adding **@int** after its name; this prevents that the action is sent to the agent's environment. The `skip` action allows the agent to do "nothing": It is always enabled and changes nothing.[1].

The **main** module randomly selects one of two rules. The first rule (line 34) moves a block that does not already sit on the table to the table. The second rule (line 35) performs the `skip` action. Recall that the **`order`**`=random` option (line 33) implies that the rules are evaluated in random order; if a block can be moved to the table, the agent therefore will with equal probability either move a block to the table or perform the skip action. The `tableAgent` also has a so-called **event** module. This module is used for handling events such as percepts and will be explained in more detail shortly.

Because both agents are connected by the MAS file to the gripper in the Blocks World environment, both agents are able to move blocks. 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 writing our first Blocks World agent we implicitly made this assumption. We also assumed that 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 initial beliefs of the agent and provide an action specification that 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. 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! Of course the agent still needs to do something to maintain a correct representation of its environment: it must update its belief base accordingly. Also note that in this case the Blocks World environment is little more than a graphical display of the configuration of blocks in this world.

All this changes when full control is lost. If another agent can 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 performed itself. 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.

6.2.1 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 its environment. This interface specifies which percepts an agent will receive from the environment. The percept interface is part of the environment interface for connecting agents to an environment [6, 5]. In general, an agent does not need to actively perform sense actions. An exception to this rule is when the environment makes special actions for observing the environment available to an agent (the UNREAL TOURNAMENT environment provides such actions). The approach for handling percepts of a GOAL agent is that each time just before the agent decides which action it will perform, the agent processes the *percepts* it receives. The old percepts are automatically removed and the new ones inserted into the *percept base* of the agent. The steps that are performed if an agent is connected to an environment are:[2]

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,

[1]Below we will see 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 a major difference between the two actions, however: the `skip` action is not an action that is made available by 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.

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

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 sits 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. Together, all percepts received from the Blocks World provide an accurate and complete picture of 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, the Blocks World provides a percept for each block and thus provides information about the complete state.

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 agent. A special keyword `percept` 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 `b1` sits on top of block `b2` is inserted into the percept base as the fact `percept(on(b1,b2))`.

Percepts represent "raw data" received from the environment that the agent is operating in. For several reasons an agent should not simply add new information that it receives to its belief base.[3] One reason is that the received information may be inconsistent with the information that is currently stored in the belief base of the agent. For example, a perceptual fact `percept(on(b1,table))` that indicates that block `b1` sits on the table may conflict with a fact `on(b1,b2)` stored in the belief base that says that block `b1` sits on top of block `b2`. In order to correctly update the belief base, the fact `on(b1,b2)` needs to be removed and the fact `on(b1,table)` needs to be added to the belief base. A more pragmatic reason for being careful about inserting new information is that perceptual facts that are received may represent signals that need further 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 (which an agent may receive in the Wumpus World environment) 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.

In order to process the percepts received from the environment, the **event** module of an agent is executed (step 4 of the cycle above). The processing of percepts is done 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 for the `tableAgent` listed in Figure 6.2 contains an **event** module with two event rules in its **program** section (lines 39-43).

Event rules are just action rules that are named differently because their main function is to handle events. An event rule thus has the same form as any other rule. Most often, out of three types of rules listed below, the second is used for processing percepts:

```
if <mental_state_condition> then <action>.

forall <mental_state_condition> do <action>.

listall <Listvar> <- <mental_state_condition> do <action>.
```

[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 can be used to directly modify the belief base of the agent.

The reason that the **forall** rule is used most for processing percepts 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)` is generated. An agent may, for example, receive the perceptual facts `percept(on(b1,b2))`, `percept(on(b2,b3))`, and `percept(on(b3,table))`. In order to process all of these percepts and insert these facts into the belief base, it should be able to perform an update for each of these three percepts. Rules of the form **`if...then...`** cannot be used for this purpose because they are only fired for a single possible instantiation. In contrast, rules of the form:

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

perform action `<action>` for each instantiation of `<mental_state_condition>` for which the mental state condition succeeds.

This explains why these rules are used in the **event** module of the `tableAgent`. Let's assume, for example, that the agent believes that `on(b1,b3)`, `on(b3,b2)`, and `on(b2,table)`. Upon receiving `percept(on(b1,b2))`, `percept(on(b2,b3))`, and `percept(on(b3,table))`, both rules in the **event** module of the `tableAgent` will then be applied three times. As a result, the facts `on(b1,b2)`, `on(b2,b3)`, and `on(b3,table)` will be inserted into the agent's belief base (by the first percept rule) and the facts `on(b1,b3)`, `on(b3,b2)`, and `on(b2,table)` that it initially believed will be removed from the agent's belief base (by the second percept rule). This is what we want for the Blocks World and why we need to use the rules of the form **`forall...do...`** 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 default behaviour of the **main** module and of the **event** module that are relevant. 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 the 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 6.2 for two examples (line 41-42). Rules that use the special keyword `percept` are 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 the **event** module is that, by default, *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 potentially useful information for making the right choice of action. This type of rule evaluation ensures that the new information that is received about the position 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).

6.2.2 Multiple Agents Acting in the Blocks World

We argued above that when both the `stackBuilder` as well as 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 have an **event** module for handling percepts. For this purpose, we can add exactly the same **event** module as that of the `tableAgent` to the `stackBuilder` agent.

```
init module {
   knowledge{
      block(X) :- on(X, _).
      clear(X) :- block(X), not( on(_, X) ).
      clear(table).
      tower([X]) :- on(X, table).
      tower([X, Y| T]) :- on(X, Y), tower([Y| T]).
   }
   goals{
      % a single goal to achieve a particular Blocks World configuration.
      % assumes that these blocks are available in the Blocks World.
      on(b1,b5), on(b2,table), on(b3,table), on(b4,b3), on(b5,b2), on(b6,b4), on(b7,table).
   }
   actionspec{
      move(X, Y) {
         pre{ clear(X), clear(Y), on(X, Z), not( on(X, Y) ) }
         post{ not( on(X, Z) ), on(X, Y) }
      }
   }
}

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

Figure 6.3: Agent Program for Solving Blocks World Problems

There is a second change we need to make to the `stackBuilder` 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 blocks configuration and there is no need any more to insert the initial configuration into the belief base of the agent. In general, we also do not know what the initial configuration is and even if we would know this, an initial belief base could immediately be invalidated because, for example, the other agent is the first to modify the Blocks World. By adding the **event** module and by removing the belief base we also make the agent more generic as it will be able to handle more blocks configurations. For completeness, we list the code of the resulting `stackBuilder` agent in Figure 6.3.

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 *instantaneous*. Because the `move` action in the Blocks World is instantaneous, each time that 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 time, however, the `tableAgent` may also perform actions that interfere with the goal by moving blocks to the table. A tower that is being built by the `stackBuilder` agent then may be broken down again by the `tableAgent`!

Will either of the agents 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? If we assume that the agents are taking turns when performing actions, it is relevant to note that the `tableAgent` randomly performs `move` and `skip` actions. The `skip` action does not change the blocks configuration and each time the `tableAgent` does nothing the `stackBuilder` may further its own goal without being hindered by the `tableAgent`. It is less clear what happens when both agents run in parallel (as GOAL agents do). Answering the question whether either of the agents will be able to achieve their goal is left as an exercise.

Exercise 6.2.1

1. Will the `stackBuilder` agent of Figure 6.3 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 in the GOAL platform. Do the agents always perform the same number of actions?

3. If one of the agents achieves its goal, that agent will terminate (note the **`exit`**`=nogoals` option). Now suppose that the `tableAgent` will always achieve its goal. Will the `stackBuilder` agent then also be able to achieve its goal? What if we remove the **exit** option from the `tableAgent`. Run the agents after removing this option from the `tableAgent` and check whether the `stackBuilder` agent will be able to achieve its goal.

4. Run the multi-agent system with the two agents for the Blocks World again but now distribute the agents physically over different machines. Do the agents perform the same number of actions?

6.3 The Tower World

The Tower World (see Figure 6.4) 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.

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 that we discussed above. In the Tower World, however, there are other reasons why being able to sense the environment is important. One of these is that the completion of actions takes time.

6.3.1 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 any more there is a need to be able to

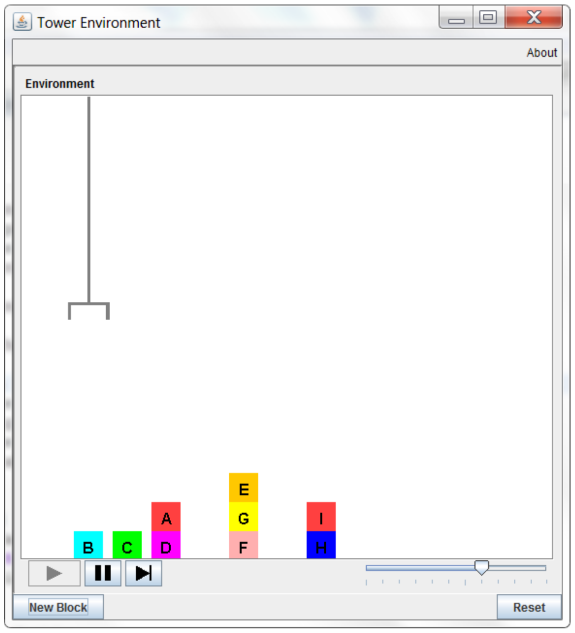


Figure 6.4: The Tower World

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 make sense. In such cases the agent should *reconsider* its actions and possibly also its goals.

An agent is able to monitor the progress of actions because it does not block on an action that it sends to the environment (that is, at least, if the environment itself does not prevent the agent from doing something). After sending an action to the environment, the agent can do other things and leave completion of the action that is being performed to its body (the entity it is connected to such as the gripper in the Tower World). An important thing to do is to to monitor the progress of the action by perceiving the environment.

Postconditions 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 another problem that we need to address: It at least appears that we cannot provide adequate postconditions for such actions. For several reasons we cannot specify, for example, that the postcondition of a durative action for picking up a block after performing that action implies that the gripper holds that block. One problem is that postconditions of actions are used immediately upon sending the action to the environment by an agent to update its mental state (how would an agent know that an action has completed?). If we would specify a non-empty postcondition for a durative action, that means that the agent would *immediately*, for example, 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. Another problem is that a durative action might fail while it is being performed. It takes time to pick up a block and only if the action is successful will the effect of holding the block be realized.

How do we solve this problem? The solution that we provide here 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.

Preconditions We are able to provide adequate *preconditions* for durative 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 makes three actions available instead of the single `move` action that is made available in the Blocks World. The three actions are:

- an action `pickup(X)` for picking up a block with the gripper,
- an action `putdown(X,Y)` for putting down one block on another block, and
- a special `nil` action that puts the gripper back in the left-upper most corner of the graphical user interface of the environment.

In order to provide adequate action specifications, it is important to know what kind of percepts are provided to agents when they are connected to an environment. In the Tower World, an agent receives the following three kinds of percepts:

- a percept `block(X)` that informs the agent that `X` is a block,
- a percept `on(X,Y)` that informs the agent that `X` sits on top of `Y`, and
- a percept `holding(X)` that informs the agent that the gripper holds `X`.

The three predicates `block/1`, `on/2`, and `holding/1` can be used to specify the preconditions of the actions.

There are two conditions that need to be satisfied in order to be able to perform a `pickup` action successfully: the block that needs to be picked up needs to be clear, and the gripper cannot already be holding a block. Using the definition of the `clear/1` predicate that we introduced in Chapter 3 (see also line 4 in Figure 6.3) 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 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 precondition for the `putdown` action is that the gripper holds the object that is being put down and that the object that the first object is put onto is clear. **`true`** is used to fill the postcondition again.

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

The precondition for the nil action is simply **true**: it can always be performed. As before, we also use **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 moves the gripper to the left-upper corner position in the graphical interface provided with the Tower World environment (see Figure 6.4). As such, the action is of little interest with respect to our main goal: creating a configuration of blocks. Also note that the predicates in terms of which percepts are provided by the environment to the agent do not say anything about the exact position of the gripper. The idea simply is to use this action when an agent has nothing to do any more to achieve its goals. We then put the gripper back into its initial starting position as a signal that the agent has finished.

6.3.2 Percepts in the Tower World

The percepts that an agent receives from the Tower World environment are exactly the same predicates block and on that we have seen before in the Blocks World environment. An agent receives one additional predicate holding in the the Tower World. The holding predicate has one parameter and indicates which block the gripper is holding, if any. In the Tower World, if the percept is absent, we should conclude that the gripper is not holding anything.

As in the Blocks World, we may also assume *full observability* in the Tower World: Every time percepts are received they provide *correct and complete* information about the state of the environment, 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 derive that the gripper is not holding any block. As a result, we can reuse the exact same percept rules for the Blocks World introduced above and we only need to add a pair of rules for the predicate holding that follows the exact same pattern we have seen 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) ).
  }
}
```

6.4 Performing Durative Actions in Environments

We have seen in Section 6.3.1 how to specify durative actions. We will discuss some other aspects related to performing durative actions in this section. One question we address here is what happens when a durative action has been initiated and still is in progress when the agent requests the environment (or, the entity it is connected to) to perform another action.

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 it is no longer feasible to perform an instance of the action `putdown(X,Y)` any more 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 and initiate another, feasible action instead. In this particular example, it would make sense to put block `X` on the table instead by means of performing the action `putdown(X,table)`. In the Tower World, if an action is in progress and an agent performs another action, this action *overwrites* the first action and action immediately continues by 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 `putdown(X,Y)` where `Y≠table` with the action `putdown(X,table)` provides an example: The gripper will immediately continue with putting down `X` on the table. 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 comes to a halt. There is simply no way to comply with such a quickly executed set of instructions for the bot. Depending on the environment, you should even keep in mind that if an agent performs *exactly the same action* again and again the result may not be what you expect.

It should be noted, moreover, that there are other options than overwriting a durative action that is in progress. An important option that may make sense for particular combinations of actions is to perform them *in parallel*. There are many examples where this option is more reasonable than overwriting. In UNREAL TOURNAMENT, for example, a bot 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. You probably also expect a gripper to maintain grip on a block it holds while moving. Perhaps less sensible but nevertheless an option is that an environment will be *queuing* actions that are being sent by the agent, executing these in order. Another option is that an environment simply *ignores* new actions that are requested by an agent while another action is still in progress.

6.5 Action Selection and Durative Actions

We have gained more insight in how durative actions are handled by entities in an environment but we have not discussed yet how an agent should deal with durative actions when it is deciding which action to perform next. Because durative actions may overwrite each other, it becomes important that an agent does not issue a new request to perform an action or, depending on the environment, continues to perform the same action as long as it has not been completed yet. Otherwise an agent might come to a halt similar to the bot that is continuously instructed to move in different directions. At the same time, we also need to take into account that an agent may have to *reconsider* its choice of action if it is no longer feasible.

All actions in the Tower World overwrite each other because a gripper can do at most one thing at the same time. Even though still simple this environment therefore provides a good example of the issues we want to discuss here. 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.

6.5.1 Action Selection and Focus

The issue with underspecification of action rules mentioned above can be illustrated by looking again at the action rules that were used in the Blocks World agent of Chapter 5 (see Table 5.2). We repeat the **main** module of that agent here with only a minor modification: The macro `misplaced` is replaced by a macro that is also able to handle obstructing blocks (i.e., blocks that are not part of the goal state but must be moved to gain access to blocks that are needed to achieve that state).

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

    if constructiveMove(X, Y) then move(X, Y).
    if obstructingBlock(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 for a moment assume that an agent can perform `move` actions but that they would take time, i.e., we assume for a moment that `move` is *durative*.

It then becomes important to realize again that the rules underspecify the selection of an action even though the default order of evaluating rules in a **program** section of a **main** module is linear. The default linear order evaluation of rules implies for the two rules above that if a constructive move can be made it will be made. That is, the first rule will be applied whenever possible. However, if it is possible to make more than one constructive move, this rule does not specify *which* of these constructive moves should be selected. For example, assume that an agent has the beliefs `on(a,table)`, `on(b,table)`, `on(c,table)`, and `on(d,table)` and has a goal `on(a,b), on(b,table), on(c,d), on(d,table)`.[4] Using the rules above this agent could perform two different constructive moves: `move(a,b)` and `move(c,d)`. An agent using these rules would select either one of these choices at random and would be completely indifferent which of these moves is selected. In the Blocks World this is fine: Any constructive move that can be performed instantaneously is as good as any other. Not having to specify which of the two should be selected is a benefit here and it would have little value to try to force the agent to make a choice (the option to leave choices like these open is one of the distinguishing features of writing agent programs in GOAL).

Because durative actions may overwrite each other, if the `move` action would happen to be durative, however, we do need to change the agent so that it will make a choice. The point is that if an agent would continuously select a different constructive move, the agent would come to a halt and do nothing sensible any more because the actions would overwrite each other. To avoid this, we argue that the agent should maintain some kind of *focus* and continuously select one and the same action.[5]

In the Tower World there is an elegant solution for creating this focus on a particular action by exploiting a simple fact about this environment: *the gripper can hold at most one block at a time*. By focusing on a single block that the agent should be holding, and if the agent holds that block, start focusing on the place where to put it down, the agent would create the focus that we are after. The idea is that by focusing on a single block and what to do with it (either holding it or putting it down somewhere) an agent can disregard any actions that are unrelated to the block that is the current focus of the agent.

The general strategy for creating the focus that we suggest to use here for dealing with durative actions in the Tower World is simple: *Introduce a goal that keeps track of (represents) the current focus*. In the Tower World environment this 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. Key to maintaining focus is that we also ensure that we only have a *single* goal to hold (or

[4]Note that in the Tower World blocks are labelled with letters from the alphabet and are not labelled `b<Nr>` with `<Nr>` a natural number as in the Blocks World.

[5]Another solution 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 need to keep track of whether an action has completed or not. Although perhaps feasible, the code needed to achieve this for the Tower World more likely than not will provide a rather ad hoc solution for the problem. The solution proposed in the main text appears much more elegant.

put down) a particular block. More concretely, we should have at most one goal of the form `holding(X)` present in the agent's goal base at any time. We call such goals *single instance goals*. The idea of a single instance goal `sig` to create focus can be implemented easily in GOAL by means of a simple pattern:

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

Note that a rule of this form only is applicable if the agent does not yet have a goal to hold a block because of the first **goal** condition. 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 block, and the second is that the block is obstructing another block that we need to have access to in order to be able to make a constructive move. 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)))), obstructingBlock(X) then adopt(holding(X)).
```

One issue that remains to be dealt with is how we can implement the idea to use single instance goals while making sure at the same time that such goals will be reconsidered whenever necessary. That is, how can we change focus if events in the Tower World make that necessary? We will see how to address this issue by a goal management strategy in the next section.

6.5.2 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. Assuming 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., when a user moves a block) which makes pursuing the goal no longer beneficial.

The selection of an action that the agent should perform next in the environment is very straightforward if we can assume that a goal to hold a block is present only if we are not yet holding a block and the agent only has goals that are feasible. In that case, if we want to hold a block, we simply pick it up. And if we are holding a block, the agent should put it in position whenever possible, and otherwise put it somewhere on the table. If none of the above applies, we put the gripper back in its initial position. The rules for implementing this strategy in our agent are pretty straightforward. The interesting part to note here is the nesting of rules. The second rule uses curly brackets {...} to include two other rules to make a case distinction between a block that can be moved into position and one that cannot.

```
main module{
  program{
    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 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 any more 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 behaviour 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 should we rather put them 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.

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 6.10).

6.6 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 this 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 sitting on top of another block are provided by means of the `on` predicate, and the `holding` predicate indicates the state of the gripper. 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 more informative representation of the environment.

Usually, however, environments do not provide sufficient information to construct a complete representation of the environment's state by means of the percepts that 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 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, as, for example, a gaming environment like UNREAL TOURNAMENT, it is impossible to ever construct a complete and accurate representation of the environment. Designing an agent's decision making gets even more complicated in such cases, as it then has no other choice than to make decisions based on incomplete information. The fact that observability is relative to a representation is important 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 environment would be the position of the agent.

6.7 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*.

6.8 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, 54].

6.9 Exercises

6.9.1

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 6.10 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.

6.10 Appendix

```
init module{
   knowledge{
      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), maintain.
   }
   program{
      #define constructiveMove(X, Y)
         a-goal( tower([X, Y | T]) ), bel( tower([Y | T]), clear(Y), (clear(X) ; 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 }
      }
   }
}

main module{
   program{
      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.
   }
}

event module{
   program{
      #define inPosition(X) goal-a( tower([X| T]) ).

      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) ).

      if goal( holding(X) ) then {
         if bel( not( clear(X) ) ) then drop( holding(X) ).
         if bel( holding(_) ) then drop( holding(X) ).
         if inPosition( X ) then drop( holding(X) ).
         listall L <- constructiveMove(Y,Z) do {
            if bel( not( L=[] ), not( member([X,_],L) ) ) then drop( holding(X) ).
         }
      }

      if not(goal( holding(X) )), not(bel( holding(Y) )) then adoptgoal.
   }
}

module adoptgoal{
   program{
      #define obstructingBlock(X) a-goal( on(Y, Z) ), bel( above(X, Z); above(X, Y) ).

      if constructiveMove(X, Y) then adopt( holding(X) ).
      if obstructingBlock(X) then adopt( holding(X) ).
   }
}
```

Figure 6.5: Tower Builder agent

Chapter 7

Communicating Agents

7.1 Introduction

A multi-agent system that consists of multiple agent processes that coordinate their behaviour in a decentralized manner has several advantages over a centralized system [61]. First, a decentralized system is more robust than a centralized system as it does not introduces a single point of failure. Second, a decentralized system consists of several logically distinct components whereas a centralized system may exhibit a lack of modularity. Third, a centralized system suffers from difficulty in scalability (by not utilizing existing agents as components). And, finally, a distributed system is often a better match with the way things are organized in reality.

In a multi-agent system, where agents need to coordinate their behaviour, it is useful for agents to communicate about their beliefs and goals. Agents may have only a partial view of their environment. By communicating, agents may *inform* each other about what they but other agents cannot perceive. Agents may also use communication to *share goals* in order to coordinate task allocation and avoid duplication of effort or interference while trying to achieve a goal. 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 and need to coordinate their actions to prevent deadlocks and inefficient interaction with other agents.

In this chapter we explain how to write agent programs for communicating agents. We start by explaining how a multi-agent system is created from a multi-agent system (MAS) file. A MAS file is used to launch a multi-agent system by creating agents and by connecting these agents to an environment, if available. A MAS file in essence is a *recipe for launching a multi-agent system.* We then introduce communication primitives and explain how messages are exchanged between agents. Processing messages is very similar to the processing of percepts but there are some important differences. For example, messages are not automatically removed from an agent's mailbox whereas percepts are removed every decision cycle from an agent's percept base. Agents can exchange three different types of messages. In Section 7.5 we explain that an agent can *inform*, *instruct*, and *ask* something from another agent. It is important to address the right agents when communicating. Section 7.6 explains how agents that should receive a message can be selected. Finally, an example multi-agent system is presented to illustrate the use of communication for coordinating the actions of multiple agents.

7.2 Launching a Multi-Agent System

This section explains how a multi-agent system is launched by means of a MAS file. We have already seen a simple example of two Blocks World agents being connected to an environment in Chapter 6. Here we discuss in more detail how a set of agents can be launched together to form a running multi-agent system. Several aspects of a multi-agent system are determined by a MAS file. For example, a MAS file determines *when an agent is launched* to become part of the multi-agent

masprogram	::=	[*environment*] *agentfiles launchpolicy*
environment	::=	**environment{ env =** "*path*"**.** [*initialization*.] **}**
path	::=	any valid path to an environment file
initialization	::=	**init** = [options]
options	::=	list of `key=value` pairs valid for environment that is used
agentfiles	::=	**agentfiles{** *agentfile*. { *agentfile*.}* **}**
agentfile	::=	"*path*" [*agentparams*] **.**
agentparams	::=	[*nameparam*] $\mid$ [*langparam*] $\mid$
		[*nameparam* **,** *langparam*] $\mid$ [*langparam* **,** *nameparam*]
nameparam	::=	**name** = *id*
langparam	::=	**language** = *id*
launchpolicy	::=	**launchpolicy{** { *launchinstruction* $\mid$ *launchrule* }$^+$ **}**
launchinstruction	::=	**launch** *basiclaunch* {, *basiclaunch*}* **.**
basiclaunch	::=	*agentbasename* [*agentnumber*] **:** *agentref*
agentbasename	::=	$*$ $\mid$ *id*
agentnumber	::=	[*number*]
agentref	::=	plain agent file name without extension, or reference name
launchrule	::=	**when** *entitydesc* **do** *launchinstruction*
entitydesc	::=	[*nameparam*] $\mid$ [*typeparam*] $\mid$ [*maxparam*] $\mid$
		[*nameparam* **,** *typeparam*] $\mid$ [*typeparam* **,** *nameparam*] $\mid$
		[*maxparam* **,** *typeparam*] $\mid$ [*typeparam* **,** *maxparam*]
typeparam	::=	**type** = *id*
maxparam	::=	**max** = *number*
id	::=	an identifier starting with a lower-case letter
num	::=	a natural number

Figure 7.1: Grammar for MAS Files

system. There are basically two options. Agents can be launched when the multi-agent system is initially created, or when an entity in the environment becomes available that can be controlled by an agent (we also say that an entity is *born*). A MAS file also determines *whether or not to connect an agent to an environment.*

7.2.1 A Recipe for Launching A Multi-Agent System: MAS Files

A multi-agent system 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 agents should be connected to, which agents should be launched, how many agents should be launched, and which agent files should be used to launch an agent. Multiple agents may, for example, be created using a single agent file. The MAS file determines which agents should be connected to an entity in an environment. It also specifies how to *baptise* agents: 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 MAS file consists of three sections (see also Figure 7.1):

- an **environment** section that specifies which environment (interface) should be launched,
- an **agentfiles** section that contains references to agent files used for creating agents, and
- a **launchpolicy** 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 an environment section is absent, this simply means that agents will not be connected to an environment. The MAS file for the Coffee Factory example that we introduce below does not have an **environment** section. 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.

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{
  env = "elevatorenv.jar".
}
```

It is often 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. For example, the type of simulation, the number of floors, the number of elevator cars, and other parameters can be initialized. The following is an example specification of parameters for the Elevator environment:

```
environment{
  env = "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 **agentfiles** section contains references to one or more agent files. The agent files are the files that are used for creating agents. Launch rules that are part of the **launchpolicy** reference these files. 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 provided next.

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

The files in the **agentfiles** section should be referenced in the **launchpolicy** section. If the **agentfiles** section only contains plain filename references such as above, the label used to reference an agent file is simply the filename *without the file extension*. For example, the reference that should be used for the elevator agent file above simply 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 **agentfiles** section are needed when an agent file is replaced with a different agent file. References to the file in the **launchpolicy** 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] .
  }
```

The example **agentfiles** section above 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 any more. 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]

The **launchpolicy** section specifies a policy for launching agents. It consists of one or more *launch rules* that indicate when an agent should be created. There are two types of launch rules: *conditional* and *unconditional* launch rules. An unconditional launch rule is of the form **`launch`** `<agentName>:<agentReference>`. The Coffee Factory multi-agent system uses these types of rules because this MAS is not connected to an environment. Unconditional launch rules cannot be used to connect agents to an environment.

```
  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 has just been born or has become available again. The next launch rule for the Elevator environment has a simple form that is often used when only one type of entity is available in the environment and only one type of agent needs to be connected to these entities.

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

Baptising Agents 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 entities that are available. If there is a free entity, applying the rule creates an agent using the agent file reference `fileref`, baptises 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 baptise the agent but a unique number is attached to distinguish the agent

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

from the other agents. For example, if four carriages are available, agents named `elevator`, `elevator1`, `elevator2`, and `elevator3` will be created.

Entities already have names themselves that have been assigned to them by the environment. It is possible to use the environment names of an entity for the agent that is connected to the entity as well by replacing the name with an asterisk $\star$ in a launch rule. This is useful for keeping track of which agent controls which entity. The asterisk means that the name of the entity in the environment is used as the base name for baptising agents.

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

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 in Chapter 6 for connecting multiple agents to an entity. There we simply listed multiple agent names with associated file references (see also the grammar in Figure 7.1).

Launch Rule Conditions A launch rule may impose additional conditions for triggering the rule. One condition that can be added is a condition that constrains the name of an entity. By using the **name** option a launch rule will only trigger if an entity with a pre-specified name is available. For example, the next rule will only be applied if the free entity has the name `car0`.

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

A second condition that can be added concerns 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. Finally, a third condition 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 these 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.

Rule Order in Launch Policy A final remark on launch policies concerns the order of rules. Rules in the **launchpolicy** 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.

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

By ordering the rules as is done above, 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`.

7.2.2 Agent Facts

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 not even be possible to determine their names when the agents are being developed.

To resolve this, the names of other agents are automatically inserted into the belief bases of all known agents. The keyword `agent` is used to insert 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, the keyword `me` is used to insert facts of the form `me(<agentname>)` into the agent's belief base where `<agentname>` is the name of the agent, as determined by the launch policy. After launching the `coffee.mas2g` file to create the Coffee Factory MAS and applying both launch rules, the belief base of the coffee maker agent will look like this:

```
beliefs{
   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 for the Elevator environment in Figure 7.2.

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

agentfiles{
   "elevatoragent.goal" [name=elevatorfile].
   "managingagent.goal" [name=managerfile].
}

launchpolicy{
   launch manager:managerfile.
   when [type=car,max=1]@env do launch elevator1:elevatorfile.
   when [type=car,max=1]@env do launch elevator2:elevatorfile.
   when [type=car,max=1]@env do launch elevator3:elevatorfile.
}
```

Figure 7.2: Example MAS File

This example uses relative paths to the files and labels to reference those files. The first rule will create an agent called `manager` that is not connected to the Elevator environment. The other three rules will launch elevator agents named, respectively, `elevator1`, `elevator2`, and `elevator3`. Each of these three agents will be associated with an entity in the environment of type `car`. Even if more elevator cars become available, no more agents will be created. After all three elevator agents have been launched, the belief base of, for example, `elevator2` will look like:

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

7.3 The Coffee Factory Example

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. Figure 7.3 lists a version of the agent program for the coffee maker without comments that we use here as a reference.

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 on 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.

```
init module{
  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).
  }
  actionspec{
    make(Prod) {
      pre{ forall(requiredFor(Prod, Req), have(Req)) }
      post { have(Prod) }
    }
  }
}

main module{
   program{
      if goal(have(P)) then make(P).
   }
}

event module{
   program{
      forall bel(agent(A), not(me(A)), not(canMake(A,_))) do A.sendonce(?canMake(A,_)).
      forall bel(me(Me), received(A, int(canMake(Me,_))), canMake(Me, Prod))
         do A.sendonce(:canMake(Me, Prod)).
      forall bel(received(Sender, canMake(Sender, Products))) do
         insert(canMake(Sender, Products))
         + delete(received(Sender, canMake(Sender, Products))).
      forall bel(agent(A), received(A, have(X))), not(bel(have(X))) do insert(have(X)).
      forall goal(have(P)), bel(requiredFor(P,R), not(have(R)), canMakeIt(Me,R), me(Me))
         do adopt(have(R)).
      forall goal(have(P)), bel(requiredFor(P, R), not(have(R))),
             bel(canMakeIt(Maker, R), not(me(Maker)))
         do Maker.sendonce(!have(R)).
      forall goal(have(P)),
             bel(requiredFor(P, R), not(have(R)), me(Me), not(canMakeIt(Me, P)))
         do allother.sendonce(!have(R)).
      forall bel(agent(Machine), received(Machine, imp(have(X))), have(X))
          do Machine.sendonce(:have(X)).
   }
}
```

Figure 7.3: The Coffee Maker

The following simplifying assumptions have been made in the design of 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.

7.4 Communication: Send Action and Mailbox

Communication between agents is supported by means of a so-called *mailbox semantics*. Messages received are stored in an agent's mailbox using the keywords `sent(<recipient>,<message>)` and `received(<sender>,<message>)` where `<recipient>` denotes the agent(s) that the message has been sent to, `<sender>` denotes the agent who is the sender of the message, and `message` denotes the content of the message. The content of a message must be a valid expression in the knowledge representation language that is used by the agents. For example, when using Prolog, the content of a message must be a conjunction of literals.

The action `<agentname>.`**`send`**`(<content>)` is a built-in action to send `<content>` to the agent with given `<agentname>`. `<agentname>` is an atom with the name of the agent as specified in the MAS file. Messages that have been sent are placed in the mailbox of the sending agent, as a fact of the form `sent(<agentname>, <content>)` (note the 't' at the end of `sent`). The message is sent over the selected middleware to the target agent, and after arrival the fact that a message has been received is recorded in the mailbox by means of a fact of the form `received(<sender>, <content>)` where `<sender>` 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.

7.4.1 Sending Messages: 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 groceryplanner.send(amountLeft(milk, 0)).
    ...
}
```

At the beginning of its decision cycle, the `groceryplanner` agent gets the following fact inserted in its mailbox:

```
    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 `<content>` from sender `<sender>`, then **`bel`**`(received(<sender>, <content>))` holds; the agent believes it has received the message. Similarly, by inspecting the sender's beliefs, we find that **`bel`**`(sent(<receiver>, <content>))` holds, where `<receiver>` 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)).
}
```

7.4.2 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 mailbox, even after execution of the action rule above. As a result, the groceryplanner will keep applying the action rule above in each of its decision cycles. Because the **send** action can always be performed, moreover, the fridge will also keep sending the same message again and again.

There are several ways to deal with 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. In most cases, however, when a message has been processed, we can also simply remove it. One way to do this is to immediately remove the message in the same rule that processes the message by adding a **delete** action to the end of the actions that are selected by 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 action rule will be selected only once.

The coffee maker agent from Section 7.2 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 groceryplanner.send(amountLeft(milk, 0)).
```

7.4.3 Sending Messages Once: The sendonce action

Because the above leads to verbose programming, a variant of the **send** action is available: 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 to the same receiver(s) in the agent's mailbox.

Writing, for example,

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

is short for

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

7.4.4 Variables

In agent programs, the use of variables is essential to writing effective agents. Variables can be used in messages as one would expect. For example, a more generic version of the `fridge`'s action rule would be

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

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

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

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

7.5 Moods

Cognitive agents have beliefs and goals but until now all examples have only shown how to communicate an agent's beliefs. As a side note, we remark that an agent that receives a message should be careful because the agent that send the message may not always speak the truth! In the examples above, every message has been a *statement* which typically is used to express what an agent *believes*. It would be useful if an agent can also communicate its *goals*. In natural language, the *mood* of what is said often indicates that a goal is communicated rather than a belief. Moods allow a speaker to express various different kinds of *speech acts* including an *indicative* speech act (‘The time is 2 o’clock’), an *expressive* speech act (‘Hurray!!’), and, for example, a *declarative* speech act (‘I hereby declare the meeting adjourned’). In the agent language, three different moods are supported. These moods are listed in Figure 7.5.

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 7.4: Moods of Messages

The indicative mood operator is optional. In other words, in absence of a mood operator, the indicative mood is implicit. That means that all examples in Section 7.4, for example, implicitly expressed the indicative mood. Using the declarative mood operator, agents can indicate that they are communicating about their goals. For example, if the coffee maker or coffee grinder needs a resource to make something but does not have it, it can let another agent know that it needs it. Using the rule below, moreover, an agent will only be bothered if the agent believes the other agent has the resource:

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

Upon receiving a message with a mood operator, the mood of the message is represented in the mailbox by means of special keywords which allow an agent to reason about the mood of a message. An imperative is represented by the keyword `imp`, and an interrogative mood by the keyword `int`. There is no special keyword for the indicative mood. Using these mood keywords, 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 messages. These messages are similar to open questions such as “What time is it?” or “What is Ben’s age?”. These cannot be represented by a closed sentence. When using Prolog, instead, a *don’t care* can be used to indicate the unknown component. For example:

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

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

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

7.6 Agent Selectors

In multi-agent systems, agents find themselves communicating with agents whose name they do not know beforehand. For example, the MAS may have been created by launching a 100 agents using a **`launch`** `agent[100]:file1` rule. It is also useful to be able to multicast or broadcast

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

Table 7.1: Syntax of the **send** and **sendonce** actions

a message to multiple receivers. *Agent selectors* can be used to select the agent or set of agents that needs to be addressed in cases like these. These selectors allow for more dynamic addressing schemes than would be possible if we only were allowed to use the agent names themselves. The available options for selecting agents to send a message to are documented in Figure 7.1.

Agent selectors are prefixed to **send** or **sendonce** actions and can be used to replace the hardcoded agent names in the previous section. An agent selector specifies which agents are selected for sending a message to. An agent selector can be a simple *agent expression*, a *quantor*, or multiple agent expressions separated by commas and surrounded by square brackets.

Some examples of agent selectors are:

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

% variable (Prolog)
Agt.send(theFact)

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

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

% example that uses quantor as agent selector
% 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 allother.send(!have(P)).
```

Agent Name The agent name is the simplest type of agent expression, which we have already seen in Sections 7.4 and 7.5. It consists of the name of the receiving agent. Several examples of prefixing **send** and **sendonce** actions with an agent name as selector have been given above. These examples have assumed that the knowledge representation language used by the agent is Prolog and that agent names start with a lower-case letter. Some additional examples including one that uses a list of agent names are:

```
alice.send(:hello).

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

If the agent name refers to an agent that does not exist in the MAS, to an agent has died, or the agent for some other reason cannot be addressed, the **send** action is still performed but the message will be lost. There is no feedback confirming or dis-confirming 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 dis-confirm the reception of the message.

Variables Using a *variable* as agent expression allows a dynamic way of specifying the message recipient. We have already seen several examples above that have used variables (when Prolog is used, an id starting with a capital is a variable). Variables allow to determine a recipient based on an agent's beliefs or goals or on previous conversations. A variable agent expression consists of a variable in the agent's knowledge representation language. This variable will be resolved when the mental state condition of a rule 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.

```
 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(_)))))
11         then X.send(!hold(gold)).
12
13      if goal(informed(Agent, fact(F))) then Agent.send(:fact(F)).
14
15      % This will never be selected:
16      if bel(something) then Agent.send(:something).
17    }
18  }
```

In the example above, the action rule on line 10 contains the variable `X`, which has two possible substitutions: `[X/john]` and `[X/mary]`. This results in there being two *options* for the action: `john.`**`send`**`(!hold(gold))` and `mary.`**`send`**`(!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 suitable for multi-casting messages.

Quantors Quantors are a special type of agent expression that allow to refer to sets of agents or to arbitrary agents. There are four possible quantors that can be used in combination with the communicative actions **send** and **sendonce**: **all**, **allother**, **some**, and **someother**. Each of these quantors is expanded to one or more agent names when a **send** or **sendonce** action is performed in the following way:

- **all** expands into the set of all names of agents that are currently present in the belief base of the agent including the name of the sending agent itself.
- **allother** expands into the set of all names of agents that are currently present in the belief base of the agent excluding the sending agent's name.
- **some** expands into an arbitrary agent name that is currently present in the belief base of the agent (including the name of the sending agent itself).
- **someother** expands into an arbitrary agent name that is currently present in the belief base of the agent (excluding the name of the sending agent itself).

Using a quantor to address the recipients when sending a message does not result in the quantor being put literally in the mailbox. Rather, the actual agent names that the quantor resolves to are substituted, and a `sent(..)` fact is inserted for every agent that is addressed by means of the quantor. This has consequences for querying the mailbox when using quantors. It is *not* possible to test if a message has been sent to *all* agents, for example, by using a condition like that in the following rule:

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

It simply is not the case that `sent(all, fact))` is present in the mailbox of an agent after performing an **`all.send`**`(fact)` action. Another condition is required to test that all agents have been addressed. Using Prolog, this can be done, for example, by a rule condition as follows:

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

This will execute if the message has not been sent to all agents that the sending agent believes exist. This also means that after sending of the original message, if new agents would join the MAS, then the condition no longer holds because new `agent` facts would have been added.

Self and This Instead of the quantors discussed above, an agent can also send messages to itself by means of the keyword **self**. Using **this** instead of **self** has the same effect. In other words, **self** resolves to the sending agent's name.

7.7 The Coffee Factory Example Continued

In Section 7.3 the Coffee Factory MAS has been introduced. In this section we will have a more detailed look at the coffee maker and coffee grinder agents. In particular, we will see how the agents coordinate their actions by communicating in various ways.

7.7.1 Capability Exploration

The agents know what they can make themselves. This is represented as beliefs in the agent program. For the coffee maker, these 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 A.send(?canMake(A,_)).

   % answer any question about what this agent can make
   forall bel(me(Me), received(A, int(canMake(Me,_)), canMake(Me, Prods))
      then A.send(: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 whether there are agents `A`, other than the agent itself, which did not yet receive a question (`int`) what they can make. For each of these agents, if the agent does not yet know what they can make, the rule is applied and those agents will be ask them which products they can make. The condition **`not`**`(me(A))` ensures that `A` will not be instantiated with the agent's own name and thus prevents that the agent will ask itself what it can make. Strictly speaking this condition is redundant because every agent can make at least something and the condition **not**(canMake(A,_)) will fail for the agent itself (i.e., **bel**(me(Me), canMake(Me,_)) holds). Also recall that after execution of a **send** action, a sent fact is inserted in the mailbox.

The second rule handles incoming questions (interrogatives). The condition inspects the mailbox and checks for received interrogative messages that ask what this agent can make. It replies to the sender with an *indicative* message to let it know what it can make. The rule also removes the received message from the mailbox in order to prevent it from being triggered again and again.

The third rule handles indicative messages that provide information about other agent's capabilities. The condition checks for received messages that contain the predicate canMake. For all these messages, the information is inserted as a fact in the agent's belief base. Thereafter, the received message is removed from the mailbox to prevent repeated execution of the action rule.

7.7.2 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 action rules, it can request the grinder to make grounds by sending it an *imperative* message:

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

When this agent has a goal to make some product P for which it needs an ingredient R which it does not have, and it knows that one or more agents can make R, by applying this rule the agent will send an imperative message to each of these agents. The message that is sent is !have(R) with R instantiated with some product. This message indicates that the agent has a goal to have ingredient R and may be taken by the other agents as a request to provide R.

In the Coffee Factory MAS where agents cooperate, agents that receive this message and can make the requested ingredient will adopt a goal to make it:

```
forall bel(received(A, imp(have(P))), me(Me), canMakeIt(Me,P)) do adopt(have(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 Factory do not need to actually 'give' each other products but because other agents do not perceive that products are available if they have been made by others, they rely on communication to inform each other about the availability of an ingredient.

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

The idea is that after informing the agent that an ingredient is available it also will have access to the ingredient. At that time we *can and should* remove the request to provide an ingredient because we have completed everything that was needed to fulfil the request.

Upon reception of an indicative message :have(P) does not automatically result in the belief by this agent that have(P). The agent needs to explicitly update its beliefs when it receives a message with new information from another agent.[2]

[2]This is where we make a leap of faith. The other agent suggests by sending the message that it believes have(P). The reason why the receiving agent starts believing it as well is because it trusts the sender.

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

7.8 Notes

As noted in Chapter 2, 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 [16], 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 [21]. Providing a formal semantics of communication has received some attention in agent-oriented programming research. Some agent programming languages used to use middleware infrastructures such as JADE [9] 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.

7.9 Exercises

7.9.1 Milk cow

The Coffee Factory example from Section 7.3 consists of a coffee maker and grinder. Suppose we now also want to make lattes, i.e., 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.[3]
   (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.

[3]as far as humans can tell.

Chapter 8

Design Guidelines

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.

8.1 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 6.

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 8.2.2 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.*

8.2 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 [11] or [58].* GOAL uses SWI Prolog; for a reference manual of SWI Prolog see [60].

8.2.1 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).

8.2.2 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.

8.3 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.

8.3.1 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.

8.3.2 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) }
}
```

8.3.3 Action Specifications for Non-Environment Actions

Actions that are included in the action specification of an agent program but are not made available by the agent's environment should be annotated with **@int**. That is, directly after the declaration of the name of the action (and its parameters) you should write **@int**. Actions that are not built-in actions of GOAL are sent to the environment for execution if the action's name is not annotated with **@int**. However, if the environment does not recognize an action, it may start throwing exceptions.

Note that in principle there is no reason for introducing specifications for actions that are not available in the environment. Any action that is only used for modifying an agent's state can also be programmed using the built-in **insert** and **delete** actions of GOAL.

8.4 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.

8.4.1 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.

8.4.2 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.

8.5 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.

8.5.1 All Modules Except for the Main Module Should Terminate

The only module that does not need to terminate is the **main module**. All other modules, including the **init** and **event** modules but especially the modules you introduce yourself should terminate. Note that otherwise the module would never return to the top-level **main module** which is considered bad practice.

8.5.2 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) :- ...
```

8.5.3 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.

8.6 Notes

The guidelines discussed in this chapter are based in part on research reported in [49, 37].

Chapter 9

Unit Testing Modules

Testing your code is an integral part of developing a multi-agent system that meets its design objectives. To this end, we introduce a unit testing framework for GOAL in this chapter. The smallest part of an agent that can be tested using this framework is a single action but typically it is more useful to test a single module (which can be viewed as a composed action).

9.1 A Simple Example: Test2g Files

Let's revisit one of our "'Hello World" agents that we created in Chapter 1. We will use the MAS with an agent that printed "Hello, world!" 10 times by using the **print** command. We want to write a test that verifies that the agent indeed prints the message exactly 10 times. That is, we want to test whether the **print** action in the **main module** of the agent is executed exactly 10 times. For convenience, we repeat this module here.

```
main module [exit = nogoals] {
   program{
      if goal( nrOfPrintedLines(10) ) then print("Hello, world!").
   }
}
```

The important first step now is to identify the condition that we want to verify. We will reuse the counter `nrOfPrintedLines` that the program maintains for keeping track of how many times the **print** action has been performed (recall the **event module** of the agent discussed in Chapter 1). Informally, we want to say that *at the end of the program this counter says the **print** action has been performed 10 times.* To express this condition in a test we use the **atend**`(msc)` operator. This operator checks, as the name already indicates, whether the `mental state` condition holds directly after the module terminates. We use the condition **bel**`(nrOfPrintedLines(10))` below to inspect the counter that is stored in the agent's belief base.

```
masTest {
   mas = "HelloWorld10x.mas2g".

   helloWorldAgent {
      testPrinted10x {
         do print("Test that agent prints Hello, world! 10 times:").

         evaluate {
            atend bel( nrOfPrintedLines(10) ).
         } in do main.
      }
   }
}
```

After running the test (see the GOAL User Manual, [36]), you should see something similar to:

```
Test that agent prints Hello, world! 10 times.
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
passed:
   test: GOALagents\HelloWorld\HelloWorld10x.test2g
   mas : GOALagents\HelloWorld\HelloWorld10x.mas2g
   passed: helloWorldAgent

Tests: 1 passed: 1 failed: 0
```

The test itself is specified within the section in the test file labelled `testPrinted10x`. The test starts with a **do** section that executes a **print** command with the text that appears at the start of the output of the test listed above. This is followed by the real test that specifies to evaluate the **atend** test condition while executing the **main module**. The `testPrinted10x` section is part of a bigger context labelled `helloWorldAgent`. As you will recall (see Ch. 1), that is the name of the agent that prints the "Hello, world!" message that also occurs in the launch rule in the MAS file `HelloWorld10x.mas2g` that you can find at the beginning of the test file. A test always requires a name of an agent that also is launched by the MAS file. This agent is needed because a test requires a mental state of the right agent to evaluate test conditions. Of course, the module that is tested also must be present in the agent's program.

It may appear to you that the agent rather than the **main module** has been tested. In this example, there is not really a difference and you would be just as right to say that the agent has been tested as it is to say that the **main module** has been tested. We will later see how to test individual modules rather than an agent, but we will need agent programs that consist of more modules to do that.

9.2 Agent Programs, Assertions, and Test Conditions

A test file always starts with a MAS file declaration that is followed by a series of tests that are performed for a particular agent program. Optionally, also a time out (a positive float) can be specified; time outs are explained in Section 9.3. A test starts with the name of an agent that is also launched in the MAS file; tests are performed on modules that are part of that agent's program. The actual test starts by providing a name for the test and includes one or more test sections that are executed in sequence. Test sections either execute an action or a module, assert a condition, or evaluate test conditions while executing a module (or action). See for details the grammar in Table 9.1.

9.2.1 Agents, Programs, and Tests

A test always is associated with an agent in a test file. The test is specified after specifying the name of the agent that it should be associated with. A test is a test on part of that agent's program. It tests, for example, a module that is part of the agent's program. Only program parts that belong to one particular agent, moreover, can be tested in a single test. And a test must have access to the mental state of an agent for evaluating test conditions.

unitTest	::=	**masTest {**
		mas = "*fileName***.mas2g".**
		[**timeout** = *float***.**]
		agentTested$^+$
		}
agentTested	::=	*agentName* **{**
		test$^+$
		}
test	::=	*testName* **{**
		testSection$^+$
		}
testSection	::=	*doSection* | *assertion* | *evaluateModule*
doSection	::=	**do** *actionOrModuleCall* **.**
assertion	::=	**assert** *mentalStateCondition* [**:** **"***someText***"**]**.**
evaluateModule	::=	**evaluate {**
		testCondition$^+$
		} in *doSection* **.**
testCondition	::=	**atstart** *mentalStateCondition* **.**
		| **eventually** *mentalStateCondition* **.**
		| **always** *mentalStateCondition* **.**
		| **atend** *mentalStateCondition* **.**

Table 9.1: Test File Grammar

Not only do we need a mental state for evaluating conditions, we also need to connect an agent to the environment it is supposed to be connected with and in which it can perform actions and receive percepts. That is, when parts of an agent's program are tested, we still need to be able to perform the actions that are performed in the program as if the agent was running normally. For this reason we do not only create an associated agent but launch the whole MAS that the agent belongs to. Another reason for launching the complete MAS is that agents might (need to) exchange messages with each other.

To better understand what happens when a test is executed, it is useful to explain how a test is run. When running a test, the first thing that is done is to launch the multi-agent system as specified in the MAS file at the start of the test. The agents are created in line with the MAS file's launch policy. So far, everything is similar to running a MAS. Even when the agent is started the same things happen as the first thing the agent does is to initialize it's mental state by means of its **init module** and thereafter execute the **event module**, if available. That is, *an agent's* ***init*** *and* ***event module*** *are performed as usual when a test is executed.* Only at this point, the control of the agent is taken over by the test and the instructions in the test are followed instead of those in the agent's **main module**.

9.2.2 Executing Actions or Modules in a Test

We have already seen an example of a test section that performs a **print** action to add, e.g., some explanation about the test. This is just one simple example of what can be done. We can add any built-in action, such as an **insert** action, or any user-specified action or module that has been defined in the agent's program.

This is often useful to setup or prepare a particular test. Built-in actions can be used to prepare the mental state of the agent before a test is performed. Actions that can be performed in the agent's environment can be used to create an initial situation in the environment in which the test should be performed. The possibility to also execute modules simply makes it easier to bring an agent in a particular state before the test starts. It is also sometimes useful to write modules that are specifically created to set up a test.

9.2.3 Assertions

An assertion simply is a test on the agent's mental state that should hold if the test is to succeed. That is, an assertion evaluates a mental state condition. We could, for example, add an assertion to the test for the "Hello World" agent to verify that before we start executing that agent's **main module** the counter initially is 0. In fact, it is better to make sure this is the case because otherwise the test condition performed at the end might not reassure us at all that the **print** action was performed 10 times.

```
masTest {
   mas = "HelloWorld10x.mas2g".

   helloWorldAgent {
      testPrinted10x {
         do print("Test that agent prints Hello, world! 10 times:").

         assert bel( nrOfPrintedLines(0) ) : "counter should be initially 0".

         evaluate {
            atend bel( nrOfPrintedLines(10) ).
         } in do main.
      }
   }
}
```

Assertions before running a module thus can be used for specifying initial conditions, or preconditions, that should hold at the start. By adding assertions after a module has been terminated, it is possible to specify postconditions that should hold.

Assertions can also include a message to remind a tester about the purpose of testing the condition. Typically, a reason is supplied that indicates what should have been the case. If a test passes, as the example above does, the assertion silently succeeds and the message is not shown. Only if a test fails, the message associated with the assertion is shown. For example, suppose we would have made a typo and asserted that **bel**(nrOfPrintedLines(1)), then the test would fail and we would get a report like this:

```
testPrinted10x {
   executed: do print('Test that agent prints Hello, world! 10 times:')
   failed: assert bel(nrOfPrintedLines(1)): counter should be initially 0
}
```

If something else would have failed, we would get a report saying that the assertion was passed successfully.

Exercise 9.2.1

1. Modify the test for the "Hello World" agent and add an assertion as postcondition to verify that performing the agent's **main module** results in a belief that nrOfPrintedLines(10). Simply replace the **evaluate ... in** section with **do main**. Run the test again. Does the modification change what is tested?

2. Remove the assertions from the test again and reinsert the **evaluate ... in** section. Can you add a test condition to this section that verifies that the precondition **bel**(nrOfPrintedLines(0)) holds initially? Check the grammar in Table 9.1.

9.2.4 Test Conditions

Assertions already get us a long way for testing whether an agent is behaving as expected. They do not allow us, however, to verify what happens while a module is being executed. In particular, there are two cases that we cannot test for with only assertions. First, we cannot test for *liveness*, i.e., that some condition becomes true or false while executing a module. Second, we cannot test for *safety*, i.e., that some condition always remains true or false while executing a module.

To be able to test for these conditions, we can add test conditions that are evaluated while a module is being executed by means of an **evaluate ... in** test section. To check for liveness, use the **eventually** operator and to check for safety use the **always** operator. Preconditions and postconditions can also be checked by means of the **atstart** and **atend** operators, respectively.

Exercise 9.2.2

If we would only be interested in testing primitive actions, i.e., built-in actions or user-specified actions, would we need other test conditions than **atstart** and **atend** conditions? Would assertions be sufficient if all we would like to test is the behaviour of primitive actions?

9.3 Time Outs

Sometimes, it is useful to add time outs to tests. One reason for doing so is that a module may never terminate. If a module does not terminate, that module should be the top level **main module**; see Section 8.5.1. If a module does not terminate, a test that uses that module will also not terminate. It may still be useful, however, to verify that a test condition holds within, say, a number of seconds. We will see an example of this below. Another reason for adding a time out is simply to verify that a module terminates within a reasonable time frame. If a module should but for some reason does not terminate, running a test will also not terminate. By simply adding a time out you will make sure that the test terminates and that you will get some output on what happened. For example, if we would not have included an exit option for the **main module** of our "Hello World" agent (something that is easy to forget), the module would never terminate (which is the default behaviour of the **main module**).

```
main module {
   program{
      if goal( nrOfPrintedLines(10) ) then print("Hello, world!").
   }
}
```

So, suppose we would have used the **main module** above in our `helloWorldAgent10x.goal` file. This module will not terminate but in order to make sure that our test will terminate we add a time out option of 1 second just below the MAS file declaration.

```
masTest {
   mas = "HelloWorld10x.mas2g".
   timeout = 1. % seconds

   helloWorldAgent {
      testPrinted10x {
         do print("Test that agent prints Hello, world! 10 times:").

         evaluate {
            atend bel( nrOfPrintedLines(10) ).
         } in do main.
      }
   }
}
```

After running the test again, you should see something similar to:

```
Test that agent prints Hello, world! 10 times:
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
failed:
   test: GOALagents\HelloWorld\HelloWorld10x.test2g
   mas : GOALagents\HelloWorld\HelloWorld10x.mas2g
   failed: helloWorldAgent
      testPrinted10x {
         executed: do print('Test that agent prints Hello, world! 10 times:')
         interrupted: evaluate {
            interrupted: atend bel(nrOfPrintedLines(10)).
         } in do main.
      }

Tests: 1 passed: 0 failed: 1
```

As you can see, the test failed because the test was interrupted by the time out. The output indicates that the initial action to print the leading text was executed successfully but that the evaluation of the **atend** operator was interrupted. Clearly, the agent did not terminate within a second. Moreover, in this case it is easy to see that the agent did successfully print "Hello, world!" 10 times and could easily have terminated within a second. The issue thus must be with the termination of the agent, as we already knew, because we removed the **exit** option.

9.3.1 Evaluating Temporal Operators in Test Conditions

Time outs also may help to clarify the meaning of the temporal operators that are used in test conditions. Because it takes time before we can establish whether a temporal test condition holds or not, at the start of the test it is not yet possible to determine whether the condition is passed or failed. For example, suppose that an agent is counting numbers but is interrupted before it gets to count till 10000; in that case, it cannot be decided whether the test condition **eventually bel**`(counter(10000))` would have been passed or not if the agent would not have been interrupted. The situation is perhaps even more clear for **always** conditions. The basic idea is that such temporal test conditions can be only be settled if the run of the agent has completed. If at some moment a test is interrupted, it simply is not yet be possible to determine whether a condition would remain always true while running the agent.

The fact that the value of a temporal test condition may be unknown is also a difference with assertions. Assertions, when they are evaluated, always pass or fail; it is never unclear whether an assertion passed or failed.

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.:

[5] 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

[6] 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)

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

[8] 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)

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

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

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

[12] 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)

[13] Bordini, R., Dastani, M., Dix, J., Seghrouchni, A.E.F. (eds.): Multi-Agent Programming: Languages, Platforms and Applications. No. 15 in Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer-Verlag (2005)

[14] Bordini, R.H., Dastani, M., El Fallah Seghrouchni, A. (eds.): Multi-Agent Programming: Languages, Tools and Applications. Springer-Verlag (2009)

[15] Boutilier, C.: A unified model of qualitative belief change: A dynamical systems perspective. Artificial Intelligence **1-2**, 281–17316 (1998)

[16] 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)

[17] 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)

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

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

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

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

[22] 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

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

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

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

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

[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., 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)

[31] 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)

[32] Hindriks, K., Riemsdijk, B., Behrens, T., Korstanje, R., Kraayenbrink, N., Pasman, W., Rijk, L.: Unreal goal bots. In: F. Dignum (ed.) Agents for Games and Simulations II, *Lecture Notes in Computer Science*, vol. 6525, pp. 1–18. Springer Berlin Heidelberg (2011). DOI 10.1007/978-3-642-18181-8_1. URL http://dx.doi.org/10.1007/978-3-642-18181-8_1

[33] Hindriks, K.V.: Programming rational agents in goal. In: A. El Fallah Seghrouchni, J. Dix, M. Dastani, R.H. Bordini (eds.) Multi-Agent Programming:, pp. 119–157. Springer US (2009). DOI 10.1007/978-0-387-89299-3_4. URL http://dx.doi.org/10.1007/978-0-387-89299-3_4

[34] 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)

[35] 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)

[36] Hindriks, K.V., Pasman, W.: The GOAL User Manual. http://ii.tudelft.nl/trac/goal/wiki/WikiStart#Documentation (2014)

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

[38] 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)

[39] Johnson, M., Jonker, C., Riemsdijk, B., Feltovich, P., Bradshaw, J.: Joint activity testbed: Blocks world for teams (bw4t). In: H. Aldewereld, V. Dignum, G. Picard (eds.) Engineering Societies in the Agents World X, *Lecture Notes in Computer Science*, vol. 5881, pp. 254–256. Springer Berlin Heidelberg (2009). DOI 10.1007/978-3-642-10203-5_26. URL http://dx.doi.org/10.1007/978-3-642-10203-5_26

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

[41] 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)

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

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

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

[48] Rao, A.S., Georgeff, M.P.: Bdi agents: From theory to practice. In: Proceedings of the First International Conference on Multiagent Systems, pp. 312–319. AAAI (1995). http://www.aaai.org/Papers/ICMAS/1995/ICMAS95-042

[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] van Riemsdijk, M.B., Hindriks1, K.V., Jonker, C.M.: An empirical study of cognitive agent programs. Journal Multiagent and Grid Systems (2012)

[51] Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 3rd edn. Prentice Hall (2010)

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

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

[54] 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)

[55] 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)

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

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

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

[59] `http://www.swi-prolog.org/` (2014)

[60] `http://www.swi-prolog.org/pldoc/man?section=builtin` (2014)

[61] Tambe, M.: Tracking dynamic team activity. In: AAAI/IAAI, Vol. 1, pp. 80–87 (1996). URL `http://www.aaai.org/Papers/AAAI/1996/AAAI96-012.pdf`

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

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

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

Index