# Programming Cognitive Agents in GOAL



January 4, 2017

# Contents

# Preface

GOAL is a rule-based programming language for programming **cognitive agents** that interact with an **environment** and with each other. Agents receive information about their environment through **percepts** and can request the environment to perform **actions**. Agents are part of a **multi-agent system** and can exchange information between themselves through **messages**. Cognitive agents maintain a cognitive state that consists of the **knowledge**, **beliefs**, and **goals** of the agent which are represented in some knowledge representation (KR) language. GOAL promotes a view of agent-oriented programming as **programming with cognitive states**. These states have additional structure compared to more traditional database programming and are very different from states in most other programming languages such as object-oriented programming. Cognitive agents are **autonomous decision-making agents** that derive their choice of action from their beliefs and goals.

The language offers a rich and powerful set of programming constructs and features for writing **agent programs**. The platform developed for running these agent programs also provides support for connecting multi-agent systems to environments such as simulators, games, and robots and for running multiple cognitive agents in a distributed computing environment. Agents can be connected to various environment where they control entities such as grippers for moving blocks, cars in traffic simulators, or bots in real-time games. There is no limit to the possibilities here and a diverse set of environments has been made available that can be used for programming cognitive agents (see Chapter 1 for examples).

The GOAL agent programming language has been significantly revised to address some of the shortcomings of the initial version of the language. The basic language elements are the same but the grammar has been changed to address issues related to dependencies between various elements used in the language. For example, the language now requires that a programmer makes dependencies on the KR and the actions available in an environment explicit. This programming guide has been revised accordingly to reflect these changes and introduces the revised language. In addition, a testing framework is now available for agent programs, developed by Vincent Koeman. A completely new chapter is dedicated to discussing this framework.

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 [38], and more challenging environments such as an elevator simulator, the Blocks World for Teams [31], and the real-time UNREAL TOURNAMENT 3 gaming environment [25]. 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 sup-

port 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

and to visit the website for the Eclipse plugin to install GOAL support for the Eclipse environment:

http://goalhub.github.io/eclipse

In order to be able to further improve the language GOAL and its development environment, we very much appreciate your feedback. We hope to learn more from your experience with working with GOAL. Please do not hesitate to contact us at goal@ii.tudelft.nl and let us know what you think! In order to make the most out of this programming guide, as is the typical advice when learning any programming language, the reader is advised to practice and do the exercises made available in this guide. A final note: this guide is about the programming language and explains how to write agent programs but does not explain the development environment. Please consult the User Manual [28] to make the most out of the extensive support integrated into Eclipse for developing agent programs.

*Koen V. Hindriks*, Utrecht, March, 2016

**Acknowledgements**
Getting to where we are now would not have been possible without many others who contributed to the development of the GOAL language and its development environment. 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, Nils Bulling, 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, to Wouter Pasman, for all the programming work, and, in particular, to Vincent Koeman, who not only developed Eclipse support for programming GOAL agents but also designed and developed a testing framework for GOAL.

# Chapter 1

# Cognitive Agents & Environments

Before we will write a simple chat agent program in Chapter 2, we first introduce the basic programming model of **cognitive agents that interact with an environment**. Agents control entities in environments and decide what these entities do. The core of **agent-oriented programming** is a model of decision-making where agents make decisions based on their beliefs and goals. We call agents that derive their choice of action from their beliefs and goals **cognitive agents**. In this chapter we identify the key components and capabilities of a cognitive agent that enable it to effectively interact with its environment.

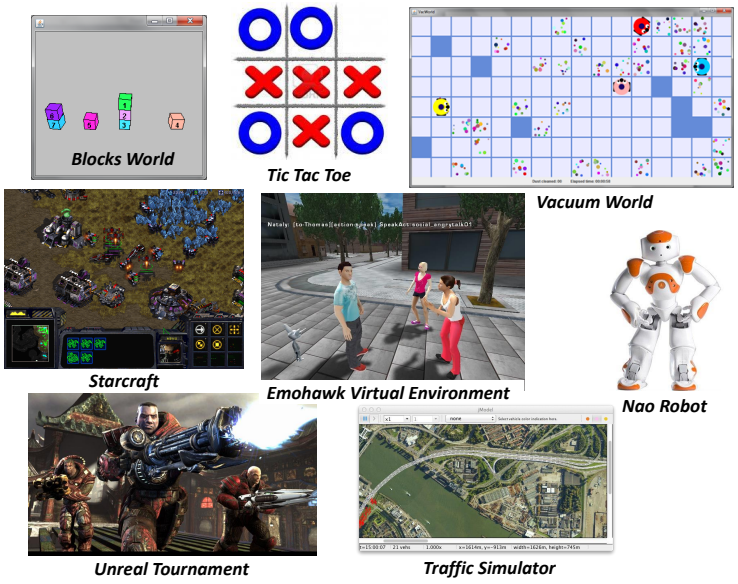


Figure 1.1: Example Environments

## 1.1 Environments and Controllable Entities

Environments can be almost anything ranging from toy worlds, grid worlds [21], simulators, games, virtual environments, to physical robots. Figure 1.1 illustrates some example environments. A classic example from artificial intelligence is the Blocks World [38]. This is a simple world (or, environment) that allows blocks to be moved by means of a gripper. The vacuum world is a grid world where virtual robots need to remove dust from the cells in the grid. StarCraft and Unreal Tournament are examples of well-known real-time strategy games where a player needs to battle with its opponents. The Nao robot is a well-known physical humanoid robot.

The environments that agents interact with consist among other things of **controllable entities** that can perform actions in the environment. Examples are the gripper in the Blocks World, a bot, character, or unit in a game, or a robot that acts in the real world. An agent that is *connected* to a controllable entity in an environment can control the **action**s that the entity performs. It will also see what the entity sees in the environment by receiving **percept**s. Figure 1.2 illustrates the connection between a cognitive agent and an entity. A useful way of thinking of a controllable entity is that it is the body of the cognitive agent that controls it. You can think of the cognitive agent as the mind that controls the body. This means that the agent makes the decisions about which action the entity should perform. Note that we do not picture the cognitive agent as part of its environment but rather think of the agent being connected to an (entity in an) environment.

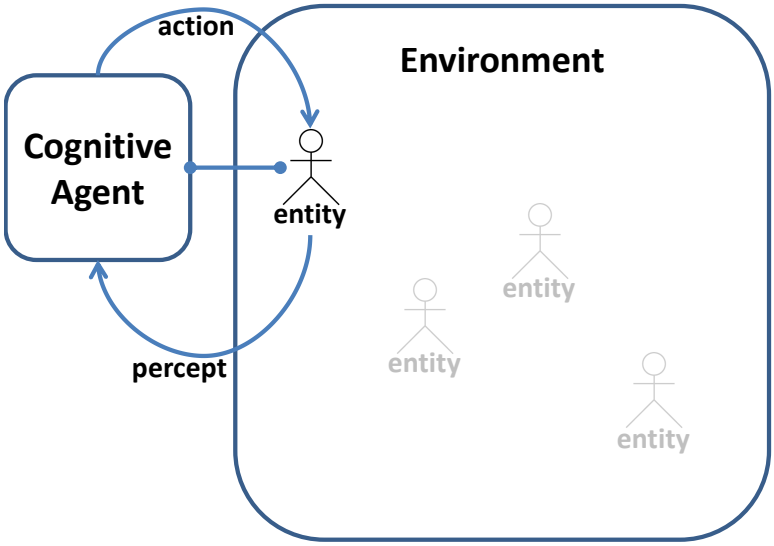


Figure 1.2: Cognitive Agent Connected to Entity in Environment

It is very important to realize that an agent and the environment it interacts with are separate processes. Things may happen in an agent's environment while the agent is deciding about what to do next. Only in a very special class of environments such as the classic Blocks World nothing changes if an agent does not decide to make the gripper move a block. In other environments, things are not fully controlled by the agent. For example, dust and dirt may (re)appear in the vacuum cleaning world in cells which the agent has cleaned before. The difficulty for programming such an agent is that it may also not perceive these changes and needs to revisit cells it already visited. In two or more player games such as Tic Tac Toe game play depends on the agent's opponent. In real-time strategy games, moreover, the time an agent takes to make a decision matters.

## 1.2 Cognitive Agents

A cognitive agent has three core abilities that enable it to control and interact effectively with an entity in an environment. The first ability is **event processing**. This enables the agent to process events such as percepts that it receives from the environment and messages that agents exchange between each other to update what it believes about its environment and other agents. Events may also prompt an agent to update what it wants. The second ability is **representing knowledge**. This enables the agent to *maintain a model of the environment*, which is essential for keeping track of the state of the environment. It also enables the agent to *reason about its environment* and to represent *what it wants to achieve*. The third ability is **decision-making**.

This enables the agent to decide on what to do next and *select an action*. These core abilities relate to important areas in artificial intelligence.

Computer vision, for example, is needed to process raw camera images (one type of percept) that a robot captures into symbolic knowledge. We will take this process of extracting knowledge from what a robot sees for granted in the remainder. One reason for doing so is that in games and other virtual environments the problem of knowledge extraction is largely absent and the task that remains is percept processing in the sense of updating an agent's state. We will therefore not be concerned with how sensor output is processed to make sense out of information obtained through sensors. What is important in this context is to realize that the percepts that an agent receives usually only inform it about what the entity that the agent controls senses or perceives, e.g., sees, feels, etcetera.

For representing knowledge, we will, however, directly rely on the area of *knowledge representation* (KR). This area in artificial intelligence concerns itself with languages and techniques that enable a system to represent and reason about something. A cognitive agent needs a knowledge representation language for representing and reasoning about the environment it interacts with. We will use a knowledge representation language for representing an agent's percepts, knowledge, beliefs, goals, and messages that agents exchange but also for specifying the pre- and post-conditions of actions. In the next chapter we will learn how the language Prolog can be used to represent the state of a cognitive agent.

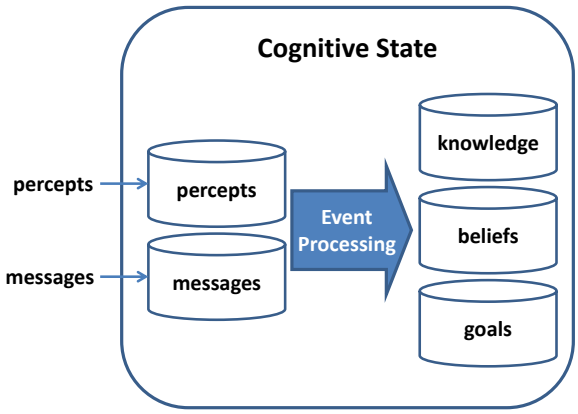


Figure 1.3: Cognitive State

A **cognitive state** may consist of various components but typically includes components for representing events, informational, and motivational components. Events include percepts and messages. Informational components contain the **knowledge** and **beliefs** of an agent. Knowledge here refers to *concept definitions* and *domain information* which is and will remain true no matter what. Beliefs refer to information that may change over time. Motivational components represent what the agent wants to do or achieve. A motivational component could represent e.g., the plans or intentions of an agent. We will use a motivational component that represents the **goals** that the agent wants to achieve. Goals are states of the environment that the agent wants to realize.

Finally, decision-making is related to automated planning. This area in artificial intelligence concerns itself with languages and techniques for representing and solving planning problems. A plan is a sequence of actions that achieves a goal of the agent. In order to find such a plan a planner needs information about what the agent believes is the initial situation and the goal(s) that the agent wants to achieve. **Action specifications** are important input needed to find a plan. Such specifications inform an agent when an action can be performed and what the effects of performing an action are. This is important knowledge that a cognitive agent needs when it makes decisions. We will therefore borrow the technique of specifying actions from planning and incorporate action specifications as a component in our agent programs.

## 1.3 Summary

The purpose of this chapter was to provide a brief introduction into **cognitive agents** that interact with an **environment** and to identify the key components that a cognitive agent needs to interact effectively with this environment. An agent is connected to a **controllable entity** that it can tell what to do and from which it receives percepts about the environment.

What makes an agent a cognitive agent is that it has a **cognitive state** that consists of:

- **percepts** that the agent receives from the environment and inform the agent about what the entity it controls perceives;
- **messages** that the agent receives from other agents;
- **knowledge** including domain information and conceptual definitions that are true always;
- **beliefs** which keep track of the state of the environment and change over time;
- **goals** which are states of the environment that the agent wants to realize.

A cognitive agent has three core abilities to effectively interact with its environment:

- **event processing** which allows an agent to update its beliefs and goals based on the percepts and messages it receives;
- **representing knowledge** which allows an agent to represent and reason with its knowledge, beliefs, and goals;
- **decision-making** which allows an agent to select an **action** to perform next based on its current beliefs and goals.

Finally, an **action specification** informs an agent about when an action can be performed and what its effects will be which is important and useful information needed for decision-making.

## 1.4 Notes

Although we have hinted at the possibility to construct systems of multiple cognitive agents, we have left the topic of programming *multi-agent systems* for later. Because a basic specification of a multi-agent system is also needed for running a single agent, the next chapter will already introduce the basic elements of a multi-agent system.

A warning is in place with regards to the use of terminology. We call agents cognitive because their state consists of knowledge, beliefs, and goals. The use of these labels is useful because it suggests a basic understanding of what the components of an agent's state are. Similarly, describing what an agent does when it selects actions as the ability to make decisions helps us understand intuitively what a cognitive agent program does. It is important to emphasize, however, that by using these labels to describe agents we do not want to suggest any strong correspondence of the components or abilities of these agents with similar human cognitive functions. In other words, we do not claim any psychological realism for our cognitive agents.

## 1.5 Exercises

**Exercise 1.5.1**

Select two environments of your own choice and for each of these do the following:

- identify a controllable entity in the environment;
- list the percepts you think the entity might send to an agent that it is connected with;
- list the actions you think that entity can perform in the environment;
- for each action specify when the action can be performed (i.e., which conditions must hold) and what the effects of successfully performing the action will be.

**Exercise 1.5.2**

For the entities that you identified in the previous exercise, specify a list of possible goals that an agent that controls those entities might have.

**Exercise 1.5.3**

In decision theory, the notion of **utility** is used to represent the preferences of an agent. In essence, what the agent prefers most is what it would like to obtain most. Use the wikipedia item https://en.wikipedia.org/wiki/Intelligent_agent to identify two key differences between a goal-based and a utility-based agent.

**Exercise 1.5.4**

*Machine learning* is also an important area in artificial intelligence. Mention two environments where the ability to learn would be useful for an agent and explain why you think so.

**Exercise 1.5.5**

Read the wikipedia item https://en.wikipedia.org/wiki/Memory about human memory. Discuss the differences between how a cognitive agent and how a human remembers by comparing the three different kinds of human memories (sensory, short-term, and long-term) with the cognitive state of an agent.

# Chapter 2

# A Simple Chat Agent

In this chapter, we write our first *agent program* in the programming language GOAL. We start with a very simple agent that says "Hello, world!". Similar to "Hello world" programs written in other languages, the agent outputs "Hello, world!" in the console. We will introduce the core notions of the programming language. After having created the simple "Hello, world!" agent, we continue with a walkthrough of the most important elements of the GOAL language. We show how the "Hello World" agent can be made to print "Hello, world!" exactly 10 times by using its beliefs to keep track of the number of times it already said hello. 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.

## 2.1 The GOAL Agent Programming Language

GOAL is a *rule-based* programming language. Rules are **condition-action rule**s that enable the agent to select an action. The rule `if true then print("Hello, world!")` selects the action to print "Hello, world!", for example. We briefly introduce the main elements of the language. An agent program consists of a collection of files that each serve a different purpose. First, a **multi-agent system (MAS)** file (`.mas2g`) is needed to launch and run an agent system. The other files correspond with the basic components and abilities identified in the previous chapter. Knowledge representation (KR) files are needed for representing knowledge and to create the initial cognitive state of an agent, i.e., its knowledge, beliefs, and goals. In our case we will use Prolog (`.pl` files). Action specification files (`.act2g`) are needed to inform the agent about the actions it can perform. Finally, **module** files (`.mod2g`) that contain rules are needed to program the event processing and decision-making of an agent.

## 2.2 An "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 file. We assume you have started the GOAL platform and that you know how to create a new MAS project.[1] Create a MAS project with the name `HelloWorld`. This will automatically create a MAS file with a `mas2g` extension. Open this file. You should see a template with an empty **launch policy** section. Complete it to make it look exactly like:

```
launchpolicy{
   launch helloWorldAgent.
}
```

[1] See the User Guide for the GOAL Eclipse plugin [28]. A new MAS file is automatically created when you create a new GOAL project in Eclipse. This file is instantiated with a minimal template which you will need to complete.

The MAS file will complain that the agent referred to is missing. We need to add an **agent definition** to create an agent named `helloWorldAgent` and to inform the system which modules it should use to create the agent. Do so by adding the following code *before* the launch section in the MAS file:

```
define helloWorldAgent as agent {

}
```

An agent definition specifies a **name** for the agent and consists of a definition section that specifies which module files should be used to create the agent. In order to complete the agent definition above, we need to add at least one **use clause**. A use clause in an agent definition must refer to a module file that should be used to create the agent. Complete the definition and make it look exactly like:

```
define helloWorldAgent as agent {
   use helloWorld as main module.
}
```

The definition above tells the system to use a module called `helloWorld.mod2g` as **main module** when creating the agent called `helloWorldAgent`. Note that the `mod2g` extension does not need to be specified. The main module of an agent contains the rules for decision-making of that agent. Modules are the basic components that agents are made of. Another way of saying this is that an *agent is a set of modules*. The most important module that an agent uses to decide *what to do next* is called the main module.

We have not created the module file referenced yet and now need to do so. Save the MAS file and create a module file named `helloWorld.mod2g`; you may want to check out the GOAL User Manual on how to do this. If not yet open, open the module file. You should see a module called `helloWorld` with an empty program section. Complete it to make it look exactly like:

```
use dummy as knowledge.
exit=always.

module helloWorld {
   if true then print("Hello, world!").
}
```

The first line above is a so-called **use clause** that indicates that the file `dummy` should be used to initialize the agent's knowledge. The extension of this file is left implicit and is used to infer which KR language will be used by the agent. For example, a `.pl` file extension indicates that Prolog will be used. The second line contains an **exit condition**. The exit condition ***always*** means that the module is immediately exited after executing it, i.e., it will be executed only once.

We are almost ready to run the agent but need to fix reference to the missing `dummy` file. Do so by creating an empty `dummy.pl` file, save it, and, if the file is open, close it. We need some KR file to tell the system which KR language is used. As we do not need any predicates for our very basic first program, we use the empty Prolog file `dummy.pl` to indicate that we want to use Prolog. Now we are ready to launch the agent. Run the agent by launching the MAS using the Run button (if you don't know how, check out the User Manual). Amongst other output, you should see the following in the Console area:

```
Hello, world!
...
'...HelloWorld.mas2g' has been terminated
```

You have successfully created your first agent program! If your agent did not terminate, you may have forgotten to add the exit condition after the KR use clause.

## 2.3 Creating a Multi-Agent System

A MAS file is used to set up a new MAS project and specifies which files are needed for creating the multi-agent system. You can view a MAS file as a *recipe for building and launching a multi-agent system.*

We will now program another agent that is created from a different module file. To get started, within the same project, create a new, second MAS file and name it `HelloWorld10x`.

Any MAS file must have at least one agent definition and a non-empty launch policy section. A launch policy tells the platform *when* to create and launch an agent. We have a single agent definition as we only want to create a single agent. We will call our agent again `helloWorldAgent` but use a different module called `helloWorld10x` as main module. To create this module, copy-paste the `helloWorld.mod2g` file and name it `helloWorld10x.mod2g`. Also add the following agent definition to the MAS file and save the MAS:

```
% Simple Hello World agent that prints "Hello, world!" message 10 times.
define helloWorldAgent as agent {
   use helloWorld10x as main module.
}
```

Note that even if we only have a single agent definition we need to include this definition in a MAS file. The reason is that only a MAS file includes all the relevant information that is needed to build 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 launch policy section consists of a single instruction to **`launch`** our "Hello World" agent. Make sure the launch policy in the MAS file looks as follows.

```
launchpolicy{
   launch helloWorldAgent.
}
```

The name `helloWorldAgent` directly following the **`launch`** command is the name of the agent definition that will be used to create the agent. Agents referenced in a launch policy must have a corresponding agent definition in the same MAS file.

## 2.4 A Cognitive "Hello World" Agent

We will now extend our basic "Hello World" agent. Our aim will be to write a new agent that says "Hello, world!" exactly 10 times. To achieve this we will use a distinguishing feature of cognitive agents in GOAL. Cognitive agents can have **goals** and be programmed so as to make their main purpose in live the realization of these goals. We will use this feature to tell the agent it has a goal to print "Hello, world!" 10 times. There are many ways to do so but the key issue that we need to decide on is how to represent this goal. We will use a simple way for representing the goal and use a counter that keeps track of the number of times the agent has outputted something to the console. To represent the counter we introduce the predicate `nrOfPrintedLines/1` with a single argument. We can use this predicate to create the simple **fact** `nrOfPrintedLines(10)` to represent the agent's goal. In order to be able to use this predicate we need to **declare** it. To this end, now create a new Prolog file `counter` and add the following declaration to it:

```
% Declaration of a predicate for counting the number of printed lines.
:- dynamic nrOfPrintedLines/1.
```

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 module that our previous agent used. **Rule**s 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. Now open and make the following changes in the module file `helloWorld10x.mod2g`:

- Replace the reference `dummy` in the use clause with `counter`; we need `counter` here because we want to use `nrOfPrintedLines/1` and need to declare it before we can use it.
- Remove the exit condition **exit=*always***; we need to remove it because we want to execute the module more than once.
- Replace the rule we copied into the module `helloWorld10x` by the one listed below.

You should now have a module that looks like:

```
use counter as knowledge.

module helloWorld10x {
   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 the initial goal that we specified above. The **goal** operator checks whether the agent has a particular goal. Below we will use the operator **bel** for inspecting the beliefs of an agent. The second part of the rule from **then** on tells the agent to perform the action **print**`("Hello, world!")` if the condition holds.

To keep track of how many times a line has been printed we will initialize the agent's beliefs with the fact `nrOfPrintedLines(0)` and to set its initial goal of printing 10 lines we will initialize the agent's goals with the fact `nrOfPrintedLines(10)`. In order to do so, we will create a new module that we will use to initialize the agent's state. We will write one rule that adds the belief and a second rule that adds the goal to the module. Now create a new module and name it `initCounter` and make sure it looks exactly like:

```
use counter as knowledge.

module initCounter {
   if true then insert( nrOfPrintedLines(0) ).
   if true then adopt( nrOfPrintedLines(10) ).
}
```

As before, because we use the `nrOfPrintedLines` predicate, we declare it by means of a use clause at the beginning of the module. We reuse the `counter` file introduced above for this. The module's program section consists of two rules. The first rule tells the agent to **insert** the fact `nrOfPrintedLines(0)` into its beliefs. The second rule tells the agent to **adopt** a goal `nrOfPrintedLines(10)`. Because we will use this module as an **init** module, the rules will be applied before all other rules. As a result, the belief and goal are inserted into the initial state that is created when the agent has just been launched.

The states of a cognitive 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 the **goal base** consists of things the agent wants to achieve. The other database is called the **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.

To make sure that the agent will actually use the `initCounter` module for initialization we still need to add it to the agent definition. Add a use clause for this module to the agent definition in the `helloWorld10x` MAS file:

```
% Simple Hello World agent that prints "Hello, world!" message 10 times.
define helloWorldAgent as agent {
   use helloWorld10x as main module.
   use initCounter as init module.
}
```

Let's see what happens if we run this version of our agent. Launch the MAS *in debug mode* (check the User Guide), start the agent, pause the agent after a second or so, and inspect the console. What you will see is not exactly what we wanted our agent to do. Instead of performing the **print** action exactly 10 times it does not stop and keeps on printing messages. One reason for this is that we removed the exit condition in the module `helloWorld10x` and this module is used as main module. The main module does not terminate if no exit condition is specified; by default the exit condition of this module is **exit**=***never***. It should no longer come as a surprise therefore that the agent does not terminate itself. Moreover, because the agent did not update its beliefs in order to keep track of how many times it performed the action it is also not able to check that it achieved its goal! Inspect the agent's beliefs by means of the Introspector (check the User Guide) to see that the initial beliefs of the agent have not been changed. You should see that the agent still believes that it printed 0 lines.

**Exercise 2.4.1**

Replace the first rule in the module `initCounter` with a rule that inserts the fact `nrOfPrintedLines(10)`. Will the agent still perform any **print** actions? Check your answer by running the modified agent and inspecting its state using the Introspector. (After finishing this exercise, make sure you undo your changes before you continue.)

## 2.5 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. An agent can respond to events such as the *event of receiving a percept* or *receiving a message*. An agent can also react to the *event that it performed an action* by using a module for event processing. Every time that an agent performs an action it will call this module. The main function of a module that is used for event processing should be to update the agent's 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 a separate module for event processing.* Create a new module called `updateCounter` and add the following use clause and rule to it so that the module looks exactly like:

```
use counter as knowledge.

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

As before, the use clause is needed to declare the predicate `nrOfPrintedLines` and a rule must be added to the module's program section. The **bel** operator in the rule's condition is used to inspect, or *query* in database terms, the agent's current beliefs 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 updated 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. As we noticed, by default a main module *never terminates* but we would like our agent to quit when it has achieved its goal. Therefore, now add an exit condition to the module `helloWorld10x` to quit the module when the agent has no more goals after the use clause in the module:

```
exit = nogoals.
```

Finally, we need to add the module `updateCounter` to the agent definition so the agent will use it. We will add another use clause that tells the agent to use it as an **event** module. Because the agent will not receive any other events, the event module will be triggered only by an action that the agent performs. We exploit this to count the number of times the **print** action has been performed. Add a third use clause to the agent definition in the `HelloWorld10x.mas2g` file and make sure it looks like:

```
% Simple Hello World agent that prints "Hello, world!" message 10 times.
define helloWorldAgent as agent {
   use helloWorld10x as main module.
   use updateCounter as event module.
   use initCounter as init module.
}

launchpolicy{
   launch helloWorldAgent.
}
```

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

Some basic magic has happened. You should have seen that the agent prints "Hello, world!" exactly ten times and then terminates. Inspect the agent's goal base this time (you should use Debug Mode to be able to do this). 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 and is removed.

**Exercise 2.5.1**

What happens if we change the number of lines that the agent wants to print in the `initCounter` module? For example, replace `nrOfPrintedLines(10)` with `nrOfPrintedLines(5)` and run the MAS again. You should see now that the agent does not print anything. The reason is that the rule in the `helloWorld10x` module is not applicable any more. 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.

## 2.6 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*. GOAL supports loading environments automatically if they implement a well-defined environment interface called EIS [3, 4]. 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. A diverse set of environments that implement this interface can be found on eishub on github.

An **environment** can be added to a multi-agent system by adding a use clause for that environment *at the start of a MAS file*. All we need to do is to indicate where the environment can be found. In our case, we want to add the `HelloWorldEnvironment` environment, which you can find here. Download and copy this `jar` file to the `HelloWorld` project folder. Copying the `jar` file to the same folder as the MAS file that you created will make sure that the environment file can be found. Before we proceed, let's copy the `HelloWorld10x.mas2g` and give this file the name `ScriptAgent.mas2g`. Add the following use clause at the start of this new MAS file:

```
use "HelloWorldEnvironment-1.1.0.jar" as environment.
```

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 launch policy section. To connect our agent to the entity made available by the `HelloWorldEnvironment` we only need to make a slight change to our earlier launch policy. Replace the launch instruction in the launch policy with the following launch rule:

```
launchpolicy{
   when * launch helloWorldAgent.
}
```

The launch rule above adds a condition in front of the launch instruction. The rule is triggered whenever an entity becomes available, as `*` matches with any entity. When triggered, the agent `helloWorldAgent` is created using the agent definition, and connected to the entity.

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 can let the agent know it can perform this action by specifying it in an **action specification** file. Create an action specification file `printText.act2g` and add the action specification below:

```
use dummy as knowledge.

% 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.
define printText(Text) with
   pre{ true }
   post{ true }
```

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 also 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.)

The only thing that remains is to start using the new service. To do so, do the following:

- copy and rename the `helloWorld10x` module to `printScript`,
- replace the reference to `helloWorld10x` in the `ScriptAgent` MAS file with `printScript`,
- replace `helloWorld` with `printScript` in the `printScript` module file,
- replace the **print** action in the module's rule with `printText`, and
- add a use clause for the action specification file we just created (see below).

The use clause is needed to declare the action `printText` used in the rule. The module `printScript` should look like:[2]

```
use counter as knowledge.
use printText as actionspec.
exit = nogoals.

module printScript {
   if goal( nrOfPrintedLines(10) ) then printText("Hello, world!").
}
```

Launch the MAS again (in debug mode) 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. Inspect the Introspector of the “Hello World" agent and select the Percept tab. You should see the following:

```
lastPrintedText('Hello, world!')
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 going wrong, however. Instead of printing “Hello, world!" 10 times the environment informed the agent it

[2] If you completed Exercise 2.5 instead of 10 your rule may have a variable.

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 `updateCounter` module used for event processing 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 `updateCounter` 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. This can be done by the **`percept`** operator which like the **`bel`** operator inspects the agent's state but inspects the agent's percept base instead of its belief base. Replace the old rule and add the following rule to the `updateCounter` module (or use a renamed copy of the module and rename references where needed):

```
if percept( printedText(NewCount) ), bel( 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.

**Exercise 2.6.1**

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 `initCounter` module used to initialize the agent before. To this end do the following:

- Remove the rule that inserts the beliefs from the `initCounter` module.
- Add a rule for processing the initial percept. Copy the rule from the `updateCounter` module and modify it as needed. (As there is no initial belief yet, remove the **`bel`** condition from the rule and also remove the **`delete`** action.)

## 2.7 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 of the SWI Prolog language. 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. Create a new Prolog file `script.pl` and add the following script facts to the file:

```
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!").
```

We will use these facts as beliefs in the `printScript` module. Because facts implicitly declare a predicate we do not need to add an explicit declaration for the `script/2` predicate. Now add the following use clause to the `printScript` module:

```
use script as beliefs.
```

We need one more modification before we can use the script stored in the belief base. We need to change the following rule for printing text in the module as well.

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

This rule does not use the script text in the Prolog program `script`. 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 **`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 `printScript` 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. The script has been 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 `initCounter` module if you completed the first exercise in Section 2.5; otherwise you will also need to modify the `10` that occurs in the rule in the `printScript` module. The next exercise asks you to resolve the issue in a more principled manner.

**Exercise 2.7.1**

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 the `initCounter` module that our script agent uses in this exercise and provide it with an alternative method to set the goal to print the entire script.

- First, remove the rule that sets the goal in the `initCounter` module. The idea is to compute a goal instead to print the entire script and to adopt that goal.
- Move the use clause for the `script` from the `printScript` to the `initCounter` module. Add a declaration for the `script/2` predicate to the Prolog file `counter.pl` (and rename the file and references if you like).
- Add the rule template `if bel(...) then adopt(nrOfPrintedLines(Max)).` to the program section of the `initCounter` module. (Don't forget the trailing '.'!)
- 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 [45] 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 3

# Inspecting Cognitive States

As we saw in Chapter 1, three of the five components of a **cognitive state** consists of the agent's **knowledge**, **beliefs**, and **goals**. An agent has a cognitive state to reason about and keep track of the current state of the environment it interacts with and the state it wants the environment to be in. In order to do so, an agent needs to use a **knowledge representation** (KR) language. GOAL supports multiple knowledge representations.[1] An agent can use at most one at the same time, however, and we need to make a choice. In this chapter we will look at how we can use **Prolog** for representing an agent's knowledge, beliefs, and goals. We will also introduce the state operators **`bel`** and **`goal`** for inspecting or querying the cognitive state of an agent and explain how to use these for writing cognitive state queries.

## 3.1 Representing Knowledge, Beliefs and Goals

One of the first steps in developing and writing a GOAL agent is to design and write the knowledge, beliefs and goals that an agent needs. An important task is to select suitable predicates or concepts for representing the agent's environment. As we will see in Chapter 6, the percepts that are received from an environment can provide a useful starting point. It is important to get the representation of the agent's knowledge, beliefs and goals right, as the rules introduced in Chapter 5 that an agent uses for decision-making depend on it. Moreover, as we will see, the action specifications introduced in Chapter 4 also depend on it. Although GOAL is not married to any particular knowledge representation language, here we will use SWI Prolog [44].

### 3.1.1 Example Environment: The Blocks World

As a running example in this chapter, we will use one of the most famous environments in artificial intelligence known as the **Blocks World** [38, 42, 47]. Admittedly, this is a toy domain, but even Blocks World problems are surprisingly hard to solve efficiently. In its most simple (and most common) form, in the Blocks World an agent can move and stack cube-shaped blocks on a table by controlling a robot gripper. One important fact is that the robot gripper is limited to holding one block at a time and cannot hold a stack of blocks. This is important information that we will need when we specify the gripper action in the Chapter 4. For now, we will focus on how to specify the configuration of blocks on the table using Prolog. A block can be directly on top of at most one other block. That is, a block is part of a stack and either located on top of a single other block, or it is sitting directly on top of the table. These are some of the basic "laws" of the Blocks World. We add one assumption about the table in the Blocks World: we will assume throughout that the table is large enough to be able to place all blocks directly on the table. This is another

[1] See https://github.com/goalhub/krTools for KR languages that are supported.

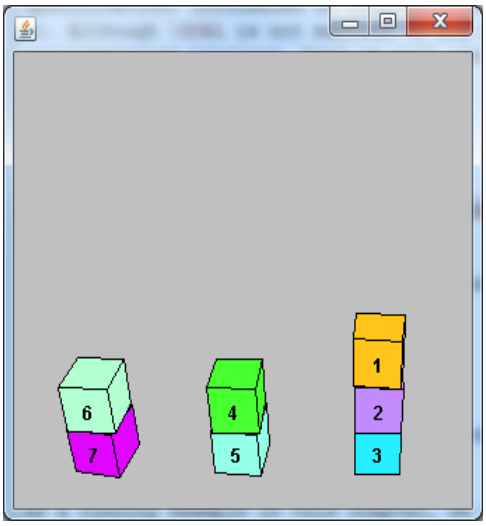


Figure 3.1: Initial Blocks World State

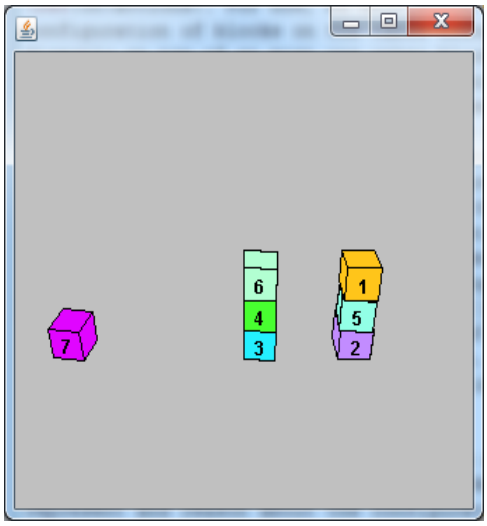


Figure 3.2: Goal State

basic "law" of the version of the Blocks World that we will use.[2] Figures 3.1 and 3.2 illustrate an example Blocks World problem. Such a problem consists of an initial state or configuration of blocks and a goal state. It is up to the agent to achieve the goal state by moving blocks in the initial state and successor states.

A cognitive agent with the task to solve a Blocks World problem, i.e., to achieve the goal state in the Blocks World, needs to be able to represent and reason about the configuration of blocks. We need to introduce Prolog predicates for specifying a configuration of blocks. As a rule of thumb, it is a good idea to introduce predicates that correspond with the most *basic concepts* in a domain. More complicated concepts then may be *defined* in terms of the more basic concepts. We will use this strategy here as well.[3] 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 binary predicate `on/2`:

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

We use the predicate `on(X, Y)` to express that *block* `X` *is (directly) on top of* `Y`. This is our informal definition of the predicate `on(X, Y)`. For example, `on(b1,b2)` is used to represent the fact that block `1` is on block `2` and `on(b2,table)` is used to represent that block `2` is on the table. In Figure 3.1, `on(b1,b2)` is the case and `on(b2,table)` and `on(b1,b3)` do not hold. `on(b1,b3)` does not hold because `on(X, Y)` only holds if a block `X` is *directly* on top of `Y` and there is no other block in between block `X` and `Y`. It is important that only *blocks* can be on top of something else in the Blocks World. This implies that whenever `on(X, Y)` holds `X` must be a block. `Y` however does not need to be a block but may also be the table. Finally, there can be *at most one* block on top of another block. These rules are specific to our version of the Blocks World. They cannot be enforced automatically. It is therefore important to realize that it is up to the agent programmer to stick to these rules and to use the predicate `on` in "the right way".

So far we have been using **name**s for blocks. We have said that `on(b2,table)` means that block `2` is on the `table`. Implicitly, we have made a *unique names assumption* here: We have

[2] See [16]. For other, somewhat more realistic presentations of this domain that consider, e.g., limited table size, and varying sizes of blocks, see [20].

[3] It can be shown that the basic concept *above* is sufficient in a precise sense to completely describe arbitrary, possibly infinite configurations of the Blocks World; that is not to say that everything there is to say about a Blocks World configuration can be expressed using only the *above* predicate [16]. Here we follow tradition and introduce the predicates *on* and *clear* to represent Blocks World configurations.

assumed that all blocks have *unique names* that we can use to refer to each block. This is very useful because it allows us to distinguish one block from another just by their names, which simplifies our task of programming an agent for the Blocks World. This is much simpler than having to identify a block, for example, by means of its position with respect to other blocks. We will also make the *domain closure assumption*: There are names *for all* objects in the Blocks World. That means that all blocks have a name; we use `table` to refer to the table in the Blocks World. Together, the unique name and domain closure assumption imply that *all* objects have a *unique* name.

**Blocks World Problem** A Blocks World state is a configuration of blocks and can be identified with a set of facts $F$ of the form `on(X,Y)`. A state $F$ must be consistent with the "laws" of the Blocks World, i.e., at most one block is directly on top of another, etc. If $B$ is the set of blocks, we can differentiate *incomplete or partial* states from *complete* states. A state $F$ that contains a fact `on(X,Y)` for each block `X` $\in B$ is called *complete*, otherwise it is a *partial* state. Using this notion of Blocks World state, we can now also formally define a *Blocks World problem*. A Blocks World problem is a pair $\langle I, G \rangle$ where $I$ is a (complete) initial state and $G$ is a (possibly partial) goal state.

**Domain Knowledge** To make explicit which blocks are present in the Blocks World we introduce the unary predicate `block/1`. Recall that a block must be on something and if we have `on(X,Y)` we must have that `X` is a block. We can use this **domain knowledge** to define the `block` predicate by the following Prolog rule:

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

In order for this definition to work we must assume that all facts about which block sits on top of another (or the table) are known to the agent.

The `block` predicate can be used to define another useful predicate `clear/1`. We will use `clear(X)` to mean that a block can be moved on top of `X`. This informal reading allows us to apply the predicate to blocks as well as the table. A block can be moved on top of another block `X` only if there is no other block sitting on top of `X`. That is, a block without any block on top of it is clear. The table is always clear in our sense, as in our version of the Blocks World the table always has room to place a block. We can capture this domain knowledge by the following Prolog clauses:

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

In order to deal with the limitations of the gripper it is important to be able to conclude that a block is clear. A block can only be moved if it is clear, i.e., there is no other block sitting on top of it. The `clear(X)` predicate can also be used to check that a block can be moved on top of `X`.

**Floundering** The definition of the `clear` predicate can be used to illustrate various important things that you need to consider when writing Prolog definitions. One issue is that variables in a negated fact need to be sufficiently instantiated. Reversing the conjuncts in the definition of the `clear` predicate, for example, would cause a problem. If the definition

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

would be used, the query `clear(X)` without `X` being instantiated would always fail! In a Blocks World with a finite number $N > 0$ of blocks, however, there must at least be one block that is clear. What goes wrong is that the negation operator is applied to the non-ground literal `on(_, X)`. As the example illustrates this is not safe as the negation operator does not bind any variables.

A Prolog program that applies negation to a non-ground literal is said to **flounder**. As a rule of thumb, to avoid the floundering problem, make sure that each variable in the body of a rule first appears in a positive literal. In our definition of the `clear` predicate, we have made sure that the variable `X` is bound first by the positive literal `block(X)` before the variable is used in the second, negative literal **not** `(on(_, X))`. Moreover, in our definition we have used a *don't care* variable _ instead of the variable `Y` above. This not only indicates that we do not care about how this variable is instantiated, but also makes sure that no bindings for this variable are passed on.

**Closed World Assumption** Another important thing to realize is that we can only be sure that the definition of `clear(X)` correctly infers that a block is clear if the Blocks World state is *completely* represented. Note that the body `block(X), not(on(_, X))` of the rule defining the predicate `clear` succeeds for every block `X` for which it cannot be shown that `on(_, X)` holds. This is because the negation in Prolog is *negation by (finite) failure*. Only in case a search for a proof of `on(_, X)` fails Prolog's negation succeeds. Absence of information thus allows us to conclude that a block is clear. But this is only correct if that information is not simply missing, i.e., if all facts about a Blocks World configuration are known.

Another way to make this point is saying 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. In other words, the state represented must be complete.

A related issue is that an agent can only keep track of the complete state of its environment if that state is *fully observable*. The lesson here is that domain knowledge needs to be carefully designed and basic assumptions about, e.g., the observability of an environment need to be taken into account when representing domain knowledge. In our running example, if an agent would not be able to keep track of the complete configuration of blocks in the Blocks World, the rule for `clear(X)` would need to be modified.

**Conceptual Knowledge** It will be useful to be able to identify the position of a block in a stack of blocks. This allows us, for example, to check which blocks are sitting below another block and whether this is what we want (or, need to solve a Blocks World problem). To this end, we introduce and define the concept of a *tower* for the Blocks World as follows:

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

The rules for the `tower/1` predicate recursively define when a list `[X|T]` of blocks is a tower. The first rule says that `[X]` is a tower if it is on the table, i.e., `on(X,table)` holds. It requires that the basis of any tower is grounded on the table. The second rule says that whenever `[Y|T]` is a tower, `[X,Y|T]` that extends this tower with a block `X` sitting on top of `Y` also is a tower.

Note that if `[X|T]` according to this definition is a tower, this does *not* mean that block `X` is clear. As a consequence, any stack of blocks that is part of a larger stack is a tower. We can, for example, derive `tower([b2,b3])` from the facts that represent the initial state of Figure 3.1. Perhaps this definition does not completely match our common sense notion of a tower, but for our purposes the definition is sufficient.

Although there are other useful concepts that we could introduce (see for an example the definition of the predicate `above/2` in the Exercise Section), the predicates that we have introduced are sufficient for our purposes of programming a strategy for solving Blocks World problems.

### 3.1.2 Creating a Cognitive State: Use Clauses

Three of the five components of a cognitive state of a GOAL agent consists of the agent's knowledge, beliefs and goals (see also Figure 1.3). An agent inspects and modifies this state at runtime similar to a Java method which operates on the state of an object. Agent programming therefore can also be viewed as *programming with cognitive states*. The **informational state** of the agent consists of the state components that concern the *current* state of the environment: the **knowledge base** and **belief base** of an agent. Knowing and believing are also called **informational attitudes** of an agent. Other informational attitudes are, for example, expecting and assuming. The **motivational state** of the agent consists of the state components that concern the *desired* state of the environment: the **goal base** of an agent. Wanting is a **motivational attitude**. Other motivational attitudes are, for example, intending and hoping.

**Initializing a Cognitive State** The content of an initial cognitive state is specified in the knowledge representation language that the agent uses. In our case, as we are using Prolog, the content of a state is specified as a Prolog program. The initial knowledge, beliefs, and goals of the agent's state therefore are specified in Prolog files. These files can be imported in the agent's state by means of **use clause**s at the start of a module file that is part of the agent program. A use clause has the form

**use** *id* **as** *usecase*.

Here, *id* is the name of a KR file (without extension), i.e. the name of a Prolog file in our case, and *usecase* either is **knowledge**, **beliefs**, or **goals**. We have seen already some examples in Chapter 1 where `counter.pl` was imported **as knowledge** and `script.pl` was imported **as beliefs**. Note that the file's extension should not be included in a use clause. We simply write, e.g.,

```
use counter as knowledge.
```

Also note that a use clause must be followed by a dot. Use clauses are not the only way that an agent can be initialized. In Chapter 1 we saw that rules in a module can also be used (as **init** module) for initializing the state of an agent. We used a module there to initialize the goal base of the agent.

**Knowledge and Beliefs** The difference between knowledge and beliefs in GOAL is that knowledge is *static and cannot change* at runtime and belief is *dynamic and changes* at runtime. As a result, the knowledge base of an agent cannot be used to keep track of the part of the state of the environment that changes over time. Instead the belief base of the agent needs to be used for keeping track of the current state of the environment and should contain facts that may change over time. If we want to specify the initial components of these bases, we need to use different files and a separate use clause for importing knowledge and for beliefs. A file that specifies the knowledge or the initial beliefs of an agent must be a Prolog program and respect the usual syntax of Prolog. It is possible to use most of the built-in operators of the Prolog system that is used (in our case, SWI Prolog [44]).

The difference between knowledge and beliefs explained above provides a basic guideline for deciding what to include in which file. Because rules are used for specifying domain knowledge and for defining conceptual knowledge, and rules cannot be changed at runtime as will be discussed in more detail in Chapter 4, rules are best included in a Prolog file imported as knowledge. This means that these rules will be included in the agent's knowledge base. As *facts* such as `on(b1,b2)` can be modified at runtime, these typically are included in a file that specifies an initial belief state for the agent and are best included in a Prolog file imported as beliefs. The exception here are facts that do not change and represent domain knowledge. A fact in the Blocks World, for example, that is best included in the agent's knowledge base is `clear(table)` because it is always true.

It is important that all predicates in a Prolog file are **declared** so that the agent knows which predicates it can use in its knowledge, beliefs, goals, decision rules, and action specifications.

In (SWI) Prolog, a predicate can be declared either by defining it by means of a Prolog rule or by explicitly declaring it by means of the `:- dynamic` directive. For example, the statement `:- dynamic on/2.` (dynamically) declares the predicate `on/2` so that it can be used in the rules that we defined above. Multiple predicates can be declared at once, for example, `:- dynamic on/2, timer/1.` declares the two predicates `on/2` and `timer/1`. Note that facts like `on(b1,b2)` in a Prolog program also (implicitly) declare the predicate `on/2`. As a rule, any predicate in a Prolog program that is used in the body of a Prolog rule but not defined, i.e., does not occur in the head of a Prolog rule or as a fact, needs to be explicitly declared using the `:- dynamic` directive.

Based on this discussion, we include the domain and conceptual knowledge introduced above as well as the fact `clear(table)` in a single Prolog file that we will import later as knowledge:

```
% Declaration of the on/2 predicate.
:- dynamic on/2.

% only blocks can be on top of another object.
block(X) :- on(X, _).

% the table is always clear.
clear(table).
% a block is clear if nothing is on top of it.
clear(X) :- block(X), not( on(_, X) ).

% the tower predicate holds for any stack of blocks that sits on the table.
tower([X]) :- on(X, table).
tower([X, Y| T]) :- on(X, Y), tower([Y| T]).
```

Because we will use this Prolog program later, create a new file and name it `bwknowledge.pl`. We can import this file by including the use clause **`use`** `bwknowledge` **`as knowledge`** in a module file. Note that the extension is not specified; it is retrieved by the system and used to infer the knowledge representation language that is used automatically (for KR known to the platform; see also footnote 1).

As the initial Blocks World state consists of facts that may change we include these in a separate file that we will import as beliefs. The initial state of Figure 3.1 is represented by the following facts:

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

Note that the `on`-facts implicitly declare the `on/2` predicate and we do not need to add an explicit declaration. Create this file and name it `bwbeliefs.pl`. We can import this file then later by including the use clause **`use`** `bwbeliefs` **`as beliefs`** in a module file. The use case keyword **`beliefs`** here indicates that the file should be used for initializing the agent's belief base. Also note that because we are using a Prolog database we cannot include *negative* literals (see the discussion above about the Closed World Assumption).

**Perception** Although it is possible to introduce initial facts into the belief base of an agent by means of a use clause that imports a Prolog file as beliefs, it is not very common to initialize the beliefs of an agent in this way. There are two reasons for this. First, it is a rather inflexible method for initializing an agent's beliefs. If we want to initialize an agent with a different set of facts, we would need a different file for each set. Second, and more importantly, if the agent is connected to an environment, there is an alternative for initializing that agent's beliefs. Usually the initial state of the environment is not known in advance and the initial beliefs need to be obtained by *perceiving* the initial state of the environment. Initial facts about the current state of the environment thus typically are collected through the sensors available to the agent. To this end, a module used as **`event`** module can be used; we saw an example of that in Chapter 1 already. For that reason, there may not be a need for a use clause for beliefs in an agent program.

**Goals** In our running example, the Blocks World, we saw that the `on/2` predicate can be used to specify an initial state. It can also be used to specify the goal state of the agent. For our specific example domain, illustrated in Figure 3.2, we can create a Prolog file that represents the goal state by listing the facts that hold in that state as follows:

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

Create this file and name it `bwgoals.pl`. We can imported this file later by including the use clause **`use`** `bwgoals` **`as goals`** in a module file. The use case keyword **`goals`** here indicates that the file should be used for initializing the agent's goal base. Intuitively, what is wanted should not be the case yet. The main reason is that a *rational* agent should not spent its resources on trying to realize something that is already true. It is easy to verify that the clause above represents the goal state depicted in Figure 3.2 and is different from the initial state.

There is one very important difference between the specification of an initial goal and of an initial belief state. We have used Prolog's **conjunction operator `,`** in the specification of a goal above instead of the dot `.` used after each fact in the specification of the initial beliefs. A goal thus is *explicitly* represented as a conjunction of facts whereas beliefs are not. Strictly speaking this means that the file with the goal above is *not* a Prolog file. Importing it directly in Prolog will produce an error:

```
ERROR: .../bwgoals.pl:1:
       Full stop in clause-body?  Cannot redefine ,/2
```

There is a good reason, however, why we use the conjunction operator for specifying a goal. The reason is that each of the facts that together represent the goal state need to be achieved *simultaneously*. If these facts would have been included as clauses separated by a dot we would get something different. The specification of the goals of an agent below thus is *not* the same as the specification of the goal state above:

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

The main difference concerns the number of goals. The conjunctive goal counts as a *single* goal, whereas the 7 atoms separated by a dot would count as 7 distinct, independent goals. It is, for example, very different to have two separate goals `on(b1,b2)` and `on(b2,b3)` instead of a single conjunctive goal `on(b1,b2), on(b2,b3)`. The two separate goals do not pose any restrictions on the order of achieving them. But combining these goals into a single conjunctive goal requires an agent to achieve both (sub)goals *simultaneously*. This restricts the actions that the agent can perform to achieve the goal. In the first case, where we have two separate goals, the agent may, for example, put `b1` on top of `b2`, remove `b1` again from `b2`, and put `b2` on top of `b3`. Obviously, that would not achieve a state where `b1` is on top of `b2` which is on top of `b3` at the same time.[4] As separate goals may be achieved at different times it also makes sense that multiple individual goals are not consistent when combined. This is not the case for beliefs (think about it, inconsistent beliefs cause problems). For example, an agent might have the two goals `on(b1,b2)` and `on(b2,b1)`. Obviously these cannot be achieved simultaneously, but they can be achieved one after the other. A goal such as `on(b1,b2)` should be read as *at some time in the future* `on(b1,b2)` is true. The temporal operator is left implicit in what we actually write in a program. The informal reading explains the difference between beliefs and goals.

Summarizing, it is very important to realize that there is a difference between using the dot or the comma when specifying one or more goals. The dot operator separates goals where the

[4]These observations are related to the famous *Sussman anomaly*. Early planners were not able to solve simple Blocks World problems because they constructed plans for subgoals (parts of the larger goal) that could not be combined into a plan to achieve the main goal. The Sussman anomaly provides an example of a Blocks World problem that such planners could not solve, see e.g. [19].

comma indicates a conjunction of sub-goals of a single goal. The sub-goals of a larger goal need to be achieved simultaneously. As a result, the content of a file specifying an agent's initial goals is not a Prolog program because the comma separator `,` is used to construct conjunctive goals.

**Goals Must Be Ground** We briefly discuss some technical aspects of specifying a goal. The most important one is that a goal must be ground. That is, it should not contain any free variables. The reason for this is that it is not clear what the meaning of a goal with a free variable is. For example, what would a goal `on(X,table)` mean? In Prolog, the reading would depend on whether `on(X,table)` is a clause (or rule with empty body) or a query. In the former case, a goal `on(X,table)` would be implicitly universally quantified and the meaning would be that *everything* should be on the table. This poses a problem as it is not clear how an agent can compute *everything*. Taking `on(X,table)` as a query would require an existential reading. The goal would mean that *something* should be on the table. Perhaps we can make sense of the latter reading but we cannot choose, however, since we want to be able to *store* `on(X,table)` in a Prolog database *and* we want to use `on(X,table)` as a Prolog *query* to verify whether it is believed by the agent (to check whether the goal has been achieved). Storing the fact in a Prolog database means it is implicitly universally quantified whereas the use of the fact as a query assumes existential quantification. As we cannot have it both ways, we require goals to be ground. Finally, also note that, just as is the case for beliefs, we cannot use *negative* literals for specifying a goal because Prolog does not support storing negative literals in a database (see the discussion on the Closed World Assumption). We thus cannot specify explicitly that something should *not* hold as part of a goal in Prolog. Another knowledge representation language than Prolog would be needed to do so.

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

```
knowledge:
   :-dynamic on/2.
   block(X) :- on(X, _).
   clear(table).
   clear(X) :- block(X), not( on(_, X) ).
   tower([X]) :- on(X,table).
   tower([X,Y|T]) :- on(X,Y), tower([Y|T]).
beliefs:
   on(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).
```

Figure 3.3: The Initial Cognitive State of an Agent That Represents Figures 3.1 and 3.2

**Example Cognitive State** By putting everything together, we can construct the initial cognitive state of our Blocks World agent. In Figure 3.3, we have indicated the various bases by means of their corresponding use case keywords.

## 3.2 Inspecting an Agent's Cognitive State

Agents that derive their choice of action from their beliefs and goals need the ability to inspect their cognitive state. In GOAL, **state queries** provide the means to do so. These queries are conditions used in rules to determine which actions the agent will perform (see Chapter 5).

### 3.2.1 Basic State Queries

Basic state queries are used to inspect either the belief or goal base. The operator **bel** is used to inspect the belief base. The **goal** operator is used to inspect the goal base. We will add other operators later in Chapter 6 and 8 for inspecting an agent's percept and message base. Note that there is no operator for inspecting the knowledge base. This is because the knowledge of an agent is queried in combination with the agent's beliefs or with the agent's goals. We will explain how this works below. A basic **belief query** is of the form **bel**(*qry*). A basic **goal query** is of the form **goal**(*qry*). In both types of queries, *qry* must be a query in the knowledge representation language that is used. As we are using Prolog in this chapter, *qry* thus needs to be a Prolog query.[5]

**Belief Queries** Informally, **bel**(*qry*) means that the the agent believes *qry*. An agent believes that *qry* holds if it follows from *both* its knowledge *and* its beliefs. In order to evaluate if this is the case, the knowledge and current beliefs of the agent are treated as a single database and *qry* is evaluated on this database. In other words, a query **bel**(*qry*) holds whenever *qry* can be derived from the content of the agent's belief base *in combination with* the content of the knowledge base. For example, the query **bel**(`clear(a)`) succeeds in the initial state that represents the Blocks World of Figure 3.1. To see this, note that there is no fact of the form `on(X,b1)` and apply the definitions of the `clear/1` and `block/1` predicates. In this state, the agent thus believes that block `a` is clear. Try to find out yourself whether **bel**(`tower([b1,b2,b3])`) holds in the initial cognitive state of Figure 3.3.

**Goal Queries** Informally, **goal**(*qry*) means that the agent has a goal *qry*. An agent has a goal *qry* if it follows from *one of the goals in its goal base in combination with its knowledge*. As with beliefs, knowledge and goals are combined in order to evaluate a goal query. There is a subtle difference, however: Only *one* of the agent's goals is combined with (all of) the agent's knowledge. In other words, a query **goal**(*qry*) holds whenever *qry* can be derived from a *single* goal of the agent that is combined with the contents of the agent's knowledge base.[6] For example, the query **goal**(`tower([b5,b2])`) succeeds in the cognitive state represented in Figure 3.3. To see this, note that the the agent can derive the (sub)goals `on(b2,table)` and `on(b5,b2)` from its single goal and apply the definition of the `tower/2` predicate in the knowledge base.

The fact that an agent can reason about current towers as well as desired towers illustrates the benefits of a separate knowledge base that can be combined with both the agent's beliefs as well as it's goals. This also motivates the rule of thumb discussed above that rules should be included in a specification of the agent's knowledge.

[5] Note that a Prolog query `qry` is not prefixed with `?-` here. That is, we simply write e.g. **bel**(`p(X,Y)`) and do not write **bel**(`?-p(X,Y)`).

[6] This reading differs from that provided in [9] where the goal operator is used to denote *achievement goals*. The difference is that a goal is an achievement goal only if the agent *does not believe that* $\varphi$. The goal operator `goal` introduced here is a more basic operator. As we will see, it can be used in combination with the belief operator **bel** to *define* achievement goals.

### 3.2.2 Cognitive State Queries

Cognitive state queries are constructed from the basic state queries **bel**(*qry*) and **goal**(*qry*). These basic queries can be negated. **not**(**bel**(*qry*)) and **not**(**goal**(*qry*)) are cognitive state queries. Basic queries and their negations are also called *state literals*. Finally, state literals can be combined in a conjunction to form a cognitive state query. For example,

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

is a cognitive state query. It says that the agent wants block `b` on the table, i.e., `on(b,table)` is a goal, and does not believe that this is currently true.

Basic queries or literals cannot be combined by disjunction. It is not possible, for example, to write **goal**(`on(b,table)`)`;` **not**(**bel**(`on(b,table)`)), where `;` denotes disjunction in Prolog. This does not pose a real limitation on what can be programmed, however. State queries are used as conditions in the rules of an agent program. Intuitively, the use of disjunction would allow to program a rule to select an action if either one state query or another holds. Such a condition would provide two different reasons for selecting the action. Instead of one rule that combines both queries you can write multiple rules that select the action in either of these cases.

**Achievement Goals** Combinations of belief and goal queries are useful for expressing several other intuitive concepts. A goal query **goal**(*qry*) only says that *qry* follows from one of the goals that the agent has. This does not mean, however, that the agent believes that nothing yet has been achieved to realize *qry*. An agent, for example, may have a goal `on(b4,b3), on(b3,table)`, as in Figure 3.2, but may also believe that part of this goal, e.g., `on(b3,table)` in our example, has already been achieved. The parts of a goal that still need to be achieved are called **achievement goals**. As achievement goals are important reasons for choosing to perform an action, it is useful to introduce a special operator **a-goal**(*qry*) to identify these goals. This operator can be used instead and simply is an abbreviation for state queries of the form **goal**(*qry*), **not**(**bel**(*qry*)).[7]

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

Achievement goal queries are useful because they provide good reasons for an agent to do something. In the Blocks World, for example, an agent needs to do something if a block is *misplaced*. That a block is misplaced means that its current position is different from the position that the agent wants it to be in. As the position on the table is not important, only the position in the stack of blocks that the agent wants the block to be in is relevant. To obtain this position from the agent's state, we can write a goal query that retrieves the desired stack or tower that the block is in using the `tower/2` predicate. We can use the following query: **goal**(`tower([X|T])`) with `X` a block and `T` a (possibly empty) tower. In order to conclude that block `X` is misplaced we additionally need to check that the desired tower has not already been realized. We can use the following query for this: **not**(**bel**(`tower([X|T]))`).[8] As the Prolog query in the goal and belief literals are the same, we can use the **a-goal** operator to combine both in a single query **a-goal**(`tower([X|T])`) to express that block `X` is misplaced. Being able to conclude that a block is misplaced is important for defining a strategy to solve a Blocks World problem because only misplaced blocks have to be moved.

The state query **a-goal**(`tower([X,Y|T])`), **bel**(`tower([Y|T])`) will also be useful for programming an agent that solves Blocks World problems. Note that the first conjunct says that block `X` is misplaced. It also says that the agent wants `tower([Y|T])`. The second part says that

[7] See [23] for a discussion of this definition.

[8] To be precise, here the common sense difference between *knowledge* and *belief* is important. Normally we would say that something is misplaced only if we *know* that it is in a different position than we want it to be. It is not good enough if we only believe it is not with a chance of being wrong. If, in fact, the block is in the desired position, in ordinary language, we would say that the block is *believed to be misplaced* but that in fact it is not. The difference is not important, of course, if we may assume that the agent's beliefs are true.

the agent believes that `tower([Y|T])`. It follows that if the state query succeeds, only block `X` still needs to be put in place. In other words, if the agent can move `X` on top of `Y`, the agent can make a *constructive move* and bring a solution one step closer. Making a constructive move, moreover, implies that the block moved will never have to be moved again. Making such a move thus always is a good idea. The only missing piece of information that we still need to be able to decide whether such a move is possible is when the move action can be performed. For this we need to evaluate the precondition of the move action. In Chapter 4 we will look at preconditions in more detail which we then will combine with the state query we just discussed in Chapter 5 to program a strategy that solves Blocks World problems.

**Sub-Goals Achieved** Part of deciding whether a constructive move can be made consists in checking whether part of the tower that the agent wants is already in place. In other words, it consists of verifying that the query **`goal`**`(tower([Y|T])), `**`bel`**`(tower([Y|T]))` succeeds. If this query holds, it means that the **sub-goal** `tower([Y|T])` has been realized. More generally, we call *qry* in a state query **`goal`**`(`*qry*`), `**`bel`**`(`*qry*`)` a **goal achieved**. We introduce the operator **`goal-a`**`(`*qry*`)` as an abbreviation for this query to denote such goals.

$$\textbf{goal-a}(qry) \stackrel{df}{=} \textbf{goal}(qry), \textbf{bel}(qry)$$

**Checking Whether an Agent Has Any Goals** The **`goal`** operator is special in the sense that it evaluates a KR query *qry* on only *one* of the goals that the agent has. But what happens if the agent has *no* goals? In that case **`goal`**`(`*qry*`)` fails for all queries *qry*. This can be exploited to check whether the agent has any goals at all. Note that the query **`goal`**`(`**`true`**`)` would succeed if the agent has a goal no matter what the goal is. By negating this query we obtain a method for evaluating whether the agent has a goal or not. The query **`not`**`(`**`goal`**`(`**`true`**`))` succeeds only if the agent has no goals, i.e., if the goal base of the agent is empty.

## 3.3 Notes

The components of an agent's cognitive state that we discussed in this chapter consist of the knowledge, beliefs, and goals of an agent *about the environment* it interacts with. These states enable the agent to represent its environment but does not provide support for representing the beliefs or goals of other agents. In particular, there is no support for representing the cognitive state of another agent. That would require the ability to model another agent's state and the ability to represent another agent's beliefs and goals. Agents that can have models of other agent's cognitive states are second- or even higher-order intentional systems. In Chapter 8 we will discuss models for representing the states of other agents.

In this Chapter we have assumed Prolog [43, 44] is used by the agent. 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 [13], the Planning Domain Definition Language (PDDL; [19]), or other, similar such languages, or possibly even Bayesian Networks [35]. The only assumption that we will make throughout is that an agent uses a *single* knowledge representation technology to represent its knowledge, beliefs and goals. For some preliminary work on lifting this assumption, we refer the reader to [18].

## 3.4 Exercises

```
knowledge:
  :-dynamic on/2.
  block(b1). block(b2). block(b3). block(b4). block(b5).
  clear(table).
  clear(X) :- block(X), not( on(_, X) ).
  tower([X]) :- on(X, table).
  tower([X,Y|T]) :- on(X, Y), tower([Y|T]).
  above(X,Y) :- on(X,Y).
  above(X,Y) :- on(X,Z), above(Z,Y).
beliefs:
  on(b1,b4). on(b2,table). on(b3,b5). on(b4,b2). on(b5,b1).
goals:
  on(b1,b2), on(b2,b3), on(b4,b5), on(b5,table).
```

Figure 3.4: An Example Cognitive State

### 3.4.1

Consider the cognitive state of Figure 3.4. Which of the following cognitive state queries succeed? Provide all possible instantiations of variables, or otherwise a conjunct that fails.

1. `goal(above(X,b4), tower([b4,X|T]))`
2. `bel(above(b1,X)), goal(above(b1,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

The query `a-goal(tower([X|T]))` can be used to express that block `X` is *misplaced*. A block, however, is only misplaced if it is part of the goal state. This is not the case for block `3` in Figure 3.4. Block `3` is, however, *in the way*. This means that the block prevents moving another block that is misplaced, and therefore needs to be moved itself (in the example in Figure 3.4, we have that `on(b5,b1)` where `b1` is misplaced but cannot be moved without first moving `b5`). Provide a cognitive state query that expresses that a block `X` is *in the way* in the sense explained, and explain why it expresses that block `X` is in the way. Keep in mind that the block below `X` may also not be part of the goal state! Only use predicates available in Figure 3.4.

# Chapter 4

# Action Specifications

An agent performs actions to effect changes in its environment in order to achieve its goals. It is important for the agent to know when an action can be performed. An agent should only choose to perform an action if the conditions for successfully performing it hold. We introduce **action specifications** in this chapter to specify when an action can be done and what its effects are. We also introduce actions that are part of the programming language itself. These actions are **internal** actions. Actions made available by an environment are called **external** actions. Unlike external actions, internal actions are not send to an environment for execution. Action specifications must be provided for all external actions and for newly defined internal actions that are not provided by the language.

## 4.1 Action Specifications

Unlike other programming languages, but similar to planners, actions that are made available by an environment to an agent need to be specified as part of the agent program. The GOAL language also provides actions that are part of the language itself. These do not need to be specified as the conditions when these actions can be performed and their effects are fixed by the language. Actions are defined by specifying the conditions when an action can be performed and the effects of performing the action. The former are also called **pre-conditions** and the latter are also called **post-conditions**.

**Action Specification File** Actions are specified in one or more **action specification** files. These files must be imported in modules that use these actions. Action specifications in GOAL are similar to STRIPS-style action specifications [33] and are of the form:

```
define id(parameters) with
   pre { qry }
   post{ upd }
```

An **action** is specified or defined by its **name** `id`, its **parameters** `parameters` of the form $(t_1, \ldots, t_n)$, where each $t_i$ must be a *variable*, a **pre-condition** `qry` and a **post-condition** `upd`. The parameters of an action are instantiated at runtime. An action does not need to have parameters and in that case only a name (without brackets) needs to be provided. A pre-condition can be any valid query in the KR language used. A post-condition must be a conjunction of literals, i.e., facts or negations of facts. In Prolog this means that the difference is that a pre-condition can also be a disjunction but a post-condition cannot. You can also not use built-in, pre-defined predicates of the Prolog language in a post-condition. Built-in predicates define useful concepts that should not be updated.

A very simple example of an action specification is the following:

```
define skip as internal with
  pre { true }
  post{ true }
```

This action specification specifies the action `skip` without parameters that can always be performed and has no effect. The use case **as internal** indicates that the system should not sent the action `skip` to an environment but treat it as an action that is handled internally by the agent (for updating its state or otherwise). The **true** precondition indicates that `skip` can always be performed. The **true** postcondition indicates that the action has no effect. It would also have been ok to write empty pre- and postconditions `{ }` here, but by making these conditions explicit you indicate that you have not just forgotten to specify them and paid attention to it.

An action specification file consists of use clauses and one or more action specifications. The use clauses should reference KR files that define the predicates that are used in the pre- and post-conditions of actions. At least one KR files should be specified to enable the system to infer which KR language is used (cf. also the empty `dummy.pl` file used for this in Chapter 1). An action specification defines the action's name and parameters, whether it is an internal or external action, and the action's pre- and post-condition. Parameters, if any, must be variables.

| | | |
|---|---|---|
| *specification* | := | *useclause*$^+$ *actionspec*$^+$ |
| *useclause* | := | **use** *id* [**as knowledge**] **.** |
| *actionspec* | := | **define** *id*[ (*parameters*) ] [*asclause*] **with pre{** *qry* **} post{** *upd* **}** |
| *asclause* | := | **as** (**internal** \| **external**) |
| *qry* | := | *a valid KR query* \| **true** |
| *upd* | := | *a valid KR update* \| **true** |
| *parameters* | := | *variable* (**,** *variable*)$^*$ |
| *variable* | := | *a variable of the KR language* |

Table 4.1: Action Specification Grammar

A use case of an action definition specifies whether the action should be treated as an internal action or not. If no use case is specified the default **external** use case is assumed. If an agent is connected to an environment, external actions will be sent to the environment for execution. In that case, it is important that the name of an action corresponds with the name the environment expects to receive when it is requested to perform the action. Check the environment's documentation to obtain this information. If an agent tries to perform and send an action to an environment that is not recognized by that environment, the agent program will generate an error and will be terminated.

Continuing the Blocks World example introduced in Chapter 3, we will now describe the robot gripper that enables the agent that controls this gripper to pick up and move blocks. The robot gripper can pick up at most one block at a time. It thus is not able to pick up stacks of blocks. As is common in the simplified version of the Blocks World, we abstract from all physical aspects of robot grippers. In its most abstract form, the gripper in the Blocks World is modeled as a robot arm that can perform the single action of picking up and moving one block to a new destination in one go. That is, the picking up and moving are taken as a single action that can be performed instantaneously. This action can be specified as follows:

```
use bwknowledge as knowledge.

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

Because we want to use this file later create it now and name it `bwmove.act2g`. We can then later import it by including the use clause **use** `bwmove` **as actionspec** in a module. (Note

that as before the extension of the action specification file does not need to be provided.) The use case keyword **actionspec** here is optional if the file name itself is unique. Note also that we need to declare the use case for the knowledge that is used in the pre-condition.

### 4.1.1 Pre-conditions

The keyword **pre** in an action specification indicates that what follows is a **pre-condition** of the action specified. Pre-conditions are queries of the knowledge representation language that is used (e.g., a Prolog query). Pre-conditions are used to verify whether it is possible to perform an action. A pre-condition $\varphi$ is evaluated by verifying whether $\varphi$ can be derived from the belief base (as always, in combination with knowledge in the knowledge base). Free variables in a pre-condition may be instantiated during this process just like executing a Prolog program returns instantiations of variables. An action is said to be **enabled** whenever the agent believes that the action's pre-condition holds.[1]

The precondition for the `move` action specifies that in order to be able to perform the action `move(X,Y)` of moving `X` on top of `Y`, both `X` and `Y` must be clear. The condition **not** `(on(X,Y))` expresses that it a `move` action cannot be done if block `X` is already sitting on top of `Y`.

The condition `on(X,Z)` in the precondition does *not* specify a condition that needs to be true in order to be able to perform the `move` action in the environment. Instead, this condition retrieves in the variable `Z` the thing, i.e. the block or table, that block `X` is currently on. The reason for including this condition in the precondition is that it is needed to be able to specify the effects of the `move` action. After moving block `X`, the block no longer is where it was before. This fact about `X` thus must be removed after performing the action which explains why we need to retrieve this information first from the agent's belief base in the pre-condition of the action.

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

Preconditions do not always need to include all conditions that are required for the successful execution of an action. Although it is good practice to provide complete action specifications, sometimes there are pragmatic reasons for not providing all required pre-conditions. For example, the precondition for the `move` action does not require that `X` is a block. Strictly speaking, the precondition including the condition `block(X)` would result in a better match with constraints in the Blocks World, as only blocks can be moved in this environment, and thus, adding the `block(X)` condition would have resulted in a more *complete* action specification. However, when efficiency is considered as well, it makes sense to provide a pre-condition that checks as few conditions as possible. It may also be that a pre-condition is better readable when not all details are included. We thus may trade off completeness of an action specification for efficiency and readability. A developer should always verify, however, that by not including some of the actual pre-conditions of an action the program still operates as expected. As we will see in Chapter 5, the cognitive state queries of a rule that provide the reason for selecting a move action prevent the

[1] Note that because an agent may have false beliefs, the action may not actually be enabled in the environment. As 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. Environments may throw exceptions when an agent attempts to execute an action that it is not allowed to at that moment.

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

variable `X` from ever being instantiated with `table`. Therefore, dropping the condition `block(X)` from the pre-condition does not do any harm.

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

**Completely Instantiated Actions**
The reason that the variables that occur in an action's parameters or in its postcondition must be instantiated with ground terms is that otherwise it is not clear what it means to execute the action. What, for example, does it mean to perform the action `move(X,table)` with `X` a variable? One option would be to select a random item and try to move that. Another would be to try and move all items. But shouldn't we exclude tables? We know intuitively that the `move` action moves blocks only. But how would the agent program know this?
The classic STRIPS-style specification [38] requires that all variables in the pre-condition also appear in the parameters of the action. This requires that instead of the `move(Block,To)` action specified above an action `moveFromTo(Block,From,To)` must be specified as the `From` variable needs to included in the action's parameters. This restriction is lifted in GOAL and no such constraints apply. In order to make sure that an action's post-condition is closed, it is required, however, that all variables that occur in the post-condition also occur in either the pre-condition or the action's parameters. Note that any variables in action parameters that do not occur in the pre-condition must be instantiated by other means before the action can be performed; in an agent program this can be done by using cognitive state queries.

### 4.1.2 Post-conditions

The keyword **`post`** indicates that what follows is a **post-condition** of the action. Post-conditions are conjunctions of literals, i.e., facts or negated facts. Each variable in a post-condition must also occur in one of the action parameters or in the action's pre-condition. This is required to make sure that all variables in the post-condition will be instantiated when the action is performed. A post-condition specifies the effect of an action. Action post-conditions are used to update the cognitive state of an agent in GOAL. A post-condition $\varphi$ is used to update the beliefs of an agent such that the agent believes after the update that the effects $\varphi$ of the action hold.

The agent's beliefs are updated by *adding* the positive literals (facts) in the post-condition to the agent's belief base and by *removing* the negative literals in the post-condition from the beliefs. A post-condition can also be thought of as an *add/delete list*, as in STRIPS-style action specifications [19, 33]. Positive literals in a post-condition are part of the add list and all facts $\varphi$ that occur in negative literals **`not`**`(`$\varphi$`)` are part of the delete list. The effects of performing an action on the cognitive state of the agent are that the belief base is updated by *first* removing all facts that are part of the delete list and *thereafter* adding all positive literals in the add list. Note that adding a fact to the belief base that is already present in it has no effect, as any fact will only appear once (i.e., the bases in an agent's state are sets). Deleting a fact that was not present to begin with also has no effect.

The postcondition **`not`**`(on(X,Z)), on(X,Y)` in the action specification for `move(X,Y)` consists of one negative and one positive literal. The add list in this case thus consists of a single atom `on(X,Y)` and the delete list consists of the atom `on(X,Z)`. It is important to remember that all variables must have been instantiated when performing an action and that both `on(X,Y)` and `on(X,Z)` are only templates. *Immediately* after the agent decides to perform the action, the agent's state is updated by first removing all atoms on the delete list and thereafter adding all

atoms on the add list. For the `move` action this means that the current position `on(X,Z)` of block `X` is removed from the belief base and thereafter the new position `on(X,Y)` is added to it.

**Instantaneous and Durative Actions** Performing an *external* action such as the `move` action not only updates the agent's state, using the action's post-condition, but also means that the action is sent to the environment that the agent is connected to. The environment forwards the action to the entity that the agent is connected to. The entity thus is requested to perform the action in the environment. As all of this and performing the action itself may take time, it is important to distinguish between so-called **instantaneous** and **durative** actions. The difference between the two is that the time needed to execute an instantaneous action can be neglected but the time needed to execute a durative action cannot. For all purposes, the effects of an instantaneous action are realized immediately when the action is performed. By default, you should assume that an external action executed in an environment is durative with only very few exceptions (which should be indicated in the documentation for an environment). In the classic Blocks World environment we may assume that the effects of a `move` action are realized instantaneously. This is not the case for durative actions, such as navigating a robot or moving a gripper to a block. It takes time to realize the effects of durative actions.

This has important implications for the specification of post-conditions. For instantaneous actions, we can specify the effects of the action in a post-condition. We cannot do the same for durative actions, however. The post-condition of an action is *immediately* used to update the agent's state. Performing this update for durative actions, however, would result in the agent believing that the effects have been achieved before they have actually been achieved. This is undesired and for this reason we cannot specify the effects of a durative action in the post-condition of its action specification. There is a second reason why we cannot always reliably specify the effects of an action in its post-condition: The effects of a durative action might never be realized because the action might fail, e.g., when an agent cannot navigate somewhere because obstacles prevent it from going there.

As a rule, we therefore use empty, or better **`true`**, post-conditions for durative actions. Generally speaking, it is only useful to specify the post-conditions of internal actions. Only in very special cases where external actions can be treated as instantaneous actions, such as the `move` action in the Blocks World, we should specify action effects in the action's post-condition. In all other cases, the effects of external actions need to be perceived by the agent in the environment (see Chapter 6).

**Closed World Assumption (Continued)** The STRIPS approach to updating the state of an agent when an action is performed is based on the Closed World Assumption. It assumes that the database of beliefs of the agent consists of *positive* facts only. Negative information is inferred by means of the Closed World Assumption (or, more precisely, negation as failure if we use Prolog). Negative literals in the post-condition are taken as instructions to remove positive information that is present in the belief base. The action language ADL does not make the Closed World Assumption and allows to add negative literals to an agent's belief base [38].

### 4.1.3 Updating an Agent's Goals

Performing an action may also affect the goal base of an agent. If an agent, after performing an action, comes to believe that a goal has been *completely* achieved, that goal is automatically removed from the agent's goal base. This automatic removal of a goal that has been achieved is based on the fact that a rational agent should not invest any resources, whether energy or time, into achieving a goal that already has been realized.

Goals that have been achieved as a result of performing an action are removed but only if they have been achieved *completely*. A goal is achieved only when all of its sub-goals are achieved. An agent should not drop any of these sub-goals before achieving the overall goal that they are part of has been achieved. Removing sub-goals instead of a complete goal from an agent's goal base

would make it impossible for the agent to ensure that all sub-goals of a goal are *achieved at one and the same moment* in time.

The removal strategy that only removes a goal when it has been achieved completely implements a so-called **blind commitment strategy** [36]. Agents should be committed to achieving their goals and should not drop goals without reason. The default strategy for dropping a goal therefore is strict: Only do this when the goal has been completely achieved. This default strategy can be adapted by the programmer for particular goals by using the built-in **drop**action.

For example, assume an agent has the following belief and goal base:

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

and successfully performs the action `move(b1,b2)`. By updating the state with the action's post-condition we would get the following state:

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

This cognitive state is *not* rational. The goal has been achieved, and, for this reason, should be removed to avoid the risk of the agent wasting resources on trying to achieve the goal while it has already been achieved. GOAL implements this *rationality constraint* and automatically removes achieved goals, which yields a new state in our example with an empty goal base:

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

## 4.2 Internal Actions Provided in the Language

In addition to the possibility of specifying internal actions, several internal actions are provided as part of the GOAL language. These actions include actions for changing the beliefs and goals of an agent, for printing messages to the console, logging information to a file, and for communicating information with other agents. We discuss these actions here except for the `send` action that can be used to communicate a message to another agent, which is discussed in Chapter 8.

**Modifying an Agent's Cognitive State** There are four internal actions available for modifying an agent's cognitive state. First, the action:

```
adopt(qry)
```

can be used to adopt or add a new goal `qry` to an agent's goal base. `qry` should be a conjunction of facts; negative literals are not allowed. In line with our discussion about Prolog databases before, only conjunctions of positive literals can be added to an agent's goal base. The action **adopt**(**not**(`fact`)) is not allowed and would generate a syntax error. The **adopt**action can only be used to add a goal and cannot be used to remove one. Recall that a goal is automatically removed when it is completely achieved. If everything works as planned and the agent has been programmed correctly to achieve its goals, an agent thus would never have to remove a goal manually. Because an agent does not always have full control over what happens, however, it may sometimes be necessary to reconsider a goal and drop it manually. If an agent considers that a

goal it has is no longer feasible, the **drop**(qry) action can be used to remove that goal. It is considered best practice to limit the use of the **drop**action as much as possible. It should only be used to remove goals that an agent comes to believe are no longer feasible.

An **adopt**(qry) action be performed if qry is not believed and qrt is not implied by any of the goals of an agent. That is, the pre-condition of an **adopt**action is that the agent does not believe that qry holds and does not have a goal qry. In other words, to perform **adopt**(qry) we must have **not**(**bel**(qry)), **not**(**goal**(qry)). The reason is that it is not rational to adopt a goal that has already been achieved nor to add a goal that follows already from the (other) goals of an agent. The effect of an **adopt**action is that qry is added as a single, new goal to the goal base.

The action:

```
drop(qry)
```

can be used to drop or remove one or more goals from an agent's goal base. qry should be a conjunction of literals; negative literals thus can be used. The **drop**action can always be performed, i.e., its pre-condition is true. The effect of a **drop**action is more subtle. Informally, a **drop**(qry) action removes each goal that implies qry. More precisely, any goal from which in combination with the agent's knowledge qry can be derived is removed from the agent's goal base. For example, if on(b1,b5) follows from a goal in the goal base, the action **drop**(on(b1,b5)) would remove that goal. In the example of Figure 3.3, the single goal present in the goal base would be removed by this action.

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

```
insert(upd)
```

can be used to insert upd into the agent's belief base. upd should be a conjunction of literals. The **insert**action can always be performed. The update performed by an **insert**action is the same as that performed by applying an action's post-condition. The **insert**action *adds* all positive literals that occur in upd to the belief base of the agent and *removes* all facts that occur in negative literals in upd from the belief base. For example, **insert**(**not**(on(b1,b2)), on(b1,table)) removes on(b1,b2) from the belief base and adds on(b1,table). As a result, after performing an **insert**(upd) action an agent believes upd.

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

Finally, the action:

```
delete(upd)
```

can be used to delete upd from the agent's beliefs. The **delete**action is kind of the inverse of the **insert**action: Instead of adding literals that occur positively in upd it removes such literals, and instead of removing negative literals it adds such literals.

Strictly speaking we could do without the **delete**action as we can achieve the same result with an **insert**action. Using both actions, however, makes it easier to read a program. We consider it best practice to *use **insert**only for adding facts and use **delete**only for removing facts.* For example, instead of writing **insert**(**not**(on(b1,b2)), on(b1,table)), it is better to use a combination of the two actions **insert**(on(b1,table)) and **delete**(on(b1,b2)). Multiple actions can be combined by the + operator. Using this operator we can write:

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

which would have exactly the same effect as the single action **insert**(**not**(on(b1,b2)), on(b1,table)). The + operator can be used to combine as many actions as needed. As a rule of thumb, however, it is best practice to at most include one external action in a list of actions that are combined by +. The actions combined by + are executed in order of appearance (see also Section 5.2).

One thing to note about the **delete**(upd) action is that it removes *all* positive literals that occur in upd. The **delete**action thus does not support *minimal change* in the sense that no more information is removed than is strictly necessary to make sure that the agent does not believe upd any more. For example, **delete**(p, q) removes both p and q instead of only removing either p or q which would be sufficient to remove the belief that p and q both are true.

**Printing a Message** We have already seen how to use the **print**action in Chapter 1. The action:

```
print(term)
```

prints term to the standard output console. term can be any term, including a variable, as long as the term is closed when the **print**action is performed.

**Logging to a File** The action:

```
log(parameter)
```

can be used to write logging information to a file. The parameter supports the following options:

- **log**(bb) means that the contents of the agent's belief base are written to the log file;
- **log**(gb) means that the contents of the agent's goal base are written to the log file;
- **log**(pb) means that the contents of the agent's percept base are written to the log file;
- **log**(mb) means that the contents of the agent's message base are written to the log file;
- **log**(parameter) for any other parameter than in the previous bullets means that the parameter itself is written to the log file.

Because logging is platform dependent, more information on this action can be found in the User Manual [28].

Finally, like all other actions, internal actions must be closed when they are executed. That is, all variables must have been instantiated with ground terms. This is also required for actions that are combined by the + operator.

## 4.3 Notes

The strategy for updating an agent's goals in GOAL is a *blind commitment strategy*. Using this strategy, an agent only drops a goal when it believes the goal has been achieved. In [36] and [15] 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 the goal is no longer achievable. For example, an agent may drop a goal to get to a meeting before 3PM if it misses the train. However, providing automatic support for removing such goals when certain facts like missing a train are believed by the agent is far from trivial. GOAL provides the **drop** action which can be used to remove goals that are no longer deemed achievable by writing rules with such actions as conclusions.

## 4.4 Exercises

### 4.4.1

We revisit the “Hello World" agent that was developed in Chapter 2 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 2. In other words, you should now make sure that the agent's beliefs remain up to date by means of the action specification for `printText`. This will require you to write another precondition and postcondition than **`true`** for the action `printText`.

- Start by removing the use case for the **`event`** module in the agent definition.
- 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 inserts the new updated fact.

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

# Chapter 5

# Cognitive Decision-Making Agents

Using the cognitive state and action specification program components introduced in the previous chapters, we will define an agent that is able to solve Blocks World problems. We need to specify one more important part: the decision-making of an agent. To that end, we will introduce **decision rules** for selecting actions in this chapter. These rules allow us to create cognitive decision-making agents, i.e., agents that derive their choice of action from their beliefs and goals. We will also learn how to write basic **module**s. Modules are program components that are used to combine all the other parts: they include the necessary **use clause**s for importing the cognitive components and action specifications that an agent needs to make decisions using rules. We will also see how to create an **agent definition**. In order to be able to run the Blocks World agent, we will look at how to write a basic **multi-agent program**.

## 5.1 Solving Blocks World Problems

Our goal is to develop a strategy for an agent that effectively solves Blocks World problems. A strategy determines what the agent should do next. In order to find a good strategy for the Blocks World, we now first continue our discussion of this environment introduced in Chapter 3. The Blocks World is a simple environment that consists of a finite number of blocks that are stacked into *towers* on a table of *unlimited* size, where each block has a unique label or name. Blocks obey a simple set of "laws": a block is either on top of another block or it is located somewhere on the table, a block can be directly on top of at most one other block, and there is at most one block directly on top of any other block.

A solution to a Blocks World problem requires a strategy for deciding which block to move and where to move it to. In our version of the problem, we do not worry about the exact positioning of towers on the table. Any configuration where the desired towers are positioned somewhere on the table is ok. We will continue with the example problem illustrated in Figures 3.1 and 3.2 and use the specification of knowledge, beliefs, and goals in Chapter 3. We will also use the action specification for the move action of Chapter 4.

There are some basic concepts and insights that help solving a Blocks World problem. First, a block is said to be *in position* if the position of the block in its current state corresponds with the goal state. A block's position only corresponds with the goal state if all blocks below it (if any) are also in position. A block that is not in position is said to be *misplaced*. Recall our definition of this concept in Chapter 3. In our example problem illustrated in Figures 3.1 and 3.2, all blocks except for block `3` and `7` are misplaced. Only misplaced blocks have to be moved in order to solve a Blocks World problem. The action of moving a block is called *constructive* if in the resulting state that block is in position. Observe that a constructive move always decreases the number of misplaced blocks. Also note that it does not make sense to move a block on top of another block if that move is not constructive. We would need to move that block again at a later time. But worse, it limits our options as it prevents us from moving the block that it is moved on top of.

Because there is no limit to the number of blocks that fit on the table, it is therefore always a better option to move the block to the table.

**Blocks World Problems Can Be Hard** We will develop a *good*, not an *optimal*, strategy for solving Blocks World problems. The performance of an agent that solves Blocks World problems is measured by means of the number of moves it uses to transform an initial state or configuration into the goal state. An agent performs optimally if it is not possible to improve on the number of moves it uses to reach a goal state. 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 [20]. See [24] for an approach to define heuristics to obtain near-optimal behaviour in the Blocks World in an extension of the GOAL language.
What makes it hard to solve some Blocks World problems optimally, is that it is not always possible to make a constructive move. If a constructive move cannot be performed in a configuration, that Blocks World configuration is said to be in a *deadlock* [42]. A special case of deadlock is where a block is in a *self-deadlock*. That is the case if it is misplaced and above another block which it is also above in the goal state; for example, block `1` 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 [42].

## 5.2 Decision Rules

Agents need to make decisions on what to do next. In the Blocks World, a decision must be made where to move a block to, for example, whereas a mobile robot controller agent must decide where to navigate to or whether to pick up something with a gripper or not. Such a decision depends on the current state of the agent's environment as well as general knowledge about this environment. In the Blocks World, an agent needs to know what the current configuration of blocks is and needs to have basic knowledge about such configurations, such as what a tower of blocks is, to make a good decision. A decision to act will usually also depend on the goals of the agent. In the Blocks World, a decision to move a block on top of an existing tower of blocks would be made, for example, if it is a goal of the agent that the block sits on top of that tower. In a robotics domain, it might be that the robot has a goal to bring a package somewhere, and therefore picks it up.

Decision rules provide an agent with the know-how *why and when* to perform an action. To select an action, an agent needs to be able to *inspect* its knowledge, beliefs and goals and specify which action to *select*. Therefore, decision rules are of the form:

**`if`** *csq* **`then`** *action*`.`

The condition *csq* of a rule is a cognitive state query (Chapter 3) and *action* is any action (Chapter 4) that the agent can perform. The action of a rule can be an internal action made available by the programming language or specified by a user, or an external action that is made available by an environment that the agent is connected to.

The cognitive state query in a rule determines whether the corresponding action may be considered for execution or not. The query is a condition on the agent's state that provides a **reason** for selecting the action. An agent thus decides which action to perform next based on its cognitive state. This is why we also say that *an agent derives its choice of action from its beliefs and goals.*

We need to write decision rules for solving a Blocks World problem. This means specifying the reasons *why* and situations *when* to perform a move action. We also need to specify *which* move action to perform, e.g., move a block to the table or onto another block. There are many strategies for solving a Blocks World problem. We note that because the table always has room for a block it is never necessary to move a block more than twice. We should thus do better than making two moves per block. One simple strategy is to first move all blocks to the table and then

start building towers. This is called the *unstacking strategy*. This is not an optimal strategy but requires on average only about 1.2 moves per block. We will slightly improve on this strategy by using the concepts of a constructive move and misplaced block. The idea is to always make a constructive move if possible, and to otherwise move a misplaced block to the table. This avoids moving a block that can be put in position more than needed and only resorts to the unstacking strategy when there is no other alternative. Using the cognitive state queries for a *constructive move* and *misplaced* in Chapter 3, we obtain the following rules that implement this strategy:

```
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).
```

If the pre-condition of an action holds (in the current state of the agent), we say that the action is **enabled**. If a cognitive state query of a rule holds and the action of the rule is enabled, we say that the rule is **applicable**. This means that the action is an **option**. Rules in this sense *generate options*.

The first rule generates the option to move a block `X` on top of another block `Y` if the agent wants to construct a tower `[X, Y| T]` and believes that the part `[Y | T]` is already in place. It thus generates the option to make a constructive move. The second rule generates the option to move a block to the table if the agent wants that block to be part of a tower `[X| T]`. It thus generates the option to move a block to the table if that block is misplaced.

**Rules with Composed Actions** It is often useful to allow an agent to select more than one action using only a single decision rule. This can be done by means of the + operator in GOAL. Multiple actions that are combined using + also is called a **composed action** or a **combo-action**. Two examples are:

```
insert(on(b1,b2)) + delete(on(b1,b3))

print("Going up") + goto(5, up)
```

Both examples combine two actions but there is no limit to the number of actions that can be combined using +. The first example combines the two internal actions **`insert`** and **`delete`**. The first action is used to insert a new fact about the position of block `1` and the second action is used to remove the fact `on(b1,b3)` from the belief base. The second composed action consists of the internal **`print`**action and the environment action `goto` that is available in the elevator simulator. The **`print`**action is used to print the string to the console (which is useful, e.g., for debugging purposes). The `goto` action makes the elevator move to a particular floor.

The actions that are part of a composed action may be put in any order in the action part of a rule. The actions are executed in the order they appear. It is not guaranteed that all actions in a combo-action are all performed. This depends on the pre-conditions of each action, which are evaluated when the action is about to be performed. For example, in the second combo-action above, the `goto` action may not be performed if its pre-condition does not hold. The **`print`**action can always be performed.

As a result, several things can happen when evaluating a rule with a combo-action. For example, when evaluating the rule:

```
if bel(clear(b1)) then adopt(on(b1,b2)) + move(b1,b2).
```

the following things can happen:

- the rule is not applicable because `clear(b1)` is not believed,
- the rule is not applicable because the agent believes `on(b1,b2)` and, as a consequence, the pre-condition of the **`adopt`**action fails,

- the rule is applicable, **adopt**(on(b1,b2) is executed, but the move action is not because its pre-condition fails, because block 2 is not clear (note that the rule condition already checks if block 1 is clear), or
- the rule is applicable, **adopt**(on(b1,b2)) is executed, and thereafter the move action is executed (and the goal just adopted has been achieved and is thus removed again).

## 5.3 Combining Everything in a Module

A rule uses various language elements that are specified elsewhere. The cognitive state query of a rule uses predicates specified in KR files. The action part of a rule uses the definition of an action in an action specification file. An agent needs all these elements together for executing a strategy specified by decision rules. All these elements are combined in a **module**:

```
use bwknowledge as knowledge.
use bwbeliefs as beliefs.
use bwgoals as goals.
use bwmove as actionspec.

exit = nogoals.

define constructiveMove(X,Y) as a-goal(tower([X,Y| T])), bel(tower([Y|T])).
define misplaced(X) as a-goal(tower([X| T])).

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

A module file is a separate program file containing a single module. A module has a **name** and a **program section** that contains rules. The module file for our module should be named stackBuilder.mod2g. The program section specifies the strategy that is used by the agent to select an action by means of decision rules. As these are all the rules, the agent will only generate options that are either constructive moves or move misplaced blocks to the table. The reader is invited to verify that the agent will never consider moving a block that is in position. Furthermore, observe that the cognitive state query 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. It follows that whenever a constructive move move(X, Y) is an option, the action move(X, table) is also an option.

At the start of a module file, **use clause**s must be specified that provide definitions for the predicates used (see Chapter 3) and specifications for the actions used (see Chapter 4). The only predicate used is the tower/1 predicate, which is defined in the bwknowledge.pl file. This file is included by the first use clause of the module. Note that the file extension is not specified; the system automatically uses the extension to check which knowledge representation language is used. The **as** part of a use clause indicates which state component should be initialized. External actions and user-specified internal actions that are used must specified in an action specification file. If no action specification use clause is present, the agent can only perform actions that are provided by the programming language itself. We include the bwmove.act2g file of Chapter 4 in which we specified the move action that is used in the decision rules of the module.

The two use clauses that follow the first are not required. They have been included to initialize the cognitive state of the agent with initial beliefs and an initial goal, respectively. We have used the KR files bwbeliefs.pl and bwgoals.pl that we created in Chapter 3. As we have seen in Chapter 3, the initial and goal state define a Blocks World problem; these two use clauses thus specify the problem the agent should solve. The use clause for the action specification bwmove is required because the move action is used in the module.

A module file can also contain **macro**s. Macro definitions must be provided after the use clauses by using the `define` keyword. Two examples are provided above that define the concept of a constructive move and a misplaced block. A macro has a name and (optional) parameters. Parameters must be variables which must occur in the definition that follows the **as** keyword. The definition itself is a cognitive state query. A macro can be viewed as an abbreviation for that cognitive state query. Macros can be used in the conditions of rules as illustrated in the module's rules. They serve to increase the readability of a program.

A module can also use a rule order and an exit condition but we delay discussion of these module options to Chapter 7.

## 5.4 Defining Agents in a Multi-Agent Program

We are close now to an agent that can solve a Blocks World problem. The module that we have defined above is pretty much all we need for creating the agent. The only thing left to do is to *define this agent*. Agents are defined in a multi-agent program. A multi-agent program also specifies the environment that an agent is connected to and a launch policy for launching agents. We use the following program file called `BlocksWorld.mas2g`:

```
use "blocksworld-1.1.0.jar" as environment with start=[2,3,0,5,0,7,0].

define stackBuilder as agent {
   use stackBuilder as main module.
}

launchpolicy {
   when * launch stackBuilder.
}
```

Figure 5.1: A MAS file that connects agent `stackBuilder` to the Blocks World

An (external) environment is specified at the beginning of a multi-agent program by a use clause. This use clause should have **as environment** as use case. The environment reference in our multi-agent program includes a Blocks World environment by means of specifying a path name to an environment file (the path to a `jar` file relative to the multi-agent program file or MAS file). Using the **with** keyword allows for passing one or more initialization parameters to the environment, in this case the initial configuration of blocks. The parameters that can be specified depend on the environment. The `start` parameter for the Blocks World requires a list of numbers, where each number $n_i$ indicates that block $i$ sits on top of block $n_i$ (or the table if $n_i = 0$). In the MAS file above, for example, block `1` sits on top of block `2` which sits on top of block `3` which sits on the table.

The environment use clause is followed by a single agent definition. An agent definition specifies the name of the agent and must have a definition section with use clauses that reference modules needed for creating the agent. In our case, we only have a single use clause that indicates that the `stackBuilder` module that we created in the previous section should be used **as main** module. The main module is the module that contains the main decision rules of the agent. A use clause in an agent definition can have three use cases: either **main**, **event**, or **init**. The only requirement an agent definition must satisfy is that it contains at least a **main** module *or* an **event** module.

Finally, the MAS file contains a **launch policy** section with one **launch rule** for launching an agent. This rule will build an agent named `stackBuilder` using the agent definition for `stackBuilder` when the gripper becomes available in the Blocks World. We will discuss this part of a MAS file further in Chapter 6.

## 5.5 Executing an Agent

Assuming that we have included all files that we need, including the environment file, in a single project folder, we are now ready to execute the multi-agent program. You can run the multi-agent program (see the User Manual [28] on how to do this) but the result is too fast gone again to see what happened. It is more useful to start the program in Debug mode (again, check the User Manual on how to do that). If you start the multi-agent program in Debug mode, the system is initially paused. This allows you to inspect the cognitive state of the agent, and to execute the program in a step-by-step fashion. You should see a window that has popped up and displays the initial state of the Blocks World of Figure 3.1. You also should see that an agent has been created that is called `stackBuilder` and that an environment process called `blocksworld-1.1.0` has been launched. Both agent and environment should be paused.

It is useful to **trace** the behaviour that the agent that we have created once in detail to understand what happens. Before we start, we should not forget to put the Blocks World in running mode. (You should check for yourself what happens if you forget to do so.) Select the environment process and put it in running mode. Now we are ready to step through the agent program.

First inspect the cognitive state of the agent. You should find that all components except for the percept base of the agent are empty. The percept base contains the facts that represent the initial state of the Blocks World. We will use these in Chapter 6 to initialize the agent's belief base. As you may recall, we have chosen to initialize the agent's beliefs by means of the Prolog file `bwbeliefs.pl` here. This has not yet happened. The use clause that imports these beliefs is applied when you step into the module. Do so now to see the results. You should not only see facts in the belief base of the agent that represent the initial Blocks World state but also a goal in the agent's goal base that represents the goal state in Figure 3.2.

If you continue stepping through the agent program, you will find that the first decision rule in module `stackBuilder` is not applicable. There is no constructive move that the agent can make. The second decision rule is applicable and will select to move block `1` to the table. If you continue stepping you will find that the first rule is selected again and the agent has begun its second **execution cycle**. In this cycle, there is still no constructive move that can be performed, and instead the agent will apply the second rule and move block `2` to the table. This move makes it possible to move block `4`. In the third execution cycle the agent will apply the first decision rule and perform the constructive move that puts block `4` on top of block `3`. The agent needs three more cycles to achieve its goal. In these cycles the agent does the following:

- Execution cycle 4: moves block `5` onto block `2`.
- Execution cycle 5: moves block `1` onto block `5`.
- Execution cycle 6: moves block `1` onto block `5`.

In cycle 6, the goal is achieved and removed from the agent's goal base. At that moment, no new cycle is started but the agent is terminated because of the exit condition **`nogoals`** that we added in the `stackBuilder` module.

## 5.6 Summary

A **module** combines the different elements needed to create a cognitive decision-making agents. **Use clause**s at the start of a module specify the required cognitive state components, e.g., the knowledge, beliefs, and/or goals, and the action specifications used by the agent. The knowledge, beliefs and goals of an agent are specified in KR files, e.g., Prolog files (see Chapter 3). Action specifications specify the pre- and post-conditions of an action and are defined in an action specification file (see Chapter 4). A module's **program section** consists of **rule**s for selecting the actions that the agent will perform. **Decision rules** are the rules that the agent uses to select

which actions to perform to achieve its goals. These rules typically are part of a module that is imported as **`main`** module.

A **multi-agent program** is a recipe for creating and launching agents. It also informs the system which environment should be launched (if any). An **agent definition** in a multi-agent program or MAS file specifies which modules should be used to create an agent.

## 5.7 Notes

The GOAL programming language was first introduced in [27]. Previous work on the programming language 3APL [26] had made clear that the concept of a declarative goal, typically associated with rational agents, was still missing. Declarative goals at the time were the "missing link" needed to bridge the gap between agent programming languages and agent logics [15, 36, 30]. The programming language GOAL was intended to bridge this gap. The design of GOAL has been influenced by the abstract language UNITY [14]. It was shown how to provide a temporal verification framework for the language as well, as a first step towards bridging the gap. In [23] it has been shown that GOAL agents instantiate Intention Logic [15], relating GOAL agents to logical agents specified in this logic.

A set of decision rules is similar to a *policy*. There are two differences with standard definitions of a policy in the planning literature, however [19]. First, decision rules do not need to generate options for each possible state. Second, decision rules may generate *multiple* options in a particular state and do not necessarily define a function from the (cognitive) state of an agent to an action. In other words, a strategy of a GOAL agent as defined by its decision rules does not need to be *universal* and may *under-specify* the choice of action of an agent.[1]

The book *Multi-Agent Programming* [10] provides references to other agent programming languages such as 3APL, Jason, and Jadex, for example.

[1] "Universal" in the sense of [39], where a *universal plan* (or policy) specifies the appropriate action for *every* possible state.

# Chapter 6

# Environments: Actions & Sensing

> 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: [11]

We connected the agent that we developed in Chapter 5 to an environment called the Blocks World. The agent performed external actions to change the configuration of blocks in that world but did not process the **percept**s that it received from the Blocks World. In this chapter we will extend the agent and add rules that enable it to process percepts. We will introduce a new operator **`percept`**for inspecting an agent's **percept base** (see Figure 3.3). The `move` action of the Blocks World could be treated as an **instantaneous** action (see 4.1.2). In this chapter we will shift our focus to the more common **durative** actions.

## 6.1 Types of Environments

Percepts inform an agent about the state of the environment it is connected to. Only in very special circumstances can an agent ignore this information. The Blocks World agent of Chapter 5 did not have to process percepts it received from the Blocks World because we could make several simplifying assumptions about this environment. This allowed us to focus all of our attention on the design of a strategy for selecting the right `move` actions. First of all, we could treat the `move` action as an **instantaneous** action. That allowed us to specify the effects of the action in a post-condition and update the cognitive state of the agent to reflect these changes immediately when performing the action. Another important assumption that we used was that the agent has **full control**. The agent has full control over the gripper and the gripper is the only way that blocks can be moved in the Blocks World. Moreover, we could be sure that a block is moved and the effects of the gripper are as expected. That is, the `move` action is **deterministic**. Full control also means that we could assume that no other events or other agents would change the configuration of blocks. That is, the Blocks world is **static** and cannot change if the agent does not do something. Finally, we also assumed that the Blocks World is a **single agent** environment and only one agent could make changes.

In summary, the Blocks World is a static, deterministic, single agent environment. In other words, the Blocks World agent has full control. Because we could also treat the `move` action as

instantaneous, we could disregard percepts. The agent could keep track of the configuration of blocks simply by using its knowledge of the effects of the action `move`. We used the post-condition of the action to represent this knowledge (see Chapter 4). The agent could keep track of the configuration by updating its beliefs using that knowledge, i.e. by applying the post-condition of the `move` action.

In most environments one or more of these assumptions do not hold. Most external actions are **durative**, i.e. take time to complete, and we cannot use knowledge about action effects to immediately update an agent's beliefs about the environment. In the Tower World, a variant of the Blocks World, it takes time before the gripper has moved a block as we will see below. An agent also hardly ever has full control over its environment. The effects of actions are not always certain and we thus cannot assume a deterministic environment. This is because actions may *fail* or the effects of performing an action are not completely predictable, e.g. effects are **stochastic**. In the Tower World the gripper does not always succeed to put a block at a particular position. An environment can also be **dynamic** in the sense that things change without an agent doing anything. In the Vacuum World, for example, dust can re-appear automatically. Finally, most environments are **multi-agent**. In such environments more than one agent can make changes and effect the environment. In the Tower World, for example, a user can also interact with the World by using the mouse to move blocks.

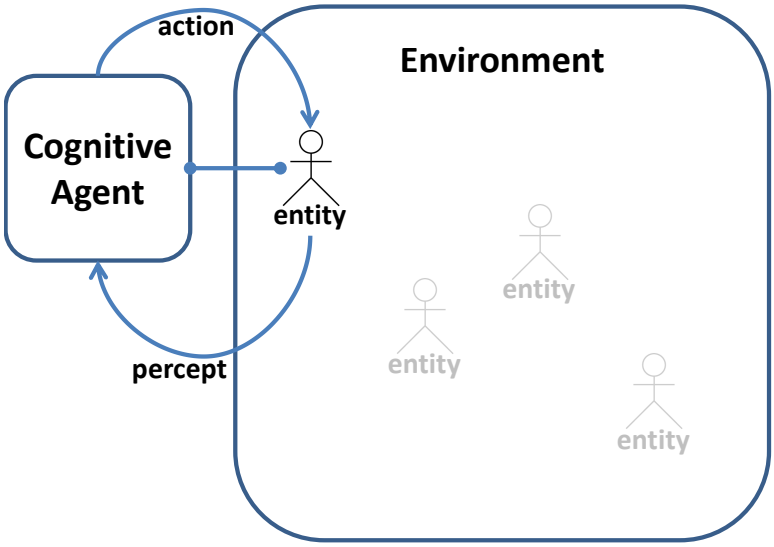


Figure 1.2 (reprinted): Cognitive Agent Connected to Entity in Environment

## 6.2 Agents, Environments, and MAS Files

Agents are connected to controllable entities (see Figure 1.2) in an environment by means of an **environment interface**. We say that agents are connected to an environment and to an entity rather than *being part* of an environment. We use this terminology to emphasize the interface that is present between the agent and the environment. The interface between the agent and the environment is used to *send requests* to the entity to perform actions in the environment and to *receive percepts* through the interface in order to observe (parts of) that environment.

The interface to an environment and the connections between agents and entities in an environment are established by a Multi-Agent System (MAS) file. A MAS file is a recipe for launching a multi-agent system. There are a number of tasks that are handled by a MAS file. A MAS file

*creates an interface* for connecting to an environment. An agent definition in a MAS file specifies how to *build an agent*. A third task supported by a MAS file is that it specifies when to *launch an agent* and add it to the multi-agent system. A MAS file also determines whether or not to *connect an agent to an environment*.

Table 5.1 specifies an Extended Backus-Naur Form (cf. [40]) grammar of a MAS file. Typewriter font is used to indicate *terminal symbols*, i.e. symbols that are part of an actual program. Italic is used to indicate *nonterminal symbols*. [...] is used to indicate that ... is optional, | is used to indicate a choice, and $^*$ and $^+$ denote zero or more repetitions or one or more repetitions of a symbol, respectively. The first clause of the grammar says that a MAS file consists of three sections. The first section is an (optional) **environment section**. Specifying an environment is optional because a agents can also be run without being connected to an environment. The second section of a MAS file consists of one or more **agent definition**s. The third section is a **launch policy** that specifies when and which agents to launch.

| | | |
|---|---|---|
| *mas* | := | [*environment*] *agent*$^+$ *policy* |
| *environment* | := | **use** *path* **as environment** [**with** *id* = *value* (**,** *id* = *value*)$^*$] **.** |
| *agent* | := | **define** *id* **as agent{** *useclause*$^+$ **}** |
| *useclause* | := | **use** *ref* **as** *usecase* **.** |
| *usecase* | := | **init** [**module**] \| **event** [**module**] \| **main** [**module**] |
| *policy* | := | **launchpolicy{** *launchrule*$^+$ **}** |
| *launchrule* | := | [**when** *entity*] **launch** *instruction* (**,** *instruction*)$^*$ **.** |
| *instruction* | := | *id* [**with** *constraint* (**,** *constraint*)$^*$ ] |
| *entity* | := | $\star$ \| **type** = *id* \| **name**= *id* |
| *constraint* | := | **name** = *id* \| **name** = $\star$ \| **number** = *num* \| **max** = *num* |
| *id* | := | *alphanumeral with underscores that starts with letter or underscore* |
| *num* | := | *natural number, starting from* 0 |
| *value* | := | *double quoted string, numeral, or list of values between square brackets* |
| *ref* | := | *id* (**.***id*)$^*$ |
| *path* | := | *double quoted string containing a file path* |

Table 6.1: Multi-Agent System Grammar

An example MAS file is provided in Figure 5.1. This MAS file connects an agent named `stackBuilder` that is built using the agent definition `BWagent` to the Blocks World. The MAS file is similar to the one we already saw in Section 5.4 but it also has some minor but important differences.

```
use "blocksworld-1.1.0.jar" as environment with start="bwconfigEx1.txt".

define BWagent as agent {
   use stackBuilder as main.
   use bwEvents as event.
}

launchpolicy {
   when * launch BWagent with name = stackBuilder.
}
```

Figure 6.1: A MAS file that connects agent `stackBuilder` to the Blocks World

### 6.2.1 Environments

The first section of a MAS file is the **environment section**. This part of a MAS file consists of a single use clause that specifies the environment that should be used. As the platform expects an

interface file to be a jar file that implements the Environment Interface Standard (EIS), a path to a jar file on the file system should be provided (between quotes). The path can be an absolute reference or specified relative to the main project folder where usually the MAS file also is located. If an environment use clause references an existing environment interface file, the environment interface is loaded automatically when a multi-agent system is launched.

The GOAL platform uses and requires a specific type of interface for connecting to environments called the Environment Interface Standard (EIS) [3, 4]. EIS provides a toolbox that facilitates the development of interfaces for connecting agents to environments. It also provides support to manage the interaction of agent platforms such as GOAL with environments. For example, it provides support for requests to pause an environment. Sometimes interface and environment are combined. The Blocks World is an example of such a combination. When creating the interface to this world, the platform also automatically launches the environment itself, i.e., the graphical interface displaying the current block configuration. The Blocks World for Teams (BW4T) environment, however, has an interface implementation that is separate from the environment itself. The environment is a server that agents can connect to using the interface.

**Environment Initialization** It is useful to configure an environment when it is launched. To do this, several **configuration parameter**s can be specified by using the **`with`** keyword followed by a comma separated list of key-value pairs of the form `key=value`. The Blocks World supports only one parameter: The `start` parameter.[1] We already saw an example how to configure the Blocks World. In Section 5.4 we provided a representation of an initial configuration in the form of a list of numbers. Instead of a list we can also specify a path to a configuration file:

```
use "blocksworld-1.1.0.jar" as environment with start="bwconfigEx1.txt".
```

**MAS Without Environment** The environment section is *optional* and there does not need to be an environment section in a MAS file. Agents need not be connected to and can run without an environment. If no environment section is present, agents will not be connected to an entity in an environment but still can be run (as any other program) as part of a MAS. This allows for a "pure" form of programming with cognitive states (cf. Chapter 5). The MAS file for the Coffee Factory MAS in Chapter 8 does not have an environment section, for example. But even when an environment has been specified, an agent does not need to be connected to an entity in that environment as we will see below.

### 6.2.2 Agent Definitions

The second part of a MAS file consists of agent definitions. An agent definition specifies which modules are used to build the agent. An agent essentially is a set of modules where some module have been given a specific function. An agent definition has a name *id* and consists of use clauses that specify which modules should be used for the main decision-making of the agent, for event processing, and for initialization. The use case of a use clause indicates the function of the module: it must be either an **`init`** module for initialization, an **`event`** module for event processing, or a **`main`** module for decision-making (see also Chapters 1 and 7). For example,

```
define BWagent as agent {
   use stackBuilder as main.
}
```

is an agent definition with name `BWagent` that specifies that the module `stackBuilder` should be used as main module. The agent does not have a module for event processing nor for initialization. (See Section 2.4 for an example of an agent definition with an init module and below for

[1] Check the environment documentation to find out about the initialization parameters that are supported by an environment.

examples using an event module.) Note that adding the keyword **module** in a use case is optional; e.g., we could have also written **as** ***main module***.

Each agent definition must be referenced in the launch policy section that we discuss next. In our example, the reference `BWagent` needs to be used in at least one launch rule.

### 6.2.3 Launching Agents

The third part of a MAS file consists of a launch policy. A launch policy consists of **launch rule**s. Launch rules are used for creating agents. Agents can be created either by an *unconditional* launch rule when a multi-agent system is created, or by a *conditional* launch rule when a controllable entity in the environment becomes available.

**Connecting Agents to Entities** A **conditional launch rule** is triggered when a controllable entity becomes available in an environment and is used for connecting an agent to a controllable entity. For example, the launch rule

```
when * launch BWagent with name=stackBuilder.
```

is triggered whenever a controllable entity becomes available. This rule launches an agent for any entity that becomes available in the environment. The $\star$ in the condition of the rule matches with any entity. If an entity becomes available in the environment, the launch rule triggers the creation of a new agent and connects it to the entity. The agent definition `BWagent` will be used to create the agent.

If a MAS has a use clause for an environment, the launch policy of that MAS file must have at least one unconditional launch rule for connecting agents to entities in that environment.

**Naming Agents** The default **baptising policy** for naming agents is to use the name of the agent definition. Agents created using the agent definition `BWagent` would be called `BWagent`, `BWAgent1`, `BWAgent2`, etc.[2] Note that a launch rule may be applied more than once and the convention then is to add consecutive numerals to the name. The example launch rule above, however, adds a **name constraint** to the rule by using the **name** keyword. A **name** constraint can be used to provide the agent that is created with a different name in two ways. One way is to specify another name explicitly that then will be used instead of the name of the agent definition. Another way is to use the wild card symbol $\star$ and add the constraint **name** = $\star$. In that case the name of the environment entity will be used to name the agent. Using this constraint makes it easier in an environment with many controllable entities to identify which agent controls which entity in an environment.

**Launch Conditions** There are three kinds of conditions of a conditional launch rule. The wild card condition $\star$ that we have seen triggers a rule each time that an entity becomes available. A rule with a condition **type** = *id* is only triggered if an entity that has type *id* becomes available, where *id* is a type of entity available in the environment. Finally, a rule with condition **name** = *id* is only triggered if an entity with name *id* becomes available.

**Unconditional Launch Rules** Agents can also be created without connecting them to an environment. **Unconditional launch rules** create agents when a multi-agent system is launched. An unconditional launch rule simply is a launch instruction of the form **launch** *id* where *id* is the name of an agent definition. As before, launch constraints can be added to these rules.

[2]The identifier used to name the agent needs to be a valid constant in the knowledge representation language that is used to represent the agent's state. If this is not the case, the identifier may be mapped onto a valid constant automatically. For example, using a name that starts with a capital such as `BWAgent` will be automatically put between single quotes and mapped onto `'BWAgent'` when using Prolog.

To illustrate unconditional rules we use the Coffee Factory MAS because it is not connected to an environment. We will discuss this example in more detail in Chapter 8 which is about communication between agents.

```
define coffeemaker as agent {
   ...
}

define coffeegrinder as agent {
   ...
}

launchpolicy{
   launch coffeemaker.
   launch coffeegrinder.
}
```

This example contains two unconditional launch rules. All unconditional launch rules are applied when a MAS is launched before the MAS starts to execute. In our Coffee Factory example, the first launch rule creates an agent named `coffeemaker` using the agent definition with the same name and the second launch rule creates an agent named `coffeegrinder`.

Agents that are not connected to an environment can also be useful if there is an environment and other agents are connected to entities in that environment. These agents even though not connected to an environment can communicate with other agents that are connected to an environment. Such an agent may be used, for example, as a central point of contact. It can collect all available information about the environment from agents that are connected to that environment. Because this agent maintains a global view it can be used to manage and instruct other agents what to do.

**Counting Constraints** There are two types of constraint that can be added to a launch rule to control the number of times that a rule is applied and the number of agents that are created when a rule is applied. The **number** = $n$ constraint can be used to create $n$ agents at the same time when a launch rule is applied. For example, if we want three coffee maker agents in the Coffee Factory MAS, the following launch rule will achieve this:

```
launchpolicy{
   launch coffeemaker with number = 3.
   launch coffeegrinder.
}
```

The first launch rule will create three agents using the agent definition `coffeemaker`. The names for these agents would be `coffeemaker`, `coffeemaker1`, and `coffeemaker2`. 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. The **number** constraint can also be used in combination with conditional launch rules. In that case all the agents that are created will be connected to the *single* entity that triggered the rule. Each of these three agents can ask the entity what it "sees" (the entity sends its current percepts when asked). Moreover, if any of these agents performs an environment action, the entity will be requested to perform it.

The **max** = $n$ constraint is used to control the number of times that a launch rule will be applied. 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. Each agent that is created by a rule counts as one rule application. This means that if a **number** = $m$ constraint has been specified as well and $m$ agents are created when a rule is applied, the count of how often the rule has been applied is raised by $m$. The **max** constraint has precedence over the **number** constraint. Even if $m > n$ therefore never more than $n$ agents will be created when a launch rule with these constraints is applied. A rule that has reached its maximum number of applications can no longer be applied.

**Rule Order** Rules in a launch policy section are applied in the order that they appear. 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, consider the order of the following two launch rules for a MAS that uses the Elevator Simulator:

```
launchpolicy{
   when type = car launch elevator with max = 3.
   when name = car0 launch elevator.
}
```

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 be applied. By reversing the order of the rules it is always possible to connect four agents to a carriage, that is, if one of them is called `car0`.

## 6.3 The Blocks World with Two Agents

We have already seen that it is possible to connect more than one agent to an entity by using the **number** constraint. This constraints allows to connect more than one agent of the same type to an entity. It is also possible to connect more than one different type of agent to a single entity by adding more than one launch instruction to a conditional launch rule. As an example, we will connect the `stackBuilder` agent of Chapter 5 to the gripper in the Blocks World as well as another agent that we call `tableAgent`. We extend the MAS file of Figure 5.1 with a new agent definition for `tableAgent` and add a launch instruction to the launch rule to also connect this agent to the gripper as follows:

```
use "blocksworld-1.1.0.jar" as environment with start="bwconfigEx1.txt".

define stackBuilder as agent {
   use stackBuilder as main.
}

define tableAgent as agent {
   use clearBlocks as main.
}

launchpolicy {
   when * launch stackBuilder, tableAgent.
}
```

Figure 6.2: A MAS file that connects two agents to the gripper in the Blocks World

The environment section uses a configuration file instead of a list of blocks. The agent definition for `stackBuilder` is the same as in the MAS file of Figure 5.1. The most important change, however, is that the launch rule now adds both `stackBuilder` and `tableAgent` to the gripper. In order for this to work, we also added a new agent definition for the `tableAgent` which uses the following module `clearBlocks`:

```
use bwknowledge as knowledge.
use bwbeliefs as beliefs.
use clearBlocks as goals.
use bwmove as actionspec.

exit = nogoals.
```

```
module clearBlocks {
   if bel(on(X, Y), block(Y)) then move(X, table).
}
```

This module is very similar to the `stackBuilder` module and reuses most components of that module but with two changes. First, the decision rule of the module moves blocks to the table that sit on top of another block. Second, the `clearBlocks` module uses the `clearBlocks.pl` file as a goal, which provides the agent with a goal that all blocks should sit on the table:

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

If we run the new MAS, both agents are connected to the gripper in the Blocks World. This means that both agents gain control over the gripper and are able to move blocks. The first thing you should note is that if two agents are connected to the same gripper that neither of these agents has *full control* over the things that happen in the Blocks World any more. This does not necessarily mean that the agents will not achieve their goals. As you may find by running the agents several times, both agents come to believe most of the time that they achieved their goals and (therefore) terminate successfully. Agents run in different threads and scheduling by the OS determines how much time each agent gets, which explains why there are different runs.

Sometimes both agents achieve their goals, as witnessed by the following run:

```
[stackBuilder] performed 'move(b1, table)'.
[tableAgent] performed 'move(b1, table)'.
[tableAgent] performed 'move(b2, table)'.
[tableAgent] performed 'move(b4, table)'.
[stackBuilder] performed 'move(b2, table)'.
[tableAgent] 'on(b1,table) , ...' has been achieved and removed ...
[tableAgent] performed 'move(b6, table)'.
[stackBuilder] performed 'move(b4, b3)'.
agent 'tableAgent' terminated successfully.
[stackBuilder] performed 'move(b5, b2)'.
[stackBuilder] performed 'move(b1, b5)'.
[stackBuilder] 'on(b1,b5) , ...' has been achieved and removed ...
[stackBuilder] performed 'move(b6, b4)'.
agent 'stackBuilder' terminated successfully.
```

But more often than not even if the agents believe they have achieved their goals, at least one of the agent failed to achieve its goal. Here's an example run:

```
[tableAgent] performed 'move(b1, table)'.
[stackBuilder] performed 'move(b1, table)'.
[stackBuilder] performed 'move(b2, table)'.
[tableAgent] performed 'move(b2, table)'.
[stackBuilder] performed 'move(b4, b3)'.
[tableAgent] performed 'move(b4, table)'.
[stackBuilder] performed 'move(b5, b2)'.
[tableAgent] 'on(b1,table) , ... ' has been achieved and removed ...
[tableAgent] performed 'move(b6, table)'.
[stackBuilder] performed 'move(b1, b5)'.
agent 'tableAgent' terminated successfully.
[stackBuilder] 'on(b1,b5) , ... , on(b4,b3) , ... ' has been achieved ...
[stackBuilder] performed 'move(b6, b4)'.
agent 'stackBuilder' terminated successfully.
```

This is a run where neither of the agents achieved its goal. Note that before the `tableAgent` performs its last action the `stackBuilder` agent moves block `5` on top of `2` and the `tableAgent` moves block `4` back to the table again right after `stackBuilder` moves it on top of block `3`. Still both agents are said to have 'terminated successfully'. What has gone wrong is that both agents *believe* they have achieved their goals but their beliefs are false as they do not correspond with the actual block configuration. They have both acted as if they were in full control but when another agent also makes changes to a block configuration, an agent can no longer maintain a correct representation of this configuration just by starting with a correct representation of the initial configuration and knowledge of the effects of the actions that the agent itself performed. An agent that does not have full control can only maintain an accurate representation of the current configuration if it also takes the changes made by other agents that it *perceives* into account. If we give both agents control over the gripper in the Blocks World, each agent needs to sense and process percepts received from the environment in order to make sure its beliefs are true and it can achieve its goals.

## 6.4 Processing Percepts and the Event Module

By **connecting** an agent to an environment it gains **control** over (part of) that environment. An environment makes available **controllable entities** and an agent can be connected to such an entity (see also Chapter 1). An entity can "see" certain things in the environment and perform certain actions. By connecting to an entity an agent obtains access to the capabilities of that entity: It can perceive what the entity can perceive and can do what the entity can do in the environment. An agent that is connected to an entity receives percepts to "see" what the entity "sees" and request the entity to perform the actions it can perform. As we have seen, in the Blocks World our agent was connected to a virtual gripper that could instantly move blocks. In the Blocks World the agent receives **percept**s about the configuration of blocks. We will now fix our two-agent Blocks World MAS and look at how we can make agents process these percepts.

**Percepts** The interface that connects agents to the gripper in the Blocks World provides percepts of the form `on(X,Y)`. We introduced this predicate ourselves in Chapter 3 to represent which block is on top of another block or on the table. The percept has exactly the same meaning. The Blocks World generates a percept for each fact (instantiation of) `on(X,Y)` that holds in the current block configuration. This means that the set of percepts that an agent receives from the Blocks World provides accurate and complete information about that 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 an agent requests percepts from the entity it is connected to, moreover, such a complete set of percepts is sent by the environment. For example, for the initial block configuration of Figure 3.1, the agent would receive the following list of percepts:

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

**The Percept Base** Percepts that are received from an environment are automatically inserted in the **percept base** of the agent (see Figure 1.3). Each time that percepts are received this percept base is first cleared from all content. The percept base thus always contains the most recently set of percepts received from an environment. Percepts are represented in the knowledge representation language that is used by the agent.

Percepts represent “raw data" received from the environment that the agent is connected to. For several reasons an agent cannot simply add the new information to its belief base.[3] One reason is that the received information more often than not is *inconsistent* with the information that is stored in the belief base of the agent. For example, receiving a percept `on(b1,table)` that informs the agent that block `b1` is on the table conflicts with a fact `on(b1,b2)` in the belief base that block `b1` is on block `b2`. The percept cannot be simply added to the agent's belief base as that would result in conflicting beliefs. The agent's beliefs need to be updated instead and the fact `on(b1,b2)` needs to be removed first before the fact `on(b1,table)` is added to the belief base. A more pragmatic reason is that percepts may need further processing and interpretation. It is more useful, for example, to insert the conclusion that there is a wall at a certain location that blocks the agent from moving forward than to insert into the agent's beliefs that the agent bumped into a wall. In the Wumpus World a percept that the agent bumped into a wall also means that the agent failed to move forward and the agent should make sure it still believes that it is located at the position it was before it performed a move action.

**Inspecting the Percept Base** A special operator **`percept`**`(qry)` is available for inspecting the percept base of an agent, where `qry` is a valid query of the KR the agent uses. For example, an agent can query whether a percept has been received that block `1` is on block `2` by the cognitive state query **`percept`**`(on(b1,b2))`. The **`percept`**operator works similar to the **`bel`** operator but queries `qry` are evaluated only on the percept base and not also on the agent's knowledge base.

**Rules for Processing Percepts** An agent processes events by means of rules. We also call rules that are used for event processing **event rules**. For event processing, however, typically a type of rule is used that is different from the **`if`** `csq` **`then`** `action`-rule that we have used for decision-making in Chapter 5. The reason is that an **`if`**-**`then`**-rule only is applied for one instance of the rule. For processing a set of percepts of the same form such as `on(X,Y)`, however, we want to be able to process *all* of these percepts. We can do so by means of rules of the following form:

```
forall csq do action.
```

where `csq` is a cognitive state query and `action` is an action. A **`forall`**-**`do`**-rule is applied for all instances for which the query `csq` and the pre-condition of `action` holds.

To process all of the percepts received from the Blocks World we want to insert all of these into the agent's belief base. We also check whether the agent does not already believe the fact, as in that case there is no need to update the agent's beliefs (which saves work). The following rule can be used for this:

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

This rule is applied for all instances that match the query of the rule and inserts a percept `on(X,Y)` in the agent's belief base only if it is not already believed. Of course, we also need to remove *all* incorrect beliefs from the agent's belief base that do not match with any of the percepts. Recall that the percepts provide *complete* information about the current block configuration. Any fact in the agent's belief base that does not match a percept therefore is incorrect and must be removed. We can use the following rule for this:

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

[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 could have been used to directly modify the belief base of the agent. But simply adding percepts to an agent's belief base is not a good updating procedure.

This rule checks whether a fact `on(X,Y)` is believed but is not perceived by the agent and removes it in that case. The rule is applied for any fact that satisfies the rule's condition.

Rules that use the **`percept`**operator are also called **percept rules**.

**Event Module** Rules for event processing should be called from the event module. the main function of an event module is to *process events and update the agent's cognitive state accordingly.* The event module is called automatically after an agent finished a decision cycle when percepts are received or other events have happened that trigger the event module (see next section below for details). We therefore create a new module that contains the two percept rules for the Blocks World. The code for this module can be found in Figure 6.3. The Prolog file `bwknowledge` is needed to declare the `on/2` predicate.

```
use bwknowledge as knowledge.

module bwevents {
   forall bel( on(X,Y) ), not( percept( on(X,Y) )) do delete( on(X,Y) ).
   forall percept( on(X,Y) ), not( bel( on(X,Y) )) do insert( on(X,Y) ).
}
```

Figure 6.3: Module for processing percepts from the Blocks World

There is an important difference between a module that is used as event module and a module that is used as main module. By default a module evaluates rules in the order they appear and applies only the first applicable rule. An event module evaluates and *applies all applicable rules* in the order that they occur in the module. Each rule of a module used as event module thus is evaluated. This facilitates the processing of percepts and events as an agent will want to process *all* events that contain potential information for making the right action choice. This type of rule evaluation ensures that the newly received information of blocks is inserted into the belief base of the agent (by the first percept rule) *and* the old information is removed (by the second percept rule).

Finally, we need to use the module `bwevents` and indicate that it should be used for processing events. To indicate that we want to use the module `bwevents` as event module, we need to add a use clause to the agent definitions of the MAS file for the Blocks World that specifies that the module `bwevents` should be used as event module. The module can be used by both agents.

```
use "blocksworld-1.1.0.jar" as environment with start="bwconfigEx1.txt".

define stackBuilder as agent {
   use stackBuilder as main.
   use bwevents as event.
}

define tableAgent as agent {
   use clearBlocks as main.
   use bwevents as event.
}

launchpolicy {
   when * launch stackBuilder, tableAgent.
}
```

Figure 6.4: A MAS file with two agents that use an event module

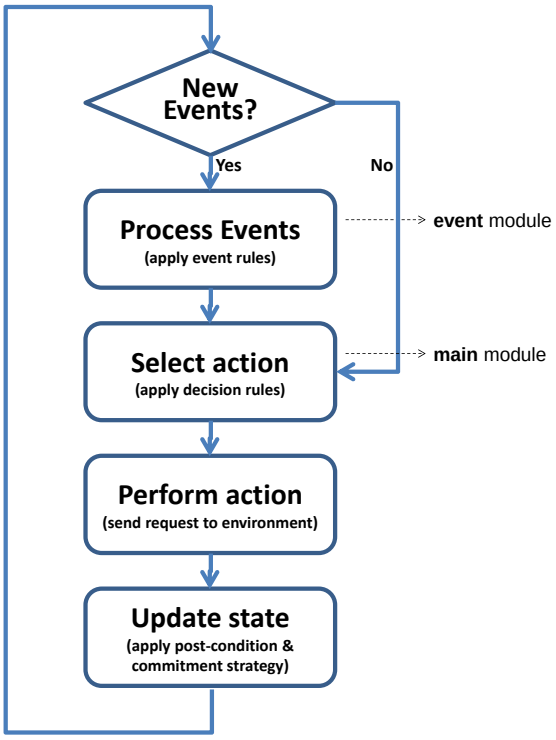


Figure 6.5: Execution Cycle

## 6.5 The Execution Cycle of an Agent

If an agent has a module for initializing the agent (init module), when the agent is launched this module is executed first. An init module can be used to initialize the cognitive state of an agent and to process percepts that are *sent only once* to an agent (such as information about a map that can be navigated by an entity). Thereafter the agent executes a fixed cycle. The approach for handling events, including percepts, is part of this execution cycle of an agent. Figure 6.5 illustrates the steps that are part of the cycle.[4]

**Step 1: Check for Events** At the start of each cycle, events are automatically collected and a check is performed whether any new events are available. As we saw in Chapter 1, receiving percepts, receiving messages, or performing an action all count as events. Events are new if they were not also received or occurred in the previous cycle. The percept base of the agent is updated by adding new percepts and removing old percepts that are no longer available. Similarly, the message base of the agent is updated by adding new messages and removing all messages that were received in the previous cycle. Another way of thinking about this is that the percept and message base are always cleared first and then filled again with events received at the start of the cycle.

**Step 2: Process Events** If a new event is available, the agent will execute its event module. All rules in the module used as event module are evaluated in order and applied if applicable. If the agent has no event module, this step in the cycle is skipped.

The event module is used for processing events. An agent that is connected to an environment and receives percepts, should have an event module. Similarly, an agent that receives messages

[4]The execution cycle is also called a *Sense-Plan-Act* (SPA) or an *Observe-Orient-Decide-Act* (OODA) cycle.

from other agents (see Chapter 8) also should have an event module for processing these messages. The module can also be used to update the cognitive state by adopting new or reconsidering existing goals of the agent. This step should be used to update the agent's state as a preparation step for decision-making.

**Step 3: Execute Main Module** After processing events, the agent selects one or more actions to perform by executing its main module. By default, the first applicable decision rule of the module used as main module is applied. (The order that rules are evaluated can be changed by the **`order`** option of a module.)

The main module is used for specifying the action selection strategy of an agent. If an agent is connected to an environment, the main module should be used to select an external action that can be performed in that environment.

**Step 4: Request Environment to Perform Action** External actions that the agent decides to perform are sent to the environment that the agent is connected to. If an agent has no main module, this and the previous step are skipped. An agent should at least have an event or a main module as without either it would do nothing.

**Step 5: Update Cognitive State** The post-condition of an external action that is performed in the previous step is used to update the cognitive state of an agent (see Chapter 4). After updating the state, at the end of the cycle an agent removes goals have been completely achieved. This implements a blind commitment strategy (see Chapter 5).

## 6.6 The Tower World

So far we have been assuming that actions are instantaneous. We can no longer make this assumption in the Tower World (see Figure 6.6) that we introduce in this section. The Tower World is a variant of the classic Blocks World. The main difference is that in the Tower World moving the gripper, and therefore moving blocks, takes time. Actions are **durative** in this environment. Moreover, a user (you!) can also move blocks by dropping and dragging blocks using a mouse. This means an agent that controls the gripper does not have full control. This introduces some new challenges that our Blocks World agents are not able to handle.

The basic design of the Tower World is very similar to the Blocks World. 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 observe changes in the Tower World and process events very similar to the two Blocks World agents that both were connected to the gripper above. The fact that actions in the Tower World take time also provides an important reason why the agent needs to be able to sense its environment.

### 6.6.1 Specifying Durative Actions

Actions that take time are called **durative action**s. 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 when the action is performed. 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 *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

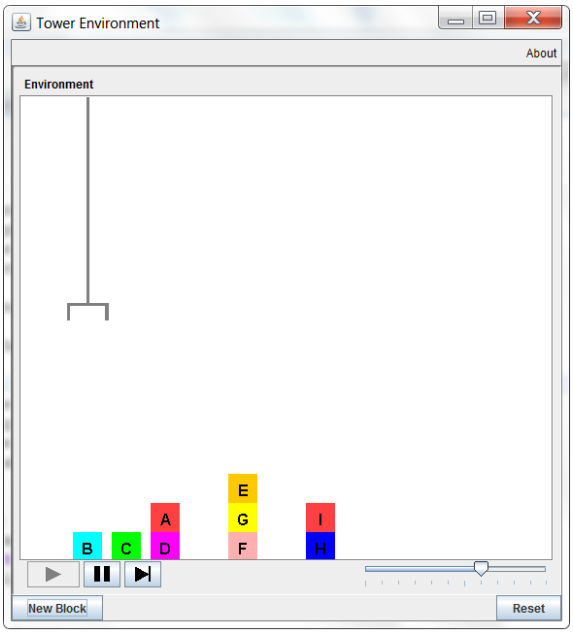


Figure 6.6: The Tower World Interface

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. If that happens, the agent should *reconsider* its actions and possibly its goals.

A GOAL agent is able to monitor progress of actions because it does not block on an action that it sends to the environment.[5] While the action is being performed by its body, i.e., an entity such as the gripper in the Tower World, the agent can monitor progress of the action by means of the percepts it receives from the environment.

**Action Specifications for Durative Actions** All actions available in the Tower World are durative. There are three actions that an agent can perform in the Tower World: An action `pickup(X)` with one parameter for picking up a block `X` with the gripper, an action `putdown(X,Y)` with two parameters for putting down a block `X` that the gripper holds on another block `Y`, and a special `nil` action without any parameters that moves the gripper back to the left-upper most corner of the graphical display of the environment.

Because these actions are durative and an agent does not block on an action when it performs it, we cannot provide post-conditions for these actions. We cannot, for example, use the effect of holding a block after picking it up as a post-condition of the `pickup` action. A post-condition is applied immediately when an action is selected. If we would use the effect as post-condition, the agent would *immediately* have a belief that the gripper holds a block while the action of picking up the block is still only in progress by moving the gripper towards the block. But this belief would simply be false as it takes time to pick up the block and only if the action is successfully completed the effect will be realized.

The solution that we provide is a simple one. Because we do not know and cannot know at the time of starting to pick up a block what the results of the action will be, we use an *empty* postcondition for the specification of the action. Instead of applying a post-condition the agent will learn over time via percepts what the effects of its (durative) actions are and by monitoring

[5] If an environment itself blocks on a request, and does not provide percepts to an agent while performing an action, there is little that an agent can do to monitor progress. The typical situation, however, is that the environment provides percepts while an action is being performed and the agent continuously receives information that enables it to monitor progress.

their progress will become aware of the resulting changes in the environment. We thus do not specify the effects of durative actions in an action specification but, instead, depend on percepts that the agent receives to conclude what happens in the environment.

Note that it is just as useful to specify pre-conditions for durative actions as it is for instantaneous actions. When an agent performs an action it is useful to inspect whether the 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.

Before we specify the three actions available in the Tower World, we briefly discuss the percepts that an agent receives from this world. This is information that is useful and needed for providing adequate action specifications. An agent connected to the Tower World receives three kinds of percepts:

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

It is good practice to use these same predicates for specifying the actions of the environment. In the Tower World, there are two conditions that must hold to be able to perform a `pickup` action: the block that needs to be picked up must be clear, and the gripper cannot already hold a block. We will reuse the definition of the `clear/1` predicate that we introduced in Chapter 3. We can then use the following action specification for `pickup`:

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

Although the post-condition could have been left empty and we could also have simply written **`post`**`{ }`, it is better practice not to do so. Instead we have used **`true`** as post-condition to indicate that we have not simply forgotten to provide a specification of the post-condition.

We can use the following action specification for `putdown`:

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

Because the `nil` action can always be performed, the action specification for this action is very simple. This action has **`true`** as pre-condition and as post-condition.

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

The action `nil` is somewhat unusual and rather different from the actions that we have seen before in Blocks World-like environments. This action moves the gripper back to a particular position in the graphical interface that comes with the Tower World (see Figure 6.6). This action is of little use for achieving our main goal: creating particular block configurations. Also note that the predicates in terms of which percepts are provided by the environment to the agent are unrelated to the exact position of the gripper. The idea is that when an agent has nothing to do any more to achieve its goals we move the gripper back into its initial starting position. This then will provide an indication that the agent is ready.

### 6.6.2 Percepts in the Tower World

The percepts that an agent receives from the Tower World are the same predicates `block` and `on` that we have already seen in the Blocks World. The Tower World introduces one new predicate `holding(X)`. The `holding` predicate has one parameter; if `holding(X)` holds for some block `X`, the gripper is holding that block, otherwise it is not holding any block.

As in the Blocks World, we can also assume **full observability** for the Tower World: Every time percepts are received they provide *correct and complete* information about the state, i.e. the configuration of blocks. Because we can assume this, and we know that the gripper is either holding a block or not, we can apply the Closed World assumption to the predicate `holding`. In other words, that is, if the agent does not believe the gripper is holding a block, we can conclude that the gripper is holding no block.

For processing percepts in the Tower World, we can reuse the same percept rules for the Blocks World that we introduced above for the `on/2` predicate. We need to add rules for the `block/1` and `holding/1` predicates. The percept rules for these predicates follow the exact same pattern that we used for the `on/2` predicate above. We use one rule to add a percept to the agent's beliefs if the agent does not already believe the fact, and use a second rule for removing a belief that does not match with what the agent perceives. We call the module that we want to use as event module `twevents`.

```
use tower as knowledge.

% module for processing Tower World percepts; assumes full observability
module twevents {
   forall bel( block(X) ), not(percept( block(X) )) do delete( block(X) ).
   forall percept( block(X) ), not(bel( block(X) )) do insert( block(X) ).

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

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

The `tower.pl` file that is used as knowledge in the module re-uses the definition of the `tower/1` predicate from the Blocks World. The definition of the `clear/1` predicate needs to be changed, however, because a block that is being held by the gripper can also not be said to be clear. We add a clause to exclude this. We also add a predicate `above/2` that is useful for defining when a block is misplaced. Finally, we need to declare the predicates `block/1`, `on/2`, and `holding` that are used but not defined in the file.

```
:- dynamic block/1, on/2, holding/1.

% Assume there is enough room to put all blocks on the table.
% This is actually only true for up to 13 blocks in the Tower World.
clear(table).
clear(X) :- block(X), not( on(_, X) ), not( holding(X) ).
clear([X|_]) :- clear(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]).
```

## 6.7 Performing Durative Actions

There are several issues that we need to consider when programming an agent that performs durative actions. One question we need to think about is what happens when an environment has been requested to perform a durative action and the agent requests the environment to perform another action. Recall that an agent does not block on a durative action until it completes, and should not do so, because it is important to *monitor the progress* of the action. If, for example, in the Tower World the action `putdown(X,Y)` is not feasible 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 `putdown(X,Y)` and initiate another, feasible action instead. In this specific example, it would make sense to send a new command to put block `X` on the table instead by means of performing the action `putdown(X,table)`. This works in the Tower World because durative actions that are still in progress are *cancelled* if another action is performed. The durative action is ended and the gripper immediately continues with performing the new action.

One example of cancelling is worth mentioning in particular: actions typically cancel actions of the *same type*. As a rule, actions are of the same type if the action name is the same. For example, the action `putdown(X,Y)` where `Y`$\neq$`table` can be cancelled by the action `putdown(X,table)` provides. Another example is provided by the `goto` action in UNREAL TOURNAMENT. If this action is performed continuously (the agent very quickly and repeatedly instructs a bot to perform this action) with different parameters, the bot being instructed to do so actually comes to a halt. There is simply no way to comply with such a quickly executed set of instructions for the bot. But what would you expect to happen if an agent performs *exactly the same action* again while already performing that action?

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

An important alternative, however, which often makes sense is to perform actions *in parallel*. Examples where this is the more reasonable approach abound. In UNREAL TOURNAMENT, for example, a bot should be able to and can move *and* shoot at the same time. A robot should be able to and usually can move, perceive *and* speak at the same time.

## 6.8 Deciding to Perform a Durative Action

Because durative actions may cancel each other, it is important that an agent continues to perform the same action as long as it has not been completed yet. An agent would not make any progress if it continuously instructs the entity that it controls to do different things. To prevent this from happening an agent must have some focus on achieving one thing at a time. At the same time, we also need to take into account that an agent may need to *reconsider* its choice of action if that action is no longer feasible.

### 6.8.1 Creating Focus

In the Tower World there is an elegant solution for creating a focus on completing a particular action based on the fact that *the gripper can hold at most one block at a time*. The idea is to focus on a single block that the agent wants to hold, and if the agent holds it, to focus on the place where to put it down. This will provide the agent with the focus that is needed. Any actions that are unrelated to the block of the agent's current focus then can simply be ignored.

There is a simple strategy for making an agent focus: *Have the agent adopt a single goal that indicates the current focus.* If needed, goals can be reconsidered and dropped, which also addresses our concern that sometimes the focus on an action needs to be reconsidered. In the Tower World we can implement this strategy by adopting a goal to hold a block. Such a goal is a good candidate for creating the right kind of focus in this environment. It is important in order to maintain focus,

of course, that we ensure that an agent only adopts a *single* goal to hold a specific block and no others. That is, we should have at most one goal of the form `holding(X)` in the agent's goal base at any time. We call such goals **single instance goals**. There is a simple rule template that we can use for adopting a single instance goal `sig` to create focus:

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

In the Tower World, 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 misplaced (and clear so we can move it). Here we use a slightly more general notion of a misplaced block: A block is *misplaced* if that block obstructs access to another block that can be put into position or no block can be put into position and at least one block needs to be moved to the table. We use the following macro definition:

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

We thus obtain two rules by instantiating the rule template above where the first rule (making a constructive move) should be preferred over the second.

```
if not(goal(holding(_)))), constructiveMove(X,Y) then adopt(holding(X)) .
if not(goal(holding(_)))), misplaced(X), bel(clear(X)) then adopt(holding(X)) .
```

An important decision remains to be made: In which module should we add these rules? Above we observed that the event module should be used to not only process events but also should be used to update an agent's state. As the rules for adopting a goal above only update the agent's state, as a rule of thumb, we therefore should add them to the module used as event module. Note also that even though an event module applies all rules (in order), at most one of the rules will be applied because if the first rule adopts a goal, the query of the second rule no longer holds.

### 6.8.2 Deciding What to Do 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. Given that we may assume that we have implemented the single instance pattern above and always will have at most a single goal of holding a block at any time, the selection of actions to be performed in the environment turns out to be quite simple. The more difficult part that we also still need to implement is the strategy for reconsidering the goal to hold a specific block if unexpected events happen (i.e. a user moves blocks) which make such a goal no longer beneficial.

If we can assume that the agent at most has one goal of holding a block and the agent only has such a goal if it is feasible, the choice which action to perform is easy. If the agent has such a goal and wants to hold a block, the agent should simply pick it up. Then, if the agent is holding a block, it should, first, try to put it in position, but, otherwise, put it somewhere on the table. Otherwise, the agent should move the gripper back to its initial position. This strategy can be implemented by decision rules as follows:

```
use tower as knowledge.
use towergoal as goals.
use tower as actionspec.

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

module towerbuilder {
   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 `tower` file used as knowledge is the Prolog file discussed above; `towergoal` specifies some block configuration as goal as we have seen before; and, the `tower` file used as action specification collects the three action specifications that we provided above for the Tower World.

The code introduces one new language feature: **nested rules**. Instead of an action a rule can also have a rule section as head. Such rules are of the form

```
if csq {
   ...
}
```

where the dots ... must be replaced by one or more rules.

### 6.8.3 Reconsidering Goals

The agent that we have developed so far is able to achieve its goal. It is not robust, however, to actions of a user that interfere with the agent's goals. The final part that we need to design therefore is a strategy for *reconsidering* a goal to hold a block. 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 a goal are sufficient for operating robustly in the dynamic Tower World. They would not always generate a *best response* to changes in the environment, however. 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. The following rules implement this reconsideration strategy:

```
% 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 goal-a( tower([X|T]) ) 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? As we have seen before, rules that manage the state of an agent are best placed in a module used as event module. This also ensures that the agent will drop a goal if any of the reasons for doing so apply because

an event module applies all applicable rules. It also ensures that the agent first updates its state and only after doing so decides on an action to perform in the environment. Since the rules for dropping a goal only affect the state of the agent, based on this design principle they should also be put in the module used as event module. We should, moreover, place them before the rules for adopting goals so that after dropping a goal the agent will adopt a new goal immediately.

This concludes the design of our agent for the dynamic Tower World.

## 6.9 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. Full observability is always *relative to the representation of the environment*. An agent will receive all information there is about which blocks there are, where a block sits, and whether the gripper is holding a block. In the Tower World, however, the agent does 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 block `b` cannot be moved constructively. In terms of efficiency, because such aspects cannot be observed the action selection of the agent may not be optimal in a dynamic environment such as the Tower World. Even though the agent does not have access to everything there is to know about the world, it has full access to the aspects that it can observe: The agent has full observability (relative to these aspects).

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

The Wumpus World, however, is a rather special world again because there is only one agent and the environment itself is *static*. By fully exploring the finite Wumpus World, which is possible if all grid cells can be reached, the agent can construct a complete representation of its environment. In dynamic environments, which most gaming environments, such as for example Unreal Tournament are, it is impossible to ever construct a complete and accurate representation of the environment. An agent's decision making gets even more involved then, as its decisions must be based on incomplete information.

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

## 6.10 Percept Types

The interface that we use for connecting to environments distinguishes four types of percepts [3, 4]: percepts that are **send once** when the agent first connects to the environment, percepts that are **send always**, percepts that are **send on change** when a feature of the environment changes, and percepts that are **send on change with negation**. For each of these percept types we provide a templates for processing it.

**Send Once** Percepts that are received only once when the agent is first connected to an environment require an agent to make the information persistent because percepts are removed each new cycle. If an agent needs to have access to the percept information that is sent only once, it needs to store the information in its belief base. An example of a percept that is sent once is a percept that provides information about the structure of a map that an entity is on. This kind of information is provided, for example, by the Blocks World for Teams and UNREAL TOURNAMENT environments. A template for processing a send once percept p($\vec{t}$) is:

```
forall percept(p(t)) do insert(p(t)).
```

The template is straightforward. What is more important is to add the template to the right type of module. Whenever an environment provides send once percepts, it is useful to create a module that is used **as init module** and add it to the agent definition in the MAS file. The percept rules for processing send once percepts should be added to this module that is only executed once when the agent is launched. All other percept rules should be added to a module used **as event module**.

**Send Always** We have already seen a template for processing send always percepts. A percept that is received always when the environment feature is present requires an agent to both add information to its belief base when it is available and remove it again when it is no longer available. The template for a send always percept p($\vec{t}$) therefore consists of a rule for adding and a rule for removing a belief.

```
forall percept(p(t)), not(bel(p(t))) do insert(p(t)).
forall bel(p(t)), not(percept(p(t))) do delete(p(t)).
```

**Send on Change** A send on change percept is received each time when an environment feature changes but not when the feature does not change. An example is a state percept that provides information about the state of movement of an agent. A percept `state(moving)` is received when the agent starts moving and a percept `state(arrived)` might be received when the agent arrives at its destination. The template for this percept type records the change by removing the old information and updating it with the new percept information.

```
forall bel(p(old)), percept(p(new))  do delete(p(old)) + insert(p(new)).
```

It is important to realize that the template assumes that a belief about p($\vec{old}$) has been inserted in the agent's belief base. Use the init module for making sure some initial belief about p($\vec{old}$) has been inserted. Note that if no belief about p($\vec{old}$) is present, the rule cannot be applied.

**Send on Change with Negation** A send on change with negation percept is received when an environment feature changes. An example of a send on change with negation percept is the `in(Room)` percept of the Blocks World for Teams environment. If an agent enters a room `'RoomA1'`, for example, it receives the percept `in('RoomA1')`. If it leaves the room again, however, it does not immediately enter a new room but ends up in the corridor. Instead of a new `in(Room)` percept the agent receives a percept **not**`(in('RoomA1'))`. The template for this percept type has two rules. One for inserting information when a 'positive' percept is received and one for removing information when a 'negative' percept is received.

```
forall percept(p(t)) do insert(p(t)).
forall percept(not(p(t))) do delete(p(t)).
```

Note that this template does not make any assumptions about what has been inserted into the agent's beliefs (neither of the rule conditions inspects the belief base). As all other percept types except for the send once percept, percept rules for send on change with negation percepts should be added to an event module.

## 6.11 Summary

As we saw in Chapter 5, if the agent has full control and actions are instantaneous, it is possible to update an agent's beliefs using knowledge about the effects of the actions that the agent performs. If the agent's initial beliefs represent the initial environment state, the agent's beliefs will also represent the correct environment states after performing actions. Agents hardly ever have full control, actions are usually durative and may fail, and initializing the agent's beliefs to reflect the initial environment state is inflexible, however. It therefore is better to process percepts that an agent receives from its environment.

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:

1. the agent does not have *full control*, and events initiated by the environment may occur (*dynamic environments*),
2. *other agents* also perform actions (another kind of dynamic environment),
3. the environment is not *fully observable* but e.g. needs to be explored, or
4. 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*.

We emphasize how important it is for the design of an agent and a multi-agent system to investigate and gain a proper understanding of the environment to which agents are connected.

As a rule of thumb, *the design of a multi-agent system should always start with an analysis of the environment in which the agents act.* It is very important to first understand which aspects are most important in an environment, which parts of an environment can be controlled by agents, and which observations agents can make in an environment. Assuming that an environment interface is present, moreover, two things are already provided that must be used to structure the design of agents. First, a basic vocabulary for representing the environment is provided in terms of a language used to represent and provide percepts to an agent. Second, an interface provides a list of actions that are available to an agent, which, if they are documented well, also come with at least an informal specification of their preconditions and effects. It is only natural to start analysing this interface, the percepts and actions it provides, and to incorporate this information into the design of an agent program. More concretely, as a rule it is best to start developing an agent by *writing percept rules and action specifications* for the agent.

## 6.12 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. [34]. Agent-environment interaction has been discussed in various other contexts. See e.g. [2, 41].

## 6.13 Exercises

### 6.13.1

1. Will the *stackBuilder* agent of Section 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 using the GOAL IDE. Select different middleware platforms in the Run menu of the IDE to run the agents and report what happens. Do the agents always perform the same number of actions?
3. Test the multi-agent Blocks World system again but now distribute the agents physically over different machines. Do the agents always perform the same number of actions?

### 6.13.2

In dynamic environments such as the Tower World it is particularly important to *test* the agent in all kinds of different scenarios that may occur. To test an agent for all the different kinds of scenarios that may occur, a thorough analysis of what can happen in an environment is needed. For the Tower World, explain why the `towerBuilder` agent we developed in this chapter 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.

# Chapter 7

# Modules

Modules are useful for structuring code. Modules are effective for handling specific situations or cases in an environment. It is useful to combine related conceptual and domain knowledge, goals, actions and rules that are relevant for handling a particular situation in a module. Modules can also be used to hide some of the details of performing some action. If it is not important to know how to build a tower, for example, these details can be put inside a module. Modules support reusability.

Modules can be used as any other action in a rule. By applying a rule that calls a module instead of that it performs an action, a module is entered and executed. Modules also offer options for controlling the evaluation of rules, the execution of an agent, and for providing a focus on what to achieve next.

## 7.1 Rule Evaluation Order

Program rules may generate multiple action options. In the example of Figure 7.1 the first rule is applicable by instantiating `X` with `b4` and `Y` with `b3`. It generates the option to do `move(b4,b3)`. There are two applicable instantiations of the second rule. It can generate two options: `move(b1,table)` and `move(b4,table)`.

```
use bwknowledge as knowledge.
use bwgoals as goals.
use bwmove as actionspec.

exit = nogoals.

define constructiveMove(X,Y) as a-goal(tower([X,Y| T])), bel(tower([Y|T])).
define misplaced(X) as a-goal(tower([X| T])).

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

```
beliefs:
on(b1,b2). on(b2,b6). 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).
```

Figure 7.1: Module `stackBuilder` and Beliefs and Goals of a Cognitive State

Which of these action options are actually performed by the agent depends on the **rule evaluation order** that is used to execute a module. By default, rules that appear in the main module are evaluated in the order they appear in a module from top to bottom. The first rule that is applicable, i.e., generates an option, is applied. In our Blocks World example of Figure 7.1, the first rule is applied and the constructive `move(b4,b3)` is selected. This is a **linear order** style of rule evaluation. The rule evaluation order of the init and event modules is different. Instead of just the first rule, all rules in modules used as event or init module that are applicable are applied (see also Chapter 6). The rules are still evaluated from top to bottom but rule evaluation does not stop when the first applicable rule has been found. We call this a **linear all** style of rule evaluation.

The order of rule evaluation can be changed by specifying an **`order`** option for a module. Just as the **`exit`** option, this option can be specified after the use clauses in a module. We can, for example, add **`order`**=**`random`** in Figure 7.1 before the **`exit`** option to change to a **random** order of rule evaluation. The effect of choosing this style of evaluation is that the agent will randomly select one action out of the set of all options. In our example, the agent would randomly choose to perform either `move(b4,b3)`, `move(b1,table)`, or `move(b4,table)`. As a result, the agent becomes non-deterministic and will execute differently each time it is run, e.g.:

```
Run 1:
stackBuilder performed move(b4, b3)
stackBuilder performed move(b1, table)
stackBuilder performed move(b2, table)
stackBuilder performed move(b6, table)
stackBuilder performed move(b5, b2)
stackBuilder performed move(b1, b5)
stackBuilder performed move(b6, b4)

Run 2:
stackBuilder performed move(b1, table)
stackBuilder performed move(b4, table)
stackBuilder performed move(b2, table)
stackBuilder performed move(b5, b2)
stackBuilder performed move(b6, table)
stackBuilder performed move(b1, b5)
stackBuilder performed move(b4, b3)
stackBuilder performed move(b6, b4)

Run 3:
stackBuilder performed move(b4, table)
stackBuilder performed move(b4, b3)
stackBuilder performed move(b1, table)
stackBuilder performed move(b2, table)
stackBuilder performed move(b6, table)
stackBuilder performed move(b6, b4)
stackBuilder performed move(b5, b2)
stackBuilder performed move(b1, b5)
```

Other options available are the **`linearall`**, **`linearrandom`**, **`linearallrandom`** and **`randomall`** options.

The **`linearall`** order is used as default for modules used as event and init modules. But it can also be used for other modules. Although this is usually a bad idea, in our Blocks World example we can even use it as the evaluation order for our main module `stackBuilder`. The agent will still be able to achieve its goal. If we would use this order in our example, the agent would first apply the first rule and perform `move(b4,b3)`. It would then evaluate the second rule

and perform `move(b1,table)`. Note that it would *not* perform the action `move(b4,table)` as the first rule just constructively moved block `4` and it is no longer misplaced. But even if both actions could have been executed, only one would have been performed, as an **`if-then`**-rule is always applied for only one instantiation.

As with any linear rule evaluation order, the **`linearrandom`** order would evaluate rules in the order that they appear in a module from top to bottom. This style of evaluation will only apply the first applicable rule. But if that rule generates multiple options, one of the options will be chosen randomly. In our example, if block `2` would sit on top of block `3`, only the second rule would be applicable and generate the options `move(b1,table)`, `move(b4,table)`, and `move(b6,table)`. One of these would then be randomly chosen with the **`linearrandom`** order of evaluation. The same concept applies for the **`linearallrandom`** order.

Finally, the **`randomall`** order will apply all applicable rules as the **`linearall`** order does. This style of evaluation will evaluate the rules in random order though and if a rule generates multiple options select one at random as well. In our example, one possible order would be to first evaluate the second rule and perform `move(b4,table)` and then continue with evaluating the first rule and perform `move(b4,b3)`.

Note that as the rule order evaluation of a module by default is linear (except for modules used as init or event module), there is no need to use the **`order`**=**`linear`** option.

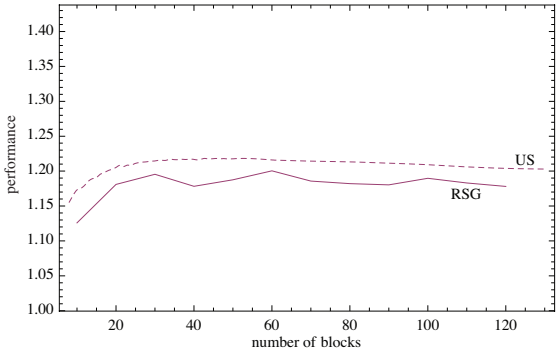


Figure 7.2: Average Performance of a Blocks World Agent that Uses Random Order

To conclude our discussion of the example Blocks World agent, we briefly look at the overall performance of an agent that uses random rule evaluation order. In Figure 7.2 the RSG line shows the average performance, i.e., number of move actions performed, of such an agent for different numbers of blocks in a Blocks World problem. This performance is compared to the performance of the simple unstacking strategy that that first moves all blocks to the table and then re-stacks the blocks to achieve the goal state. The performance of the unstacking strategy is indicated by the US line. Observe that this simple strategy never requires more than $2N$ moves where $N$ is the number of blocks. The agent that uses random rule evaluation order performs slightly better because it may perform constructive moves whenever this is possible and will at least some of the time do so instead of waiting to make constructve moves only after all blocks have been moved to the table.

## 7.2 Using Modules

We have seen how to use modules using use clauses. A use clause in an agent definition facilitates using a module as main module in Chapter 5 or as event or init module in Chapter 6. But a module can also be used as any other action in a rule. We will illustrate this by replacing the rule for handling misplaced blocks in our Blocks World example with a rule that calls a module instead that unstacks all blocks. We can re-use the module here that we used for the `tableAgent` in Section 6.3.

```
use bwknowledge as knowledge.
use bwmove as actionspec.

exit = noaction.

module clearBlocks {
   if bel(on(X, Y), block(Y)) then move(X, table).
}
```

We can now use this module in our `stackBuilder` module:

```
use bwknowledge as knowledge.
use bwgoals as goals.
use bwmove as actionspec.
use clearBlocks as module.

exit = nogoals.

define constructiveMove(X,Y) as a-goal(tower([X,Y| T])), bel(tower([Y|T])).
define misplaced(X) as a-goal(tower([X| T])).

module stackBuilder {
   if constructiveMove(X,Y) then move(X, Y).
   if misplaced(X) then clearBlocks.
}
```

The `move` action in the second rule of our previous `stackBuilder` module has been replaced with the `clearBlocks` module. This illustrates that a module can be used as any other action in a rule. In fact, it is not even possible to tell from the rule whether an action is selected or a module because the syntax is the same. You need to inspect the use clauses of the module to see that `clearBlocks` is a module. A plain use clause **use** `clearBlocks` is needed to indicate that the `clearBlocks` module is used in the module.

### 7.2.1 Entering a Module

A module is entered, executed, and exited. If a module is entered, it becomes the *active* module. There is at most one module that is active at any time. When a module is activated the rules in the module are the only rules used to generate action options. The module is executed by evaluating and applying the rules of the module using the rule evaluation order of the module. If a module does not set its rule evaluation order, *the default* ***linear*** *order is used.* Note that this also is true for modules called from a module used as event module.

A module that is called from a rule (instead of used in an agent definition to specify top-level modules), is entered if the rule is evaluated and applicable. A module does not have any pre-conditions and will always be entered when the condition of the rule holds. In our example if the second rule is evaluated and the condition `misplaced(X)` holds, the module `clearBlocks` will be entered. Of course, if a module is part of a combo-action, then the order of actions is respected, and the module is executed in the order that the actions appear in the combo-action. As modules can be called from within other modules hierarchies of modules can be used. Modules can be used recursively as well.

### 7.2.2 Controlling When to Exit a Module

You may have noticed that we changed the exit condition of the `clearBlocks` module. The default exit condition of a module is **always**. This means that a module is executed once and then control is handed back to the module that called it. This happens, for example, with a module that is used as event module. That module is (automatically) called from the main module (or

other active module called from that module) as part of the agent's cycle (see Figure 6.5). After executing the module, control is handed back to the main module. The only exception is the exit condition of a module that is used as main module. The main module's exit condition is set to **never** by default. The main module is not terminated after being executed once but remains in control over the action selection process as a top-level module that coordinates all decision-making of the agent. This option is used to guarantee that an agent keeps executing and does not terminate. Note that the agent cycle of Figure 6.5 is still executed, as the check whether to execute the event module is performed after each execution of the main or other active module in line with this cycle. Even if the exit condition of a module (including the main module) is set to **never** it is still possible to exit a module using the **exit-module** action (see below). For all modules other than the main module, it is not useful to set the exit condition to **never** without also using the **exit-module** action because otherwise that module would never hand control back to the module that called it.

The condition for exiting a module can be changed for any module using the **exit** option. Apart from the default exit option for modules **always** and the **never** option, there are two other options. One is the **nogoals** option that we have seen before. A module with a **nogoals** option is terminated when the agent has no goals to pursue any more in the module; this normally means that the goal base of the agent is empty. The other option is the **noaction** option used in our `clearBlocks` module above. The termination behaviour of a module that uses this option is changed to only end the module when *none of the rules of the module generate any options*, i.e. when the module does not select any actions to perform any more.

Finally, the internal action **exit-module** can be used to terminate a module at any time (no matter what exit condition has been specified for the module). This action can be used in a rule as any other action. It is important to realize, however, that when this action is executed, the module is terminated and any actions that would follow the **exit-module** action in the combo-action of a rule will not be executed.

## 7.3 Creating Focus

We have looked so far at agents that have a single goal they want to achieve. But agents can have multiple goals at the same time. One problem that agents they may need to handle is that these goals can conflict. An agent has conflicting goals if they cannot be achieved simultaneously. That an agent has conflicting goals, however, does not mean that they cannot be achieved one after the other. An agent that wants to go to a location `here` and a location `there` cannot be at both locations at the same time but can visit them sequentially. An agent just needs a method for handling conflicting goals. We use the Blocks World to illustrate conflicting goals and introduce the **focus** option of a module as a method for handling the conflict.

Suppose we have a variant of a Blocks World problem where our agent wants to build two different towers using the same six blocks. We use the original `stackBuilder` module again with two rules where the first makes a constructive move, if possible, and the second moves a misplaced block to the table (see Section 7.1). The agent has the following two goals (you need to change the `bwgoals.pl` file accordingly):

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

Also suppose that the agent starts in the initial Blocks World configuration of Figure 7.3 (in the `BlocksWorld` MAS file use the `start=[0,1,2,5,6,0]` option). If you run the MAS, you will find that the agent will never achieve either of its goals. The only moves it can make initially is to move either block `4` or `3` to the table (because both are misplaced given one of the goals). Thereafter the agent can make a constructive move again which moves the block moved to the table back in its original position (because the block can be used to constructively make progress with one of the two goals). The agent will repeat this forever.

| | | |
|---|---|---|
| *module* | := | *useclause*$^+$ *option*$^*$ *macro*$^*$ **module** *id* (*parameters*) **{** *rule*$^+$ **}** |
| *useclause* | := | **use** *id* [**as** *usecase*] **.** |
| *usecase* | := | **knowledge** \| **beliefs** \| **goals** \| **actionspec** \| **module** |
| *option* | := | **exit**= *exitoption* **.** \| **focus**= *focusoption* **.** \| **order**= *orderoption* **.** |
| *exitoption* | := | **always** \| **never** \| **nogoals** \| **noaction** |
| *focusoption* | := | **none** \| **new** \| **select** \| **filter** |
| *orderoption* | := | **linear** \| **linearall** \| **linearrandom** \| **random** \| **randomall** |
| *macro* | := | **define** *id*[ (*parameters*) ] **as** *msc* **.** |
| *rule* | := | **if** *csq* **then** (*actioncombo* \| **{** *rule*$^+$ **}**) **.** \| |
| | | **forall** *csq* **do** (*actioncombo* \| **{** *rule*$^+$ **}**) **.** \| |
| | | **listall** *var* **<-** *csq* **do** (*actioncombo* \| **{** *rule*$^+$ **}**) **.** |
| *csq* | := | *stateliteral* (**,** stateliteral)$^*$ |
| *stateliteral* | := | *statecond* \| **not** (statecond) \| **true** \| *id* (*parameters*) |
| *statecond* | := | [*selector***.**]*stateop* (*qry*) |
| *stateop* | := | **bel** \| **goal** \| **a-goal** \| **goal-a** \| **percept** \| *sent* |
| *sent* | := | **sent** \| **sent:** \| **sent**? \| **sent**! |
| *actioncombo* | := | *action* (+ action)$^*$ |
| *action* | := | *id* (*parameters*) \| *selectoraction* \| *generalaction* |
| *selectoraction* | := | [*selector***.**](**insert** (*upd*) \| **delete** (*upd*) \| **adopt** (*qry*) \| **drop** (*qry*) \| *send*) |
| *send* | := | **send** (*qry*) \| **send:** (*qry*) \| **send**? (*qry*) \| **send**! (*qry*) |
| *generalaction* | := | **exit-module** \| **log** (*parameters*) \| **print** (*term*) |
| *selector* | := | (*parameters*) \| **all** \| **allother** \| **some** \| **someother** \| **self** \| **this** |
| *qry* | := | *a valid KR query* |
| *upd* | := | *a valid KR update* |
| *parameters* | := | *term* (**,** *term*)$^*$ |
| *term* | := | *a valid KR term* |

Table 7.1: Module Grammar

**Focus of Attention** It is clear that the agent has conflicting goals that cannot be achieved simultaneously. To achieve its goals, the agent has to *select a goal to focus on* in order to achieve at least one of its goals. A module can be used to create a **focus of attention** on a specific goal of an agent using the **focus** option. This option allows an agent to select a goal from its goal base and to put it into a new so-called **attention set** that is associated with the module. An attention set is a new goal base that represents the current focus of an agent.

```
use bwknowledge as knowledge.
use bwgoals as goals.
use buildTower.

module stackBuilder {
   if a-goal(tower([X|T])) then buildTower.
}
```

In order to use this focus mechanism, we modified the `stackBuilder` module and replaced the rules for choosing a `move` action with a single rule that calls a new module called `buildTower`. As before, a use clause indicates that the module uses the `buildTower` module. As the choice of `move` action is delegated to this new module, there is no need any more for a use clause for an action specification of the `move` action. The agent that uses this version of the `stackBuilder` module as main module is able to handle multiple, conflicting goals in the Blocks World that our earlier Blocks World agent is not able to deal with. This is so because the `buildTower` creates a focus on a single goal of the agent.

The code for the module `buildTower` is provided in Figure 7.4. The idea is that this module

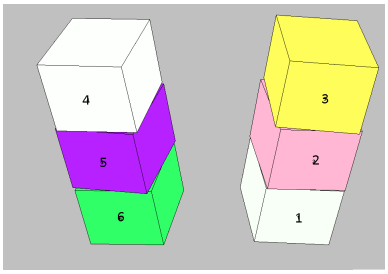


Figure 7.3: Initial Blocks World state

builds a single tower (or desired Blocks World configuration). The macros defined and the rules of the module and its exit condition are the same as those we used previously for the `stackBuilder` module. The use clauses for knowledge and the action specification are also copied. The main difference is the use of the **`focus=select`** option in the `buildTower` module. This option ensures that a focus is created on a single goal of the agent. As the agent has two goals to create two different towers (see above), this means that the module will focus on building only one of those towers at the same time.

```
use bwknowledge as knowledge.
use bwmove as actionspec.

exit = nogoals.
focus = select.

define constructiveMove(X,Y) as a-goal(tower([X,Y|T])), bel(tower([Y|T])).
define misplaced(X) as a-goal(tower([Y|T])).

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

Figure 7.4: Module `buildTower` using a focus option

Adding a focus option other to **`none`** to a module changes the way a module is executed. Each of the options **`new`**, **`select`**, and **`filter`** creates a new attention set, i.e. a new goal base that is associated with the module. This attention set (or goal base) is used for evaluating cognitive state queries instead of the attention set that is associated with the caller module. The main goal base of the agent is associated with the module used as main module. The difference between the options is how the content of the attention set is set. The **`new`** option simply creates an *empty* attention set. It is up to the module to adopt goals to create some content for the attention set. The **`select`** and **`filter`** options add a single goal from the current attention set to the new attention set each by means of a slightly different mechanism.

In our example, the `buildTower` module uses the **`select`** option. This option selects one of the goals in the current attention set that satisfy the condition of the rule that calls the module. The condition of the rule in our example is **`a-goal`**`(tower([X|T]))`. The agent has two goals in its current attention set associated with the main module `stackBuilder`:

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

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

The **goal**-condition, i.e. **goal**(tower([X|T])), of the rule's cognitive state query is evaluated using both goals as usual in combination with the agent's knowledge. A random choice is made to select one of the goals for which the condition holds. In our example, both goals satisfy the cognitive state query **goal**(tower([X|T])) and either one could be selected. The goal that is selected is added to a new attention set associated with the buildTower module, for example, the second goal as suggested above. This attention set becomes the new active attention set and will be used for evaluating the cognitive state queries of rules in the module. The idea is that the goal(s) in the active attention set are the goals that the agent currently actively pursues. Because the agent in our example will only have one goal it focuses on, the rules we used for building a Blocks World configuration are as effective again as they were before.

**Focus with the filter option** The focus option **filter** does not select one of the current goals of the agent but instead adds an instantiation of the rule's cognitive state query to the newly created attention set. The goal that is added is obtained as follows: first all **goal**-queries that occur positively in the rule's cognitive state query are collected. All instantiations for which the rule's condition holds are applied to these queries. This yields in our case a list of instantiated queries **goal**(tower([b1])), **goal**(tower([b2,b1])), etcetera. One of these instantiated queries is selected and the queries inside the **goal**-operators are collected and each of these queries is added separately to the new attention set. In our example, the following attention set could be associated with the buildTower module:

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

buildTower:
tower([b6,b3,b2,b1]).
```

**Lifecycle of an Attention Set** A module such as the buildTower creates a new attention set when it is entered. When this module is exited, the attention set is removed and the previously active attention set becomes the active attention set again. Any goals that have been added to the attention set by the module are thus automatically removed.

**The this and self selectors** Goals that are adopted by **adopt** actions are added to the currently active attention set. A selector can be used to change this. The selector **this** is the default selector and **this.adopt** and **adopt** have the same effect. Prefixing the selector **self** to the **adopt** action has a different effect. The action **self.adopt**(*qry*) adds the goal qry to the top-level goal base associated with the main module of the agent. A similar remark applies to **goal**-queries. A **goal**-query is evaluated on the active attention set, but a query of the form **self.goal**(*qry*) is evaluated on the top-level attention set. Note that there is no difference between the queries **bel**(*qry*), **this.bel**(*qry*) and **self.bel**(*qry*) as there is only one (global) belief base. Adding the **this** or **self** selector s to a **drop**(*qry*) action also do not change its effects: a goal in any of the attention sets of the agent that implies qry is removed from that attention set.

## 7.4 Notes

The original proposal to introduce modules into GOAL can be found in [22].

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 [7]. The *Tropism System Cognitive Architecture* designed for robot control also uses a random generator in its action selection mechanism [5]. As in GOAL, this architecture based on so-called tropisms ("likes" and "dislikes") may generate multiple actions for execution from which one has to be chosen. The selection of one

action from the chosen set is done by the use of a biased roulette wheel. Each potential action is allocated a space (slot) on the wheel, proportional to the associated tropism value. Consequently, a random selection is made on the roulette wheel, determining the robot's action.

# Chapter 8

# Communicating Agents

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

This chapter will explain how agents can communicate with each other. An example of a multi-agent system is introduced in Section 8.1 for illustrating the use of communication to coordinate the actions of multiple agents. Programming constructs for communication will be introduced and we will explain how agents can process messages in Section 8.2. Processing messages is similar to percept processing. The different types of messages that agents can send are introduced in Section 8.3. Section 8.4 discusses how agents can be selected that should receive a message. Finally, Section 8.5 discusses the example introduced in Section 8.1 in more detail.

## 8.1 Example: The Coffee Factory Multi-Agent System

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

```
define maker as agent {
  use makerInit as init.
  use coffeeEvents as event.
  use machine as main.
}

define grinder as agent {
  use grinderInit as init.
  use coffeeEvents as event.
  use machine as main.
}

launchpolicy {
  launch maker.
  launch grinder.
}
```

Figure 8.1: The Coffee Factory MAS (`coffee.mas2g`)

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. The MAS does not use an external environment. To focus on the communication, we just program agents that together make a coffee virtually. We can extend the two-agent MAS with additional agents, e.g., we could add an agent that represents a cow that can provide milk for making a latte. To make coffee, the coffee maker needs *water* and *coffee grounds*. It has water, and *coffee beans*, but not *ground* coffee. Grinding the beans is the task of the coffee grinder. The coffee grinder needs beans, and produces grounds. The coffee maker and the coffee grinder use the same main and event modules (and thus the same knowledge and action specifications as well); only how they are initialized differs.

The knowledge in Figure 8.2 reflects the agents' knowledge of which ingredients are necessary for which products and what it can make. 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 other agents can make. This is where communication comes in. The `maker` agent can make `coffee` and `espresso`. The `grinder` agent can make grounds.

```
:-dynamic have/1, % indicates that we have a product ready for use.
 canProduce/1, % indicates products that we can make ourselves.
 canProduce/2, % indicates that another machine can make a certain product.
 delivered/2. % indicates that a product has been delivered to a certain machine.

% Common knowledge of ingredients that are needed for making a product.
requiredFor(coffee, water).
requiredFor(coffee, grounds).
requiredFor(espresso, coffee).
requiredFor(grounds, beans).

% A machine can make a product if it has the product or it can make all ingredients.
canMake(Product) :- have(Product).
canMake(Product) :- canProduce(Product),
 forall(requiredFor(Product, Ingredient), canMake(Ingredient)).

% A short hand for having delivered a product to any machine.
delivered(Product) :- delivered(_,Product).
```

Figure 8.2: Knowledge in the Coffee Factory (`coffee.pl`)

Two simplifying assumptions have been made in the design of the Coffee Factory MAS. First, resources (like `water`, `beans`, `grounds` and `coffee`) cannot be depleted. Second, 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.

```
use coffee as knowledge.

module makerInit {
  if true
     then insert(have(water), have(beans), canProduce(coffee), canProduce(espresso)).
  if true then adopt(have(coffee)).
}
```

Figure 8.3: Module for initializing the Coffee Maker (`makerInit.mod2g`)

The modules of Figures 8.3 and 8.4 are used as init module and provide the agents with their initial goals. For example, the `maker` agent wants to have `coffee`. Note that this describes a goal *state* (`coffee` being available), not an action (e.g., 'making `coffee`').

```
use coffee as knowledge.

module grinderInit {
  if true then insert(canProduce(grounds)).
  if true then adopt(delivered(grounds)).
}
```

Figure 8.4: Module for initializing the Coffee Grinder (`grinderInit.mod2g`)

There is one action of making a product available to the agents. The `make` action is specified as an internal action (we do not use an environment).

```
use coffee as knowledge.

% If the Product can be made by us, we will have done so after this action
define make(Product) as internal with
  pre{ canMake(Product) }
  post{ have(Product) }
```

Figure 8.5: Action available in the Coffee Factory MAS (`coffee.act2g`)

## 8.2 Communication: Send Action and Mailbox

Agents use a mailbox or message base in which they receive messages sent by other agents. The mailbox can be inspected by using the cognitive state operator `(Agent).sent(Msg)`. Here `Agent` is the agent that sent the message and `Msg` is the content of the message (expressed in the knowledge representation language used). Figure 8.6 shows how this operator is used in the `coffeeEvents` module for processing messages that the agents exchange. Message processing is best done in a module used as event module.

The action `(AgentName).send(Poslitconj)` is an internal action provided by the programming language for sending `Poslitconj` to the agent with name `AgentName`. `Poslitconj` is a conjunction of positive literals, i.e. facts. `AgentName` should be the the name of the agent as specified in the MAS file. The message is sent to the target agent, and after arrival the message is placed in the target agent's mailbox. 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.

### 8.2.1 The **send** action

To illustrate the **send** action, let's first consider a simple example multi-agent system consisting of two agents, `fridge` and `groceryplanner`. Agent `fridge` is aware of its 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:

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

At the beginning of its next action cycle, the `groceryplanner` agent will receive this message. The received messages can be inspected by means of the **sent** operator. The `groceryplanner` can thus act on the received message:

```
use coffee as knowledge.

module coffeeEvents {
  % answer information requests.
  forall (Machine).sent?(canProduce(_)), bel(canProduce(Product) ; have(Product))
   do (Machine).send:(canProduce(Product)).

  % process information from other agents.
  forall (Machine).sent:(canProduce(Product)) do insert(canProduce(Machine,Product)).

  % process delivery requests.
  forall (Machine).sent!(have(Product)) do adopt(delivered(Machine,Product)).

  % process actual deliveries.
  forall (Machine).sent(have(Product)) do insert(have(Product)).

  % we have a request for a product but we don't have it yet.
  if goal(delivered(Machine, Product)) then adopt(have(Product)).

  % we can produce it but need an ingredient.
  if goal(have(Product)), bel(canProduce(Product), requiredFor(Product,Ingredient))
   then adopt(have(Ingredient)).
}
```

Figure 8.6: Event Processing in the Coffee Factory (`coffeeEvents.mod2g`)

```
if (fridge).sent(amountLeft(milk, 0))) then adopt(buy(milk)).
```

Note that the fridge will keep sending this message to the groceryplanner over and over again until it believes it will have some milk again. Some record-keeping would be required in either agent to prevent this.

### 8.2.2 Variables

Variables can be used to instantiate messages in rules. For example, a more generic version of the `fridge`'s program rule would be

```
if bel(amountLeft(P, N), N < 2)
   then (groceryplanner).send(amountLeft(P, N)).
```

Note that we used an **if-then**-rule here that will only send one message for one of the values of `N` for which we have `N<2`. By using a **forall-do**-rule messages for all products that are needed would be sent to the `groceryplanner`.

We can also use variables for the recipients and senders of messages in the cognitive state queries of a rule. For example:

```
% This isn't an argument; it's just contradiction!
% - No it isn't.
forall (X).sent(yes) do (X).send(no).

% http://en.wikipedia.org/wiki/Marco_Polo_(game)
forall (X).sent(marco) do (X).send(polo).
```

This is especially useful if you don't know upfront who will have sent a message.

```
use coffee as knowledge.
use coffee as actionspec.
exit=nogoals.

module machine {
  % if another machine needs a product and we have it, then deliver that product.
  if goal(delivered(Machine, Product)), bel(have(Product))
   then (Machine).send(have(Product)) + insert(delivered(Machine, Product)).

  % if we want to have a product and we can make it ourselves, then do so.
  if goal(have(Product)), bel(canMake(Product)) then make(Product).

  % if we can't produce something ourselves we need to order it.
  % if we know who can, tell that machine to make it for us.
  if goal have(Product)), bel(canProduce(Machine,Product))
   then (Machine).send!(have(Product)).
  % if we don't know that, ask everyone else what they can make for us.
  if goal(have(Product)), not(bel(canProduce(_,Product)))
   then allother.send?(canProduce(_)).
}
```

Figure 8.7: Module for decision making in the Coffee Factory MAS (`machine.mod2g`)

**Closed Actions** Like any other action that is selected for execution, a **send** action also must be closed, i.e. all variables in a **send** action must be bound after evaluation of the cognitive state query of a rule. This means that messages sent must be closed.

## 8.3 Moods of Messages

Up until now all examples have shown communication of an agent's *beliefs*. Every message was a statement about the sender's beliefs regarding the content. It would also be useful to be able to communicate about an agent's *goals*. In natural language, the mood of sentence is used to make clear that a message is about a goal instead of a belief. The mood of a sentence in a natural language can be *indicative* ('The time is 2 o'clock'), *expressive* ('Hurray!!'), *declarative* ('I hereby declare the meeting adjourned'), or *interrogative* ('When does the train leave?'). The GOAL programming language also supports moods by adding a **mood** operator at the end of the **send** operator. Three moods listed in Figure 8.8 are available.

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

Figure 8.8: Operators for Moods of Messages

Although there is an operator for the indicative mood, this mood operator is optional as it is the default operator. In other words, in the absence of a mood operator, the indicative mood is assumed. That means that all examples in Section 8.2 were implicitly using the indicative mood. The declarative mood can be used, for exampe, if the coffee maker or coffee grinder needs a resource to make something but does not have it. The agent then can send a message by using the declarative mood operator **send**! to all other agents using the **allother** operator to indicate that it needs the resource:

```
  % we can't produce it and need to order it.
  if goal(have(Product)), not(bel(canProduce(Product)))
```

```
 then allother.send!(have(Product)).
```

The mood operators can also be used in combination with the query operator **sent** to inspect the mailbox of an agent and check whether messages with a particular mood have been received. For example, to handle a message like the one above from the coffee maker, the coffee grinder can use the rule:

```
% if some agent needs something we can make, adopt the goal to make it
forall (Machine).sent!(have(Product)) do adopt(delivered(Machine,Product)).
```

This rule will make an agent adopt a goal to produce a product that another agent needs. Another rule in the `machine` module will then make sure that the agent that needs the product is notified when it is available.

We previously discussed that **send** ations (and thus messages) must be closed as any other action that is executed must be. But there is one exception: Interrogative messages do not need to be closed. These messages are similar to open questions, for example, "What time is it?" or "What is Ben's age?". Such questions cannot be represented by a closed sentence. Instead, a *don't care* can be used to indicate the unknown component. For example:

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

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

## 8.4 Agent Selectors

In many multi-agent systems agents may find themselves communicating with agents whose name they do not know beforehand. For example, the MAS may have been created by launching 100 agents using the **number** constraint (see Chapter 6, Figure 6.1). If some of these agents need to communicate with each other, the agent that needs to be addressed needs to be selected somehow. In a multi-agent system it may also be useful to multicast or broadcast a message to multiple receivers. For these cases a flexible way of addressing the receivers of messages is needed.

### 8.4.1 send action syntax

The **send** action allows more dynamic addressing schemes than just the agent name by using an **agent selector**. The available options for using agent selectors are listed in the module grammar of Table 7.1 and are illustrated next with some examples:

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

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

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

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

% using quantor
% if we don't know anyone who can make our required resource, broadcast our need
if goal(have(Product)), not(bel(canProduce(Product)))
 then allother.send!(have(Product)).
```

One observation with respect to the broadcasting of a message using the **allother** operator is important to avoid flooding the mailboxes of other agents. Note that we have used an **if-then**-rule for rule in the last example above for sending a message to *all other* agents. If we would have used a **forall-do**-rule here, for each instantiation of the variables a message would have been sent. Because there may be multiple ingredients required for a product `Product` (e.g., for `coffee`, see Figure 8.2), the agent would send the same message for each required ingredient it is missing, which is inefficient.

**Agent Name** The agent name is the simplest type of agent expression, which we have already seen in Sections 8.2 and 8.3. It consists of the name of the receiving agent. If the KR language of the agent is Prolog, the agent name must start with a lowercase letter. Example:

```
(alice).send:(hello).

% addressing multiple agents with a single send action
(alice,bob,charlie).send:(hello).
```

Note that when a message is sent to an agent, the agent name used must refer to an agent that is part of the MAS. If the agent name does not refer to an agent that is part of the MAS, or has died, or is otherwise unadressable, the agent that tries to send the message will terminate. An agent that successfully sends a message does not receive any feedback that confirms or disconfirms that the message has been received.

**Variables** A variable type agent expression allows a dynamic way of specifying the message recipient. Sometimes the recipient depends on the agent's beliefs or goals or on previous conversations. The variable agent expression consists of a variable in the agent's KR language. If the KR language is Prolog, this means it must start with an uppercase letter. This variable will be resolved when the program rule's cognitive state query is evaluated. This means that a cognitive state query **must** bind all variables that are used in the agent selector. If an agent selector contains unbound variables at the time that the agent decides to perform the action, the action will result in a failure and the agent will be terminated. To illustrate in an example, we assume an agent that believes there are two agents `agent(john)` and `agent(mary)` and has a goal `informed(john, fact(f))`:

```
if bel(agent(X)), goal(hold(gold)) then (X).send!(hold(gold)).

if goal(informed(Agent, fact(F))) then (Agent).send:(fact(F)).

% if applied, next rule terminates the agent as variable Agent is not instantiated
if bel(something) then (Agent).send:(something).
```

In this example, the first program rule 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 will select one of these options for execution.

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

- **all** will expand to all names of agents currently present in the MAS (including the name of the sending agent itself).
- **allother** will expand to all names of agents currently present in the MAS, with the exception of the sending agent's name.

- **some** will expand to the name of a randomly selected agent currently present in the MAS (including the name of the sending agent itself).
- **someother** will expand to the name of a randomly selected agent currently present in the MAS, with the exception of the sending agent's name.
- **self** will resolve to the sending agent's name.

## 8.5 The Coffee Factory MAS Again

In Section 8.1 the Coffee Factory MAS was introduced. In this section the workings of the coffee maker and coffee grinder are analyzed in more detail. We will see how the agents coordinate their actions by communicating in different ways.

**Capability exploration** The agents know what they can make themselves by using the basic beliefs that were inserted in their init modules (see Figures 8.3 and 8.4). To find out what the other agents can make, the following rules are used in the program (both in the main and the event module):

```
% if we don't know that, ask everyone else what they can make for us.
if goal(have(Product)), not(bel(canProduce(_,Product)))
 then allother.send?(canProduce(_)).

% answer information requests.
forall (Machine).sent?(canProduce(_)), bel(canProduce(Product) ; have(Product))
 do (Machine).send:(canProduce(Product)).

% process information from other agents.
forall (Machine).sent:(canProduce(Product)) do insert(canProduce(Machine,Product)).
```

The first rule checks if the agent already knows a machine that can make a certain product it needs. If this is not the case, the rule is applied and an *interrogative* message to ask which products *all other* agents can make is sent. Note that using **all** instead of **allother** would result in this agent asking itself what it can make.

The second rule handles such incoming interrogatives. It looks in the mailbox for received interrogative messages asking what this agent can make. It replies to the sender with one or more *indicative* messages, indicating what it can make (and what it already has). Note that as messages are deleted each cycle (just like percepts), each such interrogative message will be replied to only once. However, also note that as agents run asynchronously, this may result in one agent asking the other agents for a response multiple times, as the other agents might not have sent their reply yet, thus also resulting in multiple responses. Additional bookkeeping (i.e. of what was already sent or received) would be needed to prevent this.

Finally the indicatives are handled in the third rule. The mailbox is queried for received *indicative* messages, containing the information about what the other agents make. If such a message exists, this information is inserted as a fact in the belief base. Note that the information about the sender (i.e. the machine) and the product it can make are combined in the belief base.

**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 rules above, it can request the grinder to make grounds by sending it an *imperative* message. This is represented more generically in the following action rule (from the main module):

```
% if we know who can, tell that machine to make it for us.
if goal have(Product)), bel(canProduce(Machine,Product))
 then (Machine).send!(have(Product)).
```

When this agent has a goal for a product which it doesn't have, and it knows of a maker of this product, it sends an imperative message to that maker.

When such an imperative message is received by an agent, it can adopt a goal to make the requested product (in the event module):

```
% process delivery requests.
forall (Machine).sent!(have(Product)) do adopt(delivered(Machine,Product)).
```

Note that we assume that we are only asked to make something that we actually can make here.

**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 (in the main module). Agents in the Coffee Domain do not 'give' each other products or perceive that products are available (remember that we do not use an environment in this example), so they rely on communication to inform each other about that.

```
% if another machine needs a product and we have it, then deliver that product.
if goal(delivered(Machine, Product)), bel(have(Product))
 then (Machine).send(have(Product)) + insert(delivered(Machine, Product)).

% process actual deliveries.
forall (Machine).sent(have(Product)) do insert(have(Product)).
```

On the receiving side of this message, reception of an indicative message `have(Product)` does not automatically result in the belief by this agent that `have(Product)` is true. This insertion of the belief must be done explicitly (in the event module).[1]

**Pro-active inform** In the Coffee World, we could be more pro-active by always informing all other agents about what we can produce (or already have in stock). On one hand, this could reduce the number of questions the agents have to ask each other. On the other hand, this could lead to an increase in the number of unnecessary messages that are sent. For example, when we would add a milk cow in this MAS (see the Exercises at the end of this Chapter), it does not need to know about what the other agents can produce at all.

## 8.6 Notes

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 [12], 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 [17]. Providing a formal semantics of communication has received some attention in agent-oriented programming research. Some agent programming languages use middleware infrastructures such as JADE [6], 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

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

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.

## 8.7 Exercises

### 8.7.1 Milk cow

The coffee domain example from Section 8.1 has a coffee maker and a coffee grinder. Suppose we now also want to make lattes. A latte is coffee with milk. To provide the milk, a cow joins the scene. The cow is empathic enough that it makes milk whenever it believes that someone needs it.

1. Expand the list of products the coffee maker can make with `latte`.
2. Add the knowledge that `latte` requires `coffee` and `milk` to that of the coffee maker.
3. Write a new agent that uses a module `milkcow.mod2g` which has the following properties:
   (a) It does not do capability exploration, but it does answer other agent's questions about what it `canProduce`.
   (b) When it notices that another agent needs milk, it will make the milk resulting in the cow's belief that `have(milk)`.
   (c) When it notices that another agent needs milk and the cow has milk, it will notify that agent of that fact.

# Chapter 9

# The Design of Agent Programs

This chapter aims to provide some *guidelines* for developing agents 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 more elegant agent programs.

We also discuss how to structure and reuse parts of an agent program by means of *importing* modules 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.

## 9.1 Design Steps: Overview

As writing event and decision rules and action specifications requires that the predicates used to describe an environment are known, 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 at the end of 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 9.2.2 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 and developing a 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 event rules
   (b) Write action specifications

(c) Determine action selection strategy

(d) Write decision rules

Of course, it should be kept in mind that the steps that are part of this plan provide a rough guideline only 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 environment, 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 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. A basic guideline here is that **you should never introduce special predicates that are used *only* for representing goals**. An agent can never come to *believe* that it has achieved a *goal* if predicates are used only for representing goals. This indicates bad programming practice as an agent should always, at least in principle, be able to believe a goal has been achieved.

## 9.2 Guidelines for Designing an Ontology

A key step in the development of an 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. The main task of a programmer is to make 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 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. In this guide, we have used Prolog. *We assume the reader is familiar with Prolog and we refer for more information about Prolog to [8] or [43].* GOAL uses SWI Prolog; for a reference manual of SWI Prolog see [45].

### 9.2.1 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.

A first guideline is that it is best to **avoid the use of the don't care symbol "_" in cognitive state queries**. 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]

Second, it is important to understand that only a subset of all Prolog built-in predicates can be used for writing an agent program. Check the documentation that is made available for GOAL for the list of operators that can be used.

Finally, we comment on a subtle difference between Prolog itself and Prolog used within the context of an 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 in GOAL. The reason is that GOAL assumes that 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).

### 9.2.2 Knowledge, Beliefs, and Goals

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

- **knowledge**: represent the *environment or domain logic*
- **beliefs**: represent the *current and actual* state of the environment
- **goals**: 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 **as knowledge**. Examples are definitions of the concepts of `tower` in the Blocks World or `wumpusNotAt` in the Wumpus World.

Use the **beliefs** of an agent 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`. Logical rules are should be used **as knowledge**.

It is often tempting to define the logic of some of the goals the agent should pursue **as knowledge** instead of **as goals**. This temptation should be resisted, however. Predicates like `priority` or `needItem` with strong motivational connotations should not be used in **as knowledge**. 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

[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**($\varphi$) is defined by **goal**($\varphi$), **not**(**bel**($\varphi$)). We can illustrate what goes wrong by instantiating $\varphi$ 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!

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.

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** Automatic support is available that provides information about the use or lack of use of predicates in an agent program. Check out the user manual to find out more about how and where this information is provided [28]. 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.

## 9.3 Action Specifications

Actions that the agent can perform in an environment must be specified in an action specification file (with extension `act2g`). Actions that have not been specified cannot be used by an agent.

### 9.3.1 Action Specifications Should Match with the Environment Action

An action specification consists of a *pre-condition* of the action and a *post-condition*. The pre-condition 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 pre-condition should be set to `true` *only if* the action can *always* be performed. It also means that a pre-condition should *not* include conditions that are more restrictive than those imposed by the environment. For example, the pre-condition 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 decision rule(s) that select the action.

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

```
define forward with
  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:

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

### 9.3.2 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 must add **`as internal`** as use case. That is, directly after the declaration of the name of the action (and its parameters) you should write **`as internal`**. Actions that are not internal actions of the GOAL language itself are sent to the environment for execution if the action specification does not indicate that the action should be treated as an

internal action. If an environment does not recognize an action, it may throw an exception and terminate the agent.

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 **insert**and **delete**actions.

## 9.4 Readability of Your Code

GOAL is an agent oriented programming language 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 cognitive state queries and to use these macros instead of the cognitive state queries 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.

### 9.4.1 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.** It is also possible to use block comments using the start `/*` and end `*/` separators.

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 a KR file 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 post-conditions of the action (as specified in a manual for an environment, for example; doing so allows others to check your specifications),
- at the start of a module file 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*.

### 9.4.2 Introduce Intuitive Labels: Macros

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

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

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

*Macros should be placed in the modules that use them, before the rules.*

## 9.5 Structuring Your Code

GOAL provides various options to structure your code. A key feature for structuring code are *modules* (see Chapter 7). Another approach to structuring code is to *group similar rules together*. It is often useful to group rules that belong together in a module. An example design guideline, for example, is to put all rules which select environment actions in a module that is used as main module.

### 9.5.1 All Modules Except for the Main Module Should Terminate

The only module that does not need to terminate is the **`main module`**. A module used as main module will by default never be exited. This allows an agent to run, in principle, forever. 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.

### 9.5.2 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.

#### Decision Rules

Although we use the label *decision 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, *decision rules should be placed inside a module that is used as main module or in modules called from that module.*

#### Percept Rules

A rule is a *percept rule* if its cognitive state query inspects the percept base of the agent and *only* updates the cognitive state of the agent. That is, if the rule has a query that uses the

**percept**operator and only has actions that modify the cognitive 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 a module that is used as 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 a module used **as init** module. Percept rules placed inside such a 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 rules that can be classified as communication rules. A rule is a *communication rule* if its cognitive state query inspects the mailbox of the agent. That is, if the rule has a condition that uses the **sent**operator, that rule is a communication rule. A rule that selects a communicative action such as **send** also is communication rule.

As a design guideline, *communication rules that only update the cognitive state of the agent should be placed directly after percept rules* in an event module. 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 an event module, or possibly in a module that is called there.

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

- **Reconsideration**: Drop goals that are no longer considered useful or feasible,
- **Motivation**: 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 to make sure that 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. A 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 an event module.

#### How NOT to Use Rules

Rules 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*. 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 a 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 not(percept(stench)),
       bel( 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) :- ...
```

### 9.5.3 Small Modules

Small modules are to be preferred over modules with a large number of rules. By using more small modules code becomes more readable. Modules are abstract actions and can be given intuitive names to indicate what will happen when a module is executed. Smaller modules also facilitate re-use. Finally, smaller modules are easier to maintain and test, a topic we will discuss in the next chapter.

## 9.6 Notes

The guidelines discussed in this chapter are based in part on research reported in [29, 37].

# Chapter 10

# Automated Testing of Agents

Manual testing, using, for example, a debugger to identify differences between observed and intended behaviour, is not the most efficient failure detection method. It also heavily relies on the programmer to identify the failure and does not support performing the same test repeatedly. The **automated testing framework** that we introduce in this chapter facilitates running tests repeatedly at no additional costs. We also provide **test templates** for writing tests for specific aspects of an agent program such as event processing and action selection.

## 10.1 Modules as Basic Unit for Testing

As is important for any other testing framework, it is important to identify what the unit that will be tested should be. A testing framework for agent programs, for example, should not focus on the knowledge that an agent uses. That would be reinventing the wheel as developers can already use existing (unit) testing frameworks for the underlying KR technology used by an agent program. For example, when using SWI Prolog, a developer should use the available unit testing framework PlUnit [46] to test Prolog programs. Testing at the level of individual goals or rules is too fine-grained and also not that useful. Writing tests for individual rules, for example, would not only result in more test than source code, but even worse, would not focus on the failures that need to be detected. A more suitable level is the aggregate level that collects multiple rules in a single unit. We therefore focus on *modules* as units for testing.

Test conditions will evaluate conditions on the cognitive state of an agent. Test conditions will be evaluated when a module is entered and when it is exited again. Entering a module when it starts executing and exiting a module when module execution is finished provide two execution points that are natural places for evaluating test conditions. As modules can be viewed as abstract actions, these execution points can be viewed as the **pre-condition** and **post-condition** of a module which are evaluated, respectively, when entering and when exiting the module. Different from basic actions, however, is that we also want to evaluate tests while a module is executed. To be able to evaluate a module's behaviour, we therefore also use so-called **in-condition**s that are evaluated while a module is executed. An in-condition is a property that is evaluated on the execution run generated by a module. An execution run consists of a sequence of cognitive states that the agent creates by performing updates on its state.

The pre-, post, and in-conditions allow the detection of failures that occur during module execution. These conditions also provide better support for fault localization as a test indicates the code location where a failure was detected (when debugging; also see the User Manual [28]).

## 10.2 Test Language

Tests are programs themselves that we write in a **test language**. The test language is built on top of the GOAL programming language and re-uses parts of that language. The language

| | | |
|---|---|---|
| *test* | := | *useclause*$^+$ [*timeout*] *moduletest*$^*$ *agenttest*$^+$ |
| *useclause* | := | **use** *id* [*ascase*] **.** |
| *ascase* | := | **as** (**knowledge** \| **actionspec** \| **module** \| **mas**) |
| *timeout* | := | **timeout** = *integer* **.** |
| *moduletest* | := | **test** *id* **with** [**pre{** *statecond* **}**] [**in {** *testcond*$^+$ **}**] [**post{** *statecond* **}**] |
| *statecond* | := | *stateliteral* \| *stateliteral*, *statecond* |
| *stateliteral* | := | *stateatom* \| **not** (*stateatom*) |
| *stateatom* | := | *stateop* (*qry*) \| **done** (*action*) |
| *testcond* | := | (**always** \| **never** \| **eventually**) *statecond* **.** \| *statecond* **leadsto** *statecond* **.** |
| *agenttest* | := | *id* (**,** *id*)$^*$ **{** *testaction*$^+$ **}** |
| *testaction* | := | **do**(*action* \| *id*) [**until** *statecond*] **.** |
| *id* | := | *alphanumeral with underscores that starts with letter or underscore* |
| *stateop* | := | **bel**\| **goal** \| **a-goal** \| **goal-a** \| **percept**\| **sent** |
| *qry* | := | *a valid KR query* |
| *action* | := | *a valid action of the programming language or environment* |

Table 10.1: Test Language Grammar

provides support for two main tasks: *setting up a test* and *specifying* which *test conditions* should be evaluated. The grammar of the test language is specified in Table 10.1.

**Test Conditions** Test conditions are built on top of the cognitive state queries that are used in program rules. A condition **done**(*action*) can be used to test whether some action has just been performed. We call cognitive state queries and conditions of the form **done**(*action*) also state conditions.

A test condition is a temporal condition that expresses that something should happen always, never, eventually, or when some other condition has been true before. Test conditions are of the form:

- **always** *sc*, which means that the state condition *sc* should continuously (always) hold while executing a module.
- **never** *sc*, which means that the state condition *sc* should never hold while executing a module.
- **eventually** *sc*, which means that the state condition *sc* should hold at least once during the execution of a module.
- *sc1* **leadsto** *sc2*, which means that whenever the state condition *sc1* holds, some time thereafter the state condition *sc2* should hold.

The conditions **always** *sc* and **never** *sc* can be used to specify *safety conditions*, i.e., things that always or never should occur. The conditions **eventually** *sc* and *sc* **leadsto** *sc* can be used to specify *liveness conditions*, i.e., things that are supposed to occur sooner or later after something else has happened. **eventually** *sc* is a shorthand for **true leadsto** *sc*.

### Test Setup

A test needs to specify everything that is needed for the test. The first thing that is needed is a MAS file. A MAS file is used to launch an environment, and to launch and connect agents to this environment. As an example, we will test for a failure to handle 'incomplete' goals of our first Blocks World agent. We need to include the MAS file for the Blocks World:

```
use BlocksWorld as mas.
```

With out test we will show that something we want to happen eventually actually never happens. Our agent should move a block in order to achieve its goal but does not do it. We thus want our test to fail. But to show that something will never happen takes a long time. We will instead be satisfied if our agent does not move the block within a window of 1 second. This is a reasonable time window because the Blocks World agent is very fast and we will use a problem with only 8 blocks. We can use a **time out** to ensure termination of the test after a specified time. A time out is global and specifies how much time (in seconds) is allowed to pass before the entire test should be completed. If a time out happens, the test is aborted. It is useful to note that a test that is aborted does not always fail. A test that is aborted only fails if at least one test condition failed (see below). We add a time out as follows:

```
timeout = 1.
```

**Test Programs** A test is a program that specifies what should be done. First, a test should make clear which agents should take part in a test. Not all agents of a MAS have to be part of a test. The agents that take part in a test need to be referenced explicitly in a test program by naming them using their *id*s. These agents are launched when the test is started and automatically connected to an environment, if available, to receive percepts from and perform actions in that environment.

In a test we can also execute only part of an agent and even make the agent do things it would not otherwise do. The latter is useful for modifying the cognitive state of an agent and prepare it as desired for the test. Although we can execute the program code of an agent it does not need to be executed. Instead, the *testaction*s that are specified in an *agenttest* clause (see Table 10.1) are performed when the test is run. Test actions can be preparatory actions **do** *action* for, e.g., initializing an agent's state, where *action* can be a combo action that consists of one or more actions that are available to the agent. Test actions can also be instructions **do** *id* to execute a module with name *id*. We simply want the `stackBuilder` agent to execute as is, which we can achieve by "doing" the `stackBuilder` module that is used as main module (the same name is used to name the agent and the module used as main module):

```
stackBuilder {
   do stackBuilder.
}
```

It is important to note that modules that are used as init or event modules will also be executed like they usually would during agent cycles (see Figure 6.5).

An agent test can also be shared by multiple agents, by simply listing all agent names that should perform the test actions separated by commas. By specifying multiple agent tests it is also possible to define different actions for different agents, which will then be executed in parallel.

Finally, a condition **until** *sc* can be associated with a module (or an action but that is not very useful) that terminates execution when the state condition *sc* holds. An agent test thus determines which actions and modules are executed and when they should be terminated.

**Tests for Modules** The most important part of writing a test is specifying the conditions that should be evaluated while executing a module. The conditions that should be evaluated when a module is executed are specified by a **test** *id* **with** statement, where *id* is a module name. It is possible to associate a pre-condition **pre{***sc***}**, a post-condition **post{***sc***}**, and an in-condition **in {***tc*$^+$**}** with the module test. The pre-condition of a module is a state condition *sc* that should hold when a module is entered (otherwise, the test fails). Similarly, a post-condition is a

state condition `sc` that should hold when a module is exited. An in-condition `tc` is a temporal test condition that specifies which behaviour is expected of a module while it is executed.

We want to check whether our Blocks World agent will move a block at some point in time during the execution of its main module. More precisely, we want to know whether at some point in time the agent will perform the action `move(b8,X)` where `X` can be any other block or the table. We can use the **`eventually`** operator for this. As temporal conditions are specified as in-conditions, and we want to evaluate the `stackBuilder` module, we get the following module test:

```
test stackBuilder with
   in { eventually done(move(b8,X)). }
```

### Test Evaluation

By putting everything together, we get our first test. We need to add one use clause to indicate that the module `stackBuilder` is used:

```
use BlocksWorld as mas.
use stackBuilder as module.
use bwmove as actionspec.

timeout = 1.

test stackBuilder with
   in { eventually done(move(b8,X)). }

stackBuilder {
   do stackBuilder.
}
```

Figure 10.1: Test whether Blocks World agent moves block `8`

Because we do not want our agent to simply fail because there is no block `8`, we moreover add a block `8` that sits on top of block `1` to the MAS we developed in Figure 5.1:

```
use "blocksworld-1.1.0.jar" as environment with start=[2,3,0,5,0,7,0,1].

define stackBuilder as agent {
   use stackBuilder as main module.
}

launchpolicy {
   when * launch stackBuilder.
}
```

We do not change the goal of the agent, which we repeat here for completeness:

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

Note that this goal does not include block `8`.

A run or trace of an agent program consists of a (finite or potentially infinite) sequence of cognitive states of the agent. Test conditions associated with a module are evaluated on (partial) traces generated by that module. These conditions are assigned one of three values: *undetermined*, *passed*, or *failed*. Initially, all test conditions of a module have the value *undetermined*. The pre-condition of a module, if specified, is evaluated on the current state when entering the module and

assigned *passed* when the condition succeeds, and *failed* otherwise. Similarly, the post-condition is evaluated on the current state when a module is exited. The value of an in-condition is (re-)evaluated every time the cognitive state of the agent changes while the module is being executed. The temporal operator of the condition determines whether and how the value is updated:

- **always** *sc*: the value is changed to *failed* if *sc* does not hold in the cognitive state in which the condition is evaluated; the value is changed to *passed* if the test is terminated and the value still is *undetermined*; otherwise, its value remains *undetermined*.
- **never** *sc*: the value is changed to *failed* if *sc* holds in the state; the value is changed to *passed* if the module (or test) is terminated and the value still is *undetermined*; otherwise, its value remains *undetermined*.
- **eventually** *sc*: the value is changed to *passed* if *sc* holds in the cognitive state in which the condition is evaluated; the value is changed to *failed* if the test is terminated and the value still is *undetermined*; otherwise, its value remains *undetermined*.
- *sc1* **leadsto** *sc2*: if the module (or test) is terminated, the value is changed to *passed* if every state where *sc1* holds has been followed by a state where *sc2* holds (and vacuously so if *sc1* did never hold); otherwise, the value is changed to *failed*. If the module (or test) has not been terminated yet, the value is *undetermined*.

A test is aborted as soon as a condition is assigned the value *failed*. In that case, the entire test is regarded as failed, indicating that something needs to be fixed. Note that when a test is terminated (whether aborted or not), all conditions will have been assigned the value *passed* or *failed*. If a test is terminated because of a time out, this does not always imply that the test is a failure; if all conditions are passed, the test is considered to have passed as well.

Now it is time to run our test. We refer to the User Manual for instructions on how to do this [28]. You should get output that looks like:

```
[stackBuilder]  +++++++ Cycle 628 +++++++
[stackBuilder] 'stackBuilder' did not complete successfully during the
   test of agent 'stackBuilder' because: In-condition(s) failed: [
   'eventually done(move(b8, X))' with []].

test failed:
  test: ...\BlocksWorld2Agents\incompleteGoal.test2g
  mas:  ...\BlocksWorld2Agents\BlocksWorld.mas2g
  'stackBuilder' did not complete successfully during the test of
  agent 'stackBuilder'
  because: In-condition(s) failed: [
  'eventually done(move(b8, X))' with []].
```

We have identified a failure in our program. The agent never moves block `8` (at least not within the 1 second window that we used).

## 10.3 Test Templates

Test templates facilitate writing tests. Test templates also help increase the coverage of aspects that need testing. We introduce test templates for all aspects of an agent program. The test templates are split into three main categories: templates for percepts with labels that start with **P**, templates for goals with labels that start with **G**, and templates for actions with labels that start with **A**. We briefly introduce the templates here and discuss how to use them in the next section.

### 10.3.1 P-templates: Failures in Percept Processing

In order to support various options for percept processing, we distinguish between the four percept types and associate specific test templates with each type, based on the assumption that the percept information needs to be made persistent in the agent's belief state one-to-one. Test conditions for percepts should be associated with the module that processes the percept. This is usually a module used as either init or event module. This ensures the templates are evaluated while percept are processed in the module. Because the event module is executed once each cycle of the agent, in order to not violate the test conditions, percepts must have been processed and beliefs updated accordingly at the end of that module.

**Template *P-once*** : concerns percepts `p` that are only received *once*, typically when the agent is launched to inform about static information such as locations on maps. The test template expects that after receiving the percept, it will be made persistent such that the agent believes it. This templates should be associated with a module used as **`as init module`**.

```
percept(p) leadsto bel(p)
```

**Template *P-always*** : concerns percepts about facts `p` that are *always* received when `p` is true. This also implies that if such a percept is not received that `p` does not hold. The test template therefore consists of two test conditions. The first is the same as the condition of the ***P-once*** template. The second condition says that when `p` is not perceived, which indicates that `p` does not hold, a belief `p` should be removed (if present).

```
percept(p) leadsto bel(p)
not(percept(p)), bel(p) leadsto not(bel(p))
```

**Template *P-on-change*** : concerns percepts $\texttt{p}(\vec{t})$ that are sent only when the parameters $\vec{t}$ of a percept `p` change. A percept `loc(place)`, for example, might be sent only when an agent's location changes.

```
percept(p(t⃗)) leadsto bel(p(t⃗))
percept(p(s⃗)), bel(p(t⃗), not(s⃗ = t⃗)) leadsto not(bel(p(t⃗))
```

**Template *P-on-change-with-negation*** : concerns percepts `p` that are received once when `p` becomes true, and percepts **`not`**`(p)` that are received once when `p` becomes false (again). For example, `in(room)` is received when an agent enters a room, and **`not`**`(in(room))` is received when it leaves again.

```
percept(p) leadsto bel(p)
percept(not(p)), bel(p) leadsto not(bel(p))
```

### 10.3.2 G-templates: Failures in Goal Management

There are five failure templates that concern the management of goals. Each of these categories, with the exception of *G4*, suggests that a *reason* for (not) having a goal has *not* been adequately taken into account.

**Template *G-adopted* (G1)** : concerns a goal `p` that the agent should adopt because of some reason `sc`. If the agent does not adopt the goal when the reason holds, this template will identify the failure.

```
sc leadsto goal(p)
```

**Template *G-reconsideration* (G2)** : concerns a goal `p` that should be reconsidered and dropped for reason `sc`. If an agent does not drop the goal when `sc` holds, a failure to drop a goal that should be dropped is identified. The agent did not adequately reconsider the goals that it has.

```
sc leadsto not(goal(p))
```

An agent would normally reconsider its goals if the environment has changed outside the control of that agent. Failures of this type would therefore most likely only occur in dynamic environments or in a multi-agent context.

**Template *G-incorrect* (G3)** : concerns a situation in which there is a reason `sc` for *not* adopting or having a goal. This template can be viewed as the counterpart of the template *G-adopted*. Instead of a liveness (**`leadsto`**) condition we use a a safety (**`never`**) condition here.

```
never goal(p), sc
```

**Template *G-duplicate* (G4)** : concerns a single-instance goal that should be instantiated at most once. Some goals should only occur once and it should never be the case that the goal is instantiated twice (see also 6.8.1). For example, an agent might have a goal `in('RoomA1')` of visiting a room but should never have another goal of the same form, e.g., `in('RoomB1')`, at the same time.

```
never goal(p(s⃗)), goal(p(t⃗)), not(bel(s⃗ = t⃗))
```

**Template *G-maintain* (G5)** : concerns a situation in which `sc` is a reason why an agent should have a goal `p`, and should maintain it for that reason. This template can be viewed as the counterpart of template *G-reconsideration* that requires an agent to reconsider, i.e. to not maintain a goal.

```
never not(goal(p)), sc
```

### 10.3.3 A-templates: Failures in Action Selection

The final two templates concern failures in the action select strategy of an agent. An agent may have a reason to perform an action but not do so, or, vice versa, may have a reason to not perform an action but do so nevertheless.

**Template *A-selected* (A1)** : concerns an action `action` that the agent is should select because of reason `sc`. Failure to meet this test condition suggests that some reason for selecting an action has not been adequately taken into account.

```
sc leadsto done(action)
```

**Template *A-incorrect* (A2)** : concerns a situation `sc` in which an action `action` should never have been selected. A failure to meet the test condition suggests that something happened that should never have happened. This template can be viewed as the counterpart of previous template.

```
never(done(action)), sc
```

## 10.4 Test Approach

The test templates provide a useful starting point for writing tests. They facilitate a structured approach to testing an agent. Here we introduce a systematic test approach that consists of a number of concrete steps. The main steps of this approach are:

1. define success in terms of functional requirements,
2. test cognitive state updating, and
3. classify failures that concern actions and goals.

We also provide guidelines for instantiating the templates for a specific application. These guidelines suggest ways, for example, for finding specific reasons for instantiating the state conditions `sc` that need to be filled in in the *G*- and *A*-templates. For this purpose, it is important to be able to retrieve relevant information from the sources that we have available. Table 10.2 lists information resources that are particularly useful for writing tests.

| Source | Type of Information |
|---|---|
| Agent program (comments) | Clues for reasons & design |
| Agent trace (screen, logs) | Observable behaviour |
| Agent design & specification | Functional requirements |
| Environment (documentation) | Percepts, actions available |

Table 10.2: Information sources for testing

### Step 1: Defining Success

The first step is to identify **functional requirement**s from available agent design documentation (Table 10.2). These requirements define success and provide a concrete method for checking that a program does what it is supposed to do. A program can be considered free of failures if it meets requirements.

In order to automatically check this, functional requirements must also be specified in the test language. Typically, these requirements will be associated with a module `modname` that is used as main module. We can specify functional requirements as the pre-, post-, or in-conditions of this module using **`test`** `modname` **`with`** statements, or by adding a test action of the form **`do`** `modname` **`until`** *`sc`*.

Using a test action is particularly useful for checking that some overall objective `sc` is realized. If the objective is achieved, the test action will be automatically terminated. A **`timeout`** should be specified to guarantee termination in case `sc` would never occur. For example, the requirement or objective to pickup and deliver a sequence of packages `[p1,...,pn]` can be specified by **`do`** `modname` **`until bel`**`(delivered([p1,...,pn]))`.

### Step 2: Testing Cognitive State Updating

What an agent decides to do depends to a large extent on the content of its cognitive state. Test conditions also depend on the evaluation of state conditions on the cognitive state of an agent. If these state conditions incorrectly succeed or fail because the updating of the state of an agent has not been implemented correctly, tests will also very likely fail for unclear reasons. For example, a condition `never done(putDown),not(bel(in(Room)))`, which says that a package should never be put down when not in a room, could fail just because the beliefs about `in(Room)` are not updated correctly. It is therefore important to first make sure that the updating of an agent's state works as expected.

**Identify the Percepts, Actions, and Goals used in a MAS** As a preparatory step, it is useful to create an **ontology** of and collect all percepts, including their type (*once*, etc.), that may be received and the actions that may be performed from environment documentation (Table 10.2). Similarly, all goals that an agent may have should be collected from the agent program code and added to the ontology.

**Validating Percept Processing** The first step now is to instantiate the appropriate test templates ***P-once***, etc. for each percept based on their type. The resulting test conditions should be associated as in-conditions with the module(s) used for percept processing module. Tests should be repeated sufficiently often as percepts generated will differ per run, if only because environments are more often than not non-deterministic. To gain confidence that percepts are correctly handled, it is important to check against the list of actions created above whether a sufficient variation of actions has been performed during runs, as different actions often yield other percepts.

**Check Single-Instance Goals** Based on program design, and intended use of goals in comments in a program (Table 10.2), for example, and using the overview of goals in the ontology, the subset of goals that are single-instance goals should be identified. For each of these goals, the test template ***G-duplicate*** should be instantiated and associated as an in-condition with the module where the goal is adopted.

If these initial tests succeed, this will give a high level of confidence that cognitive states are updated correctly.

### Step 3: Classifying Failures

Instantiating the remaining template types requires some understanding of the program design and the agent's behaviour in order to be able to instantiate the required state conditions `sc`.

**Action Failures** For identifying action related failures, *A*-templates should be instantiated with actions and a state condition needs to be identified that provides a reason, i.e. a state condition `sc`, for (not) selecting it. For the template *A-selected* (respectively, *A-incorrect*), the question is in which situations `sc` an action should (never) be executed. The instantiated conditions should be associated as in-conditions with the module(s) where the action might (not) be selected. There are two basic approaches for identifying the conditions `sc`:

1. By inspecting the agent program, clues may be obtained for useful state conditions `sc`. In particular, the conditions of rules can be useful, as they typically indicate reasons for selecting an action. For example, a condition `bel(in('DropZone'), holding(Block))` that triggers execution of an action `putDown` suggests that an agent should execute `putDown` when it is holding a block in the `'DropZone'`. By simply using this condition for `sc`, we can instantiate template *A-selected* as follows:

```
bel(in('DropZone'), holding(Block)) leadsto done(putDown).
```

This approach is able to detect failures, e.g., in case the rule order prevents the rule for `putDown` from ever being applied. Similarly, by negating conditions found in a program, we can find useful conditions for instantiating *A-incorrect*. It is important to note that this works only if the condition used must hold if the action is selected. This is not always the case but when it can be assumed this approach provides a useful starting point. Moreover, you can consider how weakened or strengthened variants of conditions used in program rules can be used in test conditions.

2. If an action failure is suspected because, for example, a functional requirement is not satisfied, observing an agent's behaviour may provide clues for identifying a useful condition `sc` for instantiating a test template for that action. Suppose that a requirement **`bel`**`(in(Room))` **`leadsto not`**`(`**`bel`**`(in(Room)))` formulated in step 1 fails. That is, an agent does not always leave a room after entering it. If we now observe that the `goTo` action is never performed, we can conclude that we have identified a failure to select this action. To confirm this by a test, we can use the template *A-selected* and instantiate `sc` with the reason for leaving and `action` with the `goTo` action. This would give:

   **`bel`**`(in(Room),` **`not`**`(Room=OtherRoom))` **`leadsto done`**`(goTo(OtherRoom))`

   We can repeat this line of reasoning until a root cause for the failure has been identified.

**Goal Failures** The approach for instantiating *G*-templates, apart from identifying the goal that might cause the failure, is similar to that for *A*-templates. The questions that you should ask for each of the templates are:

- *G-adopted*: for which `sc` should a goal `p` be added?
- *G-reconsideration*: for which `sc` should a goal `p` be dropped?
- *G-incorrect*: for which `sc` should a goal `p` never be added?
- *G-maintain*: for which `sc` should a goal `p` never be removed?

The instantiated conditions should be associated as in-conditions with the module(s) that are related to the goal.

As an example, we create a test for a goal `in(Room)`. We assume that this goal is adopted by a rule with the following condition: **`bel`**`(room(Place)),` **`not`**`(`**`bel`**`(visited(Place)))`. This condition suggests that the agent should adopt (multiple) `in(Room)` goals for each room that it has not visited before. By using the goal and this condition for `sc` to instantiate the template *G-adopted*, we get:

**`bel`**`(room(Place)),` **`not`**`(`**`bel`**`(visited(Place)))` **`leadsto goal`**`(in(Place))`

This rather straightforward approach of re-using rule conditions can already provide an effective method for detecting failures, e.g., in case the rule order prevents the rule from ever being applied. Similarly, the negations of conditions found in a program can sometimes be used to instantiate the template *G-incorrect* to obtain useful test conditions. This approach for instantiating *G-incorrect* only works if the condition must hold whenever the goal is adopted, e.g., if an agent never wants to go to rooms it has visited before. A similar approach can be used for the templates *G-reconsideration* and *G-maintain*.

## 10.5 Debugging, Testing, and Fault Localisation

It is important to realize that how agents are executed can make a difference. For example, agents that are executed using the automated testing framework are never paused. Debugging agents with a debugger by pausing and/or stepping a program may result in agent behaviour

that is quite different from an agent that is executed without pausing it. Moreover, each run can produce different behaviour (and thus failures) because of non-determinism in the agent (e.g., due to random rule order evaluation), the environment, or executing multiple agents. You should therefore always run the same tests multiple times in different scenarios to gain assurance that the agents work as expected.

When a failure is detected, i.e., a test fails, the fault must be located. The program location where the agent is at when the test failed is indicated by the testing framework. Although it is often the case, it is not always true that this location also is the fault location, i.e., the place of the actual error in the code. If the fault is not located immediately additional debugging is needed using a debugger (see the User Manual [28]). In particular, faults related to actions that are performed but should not have been performed are usually more difficult to locate.

## 10.6 Notes

The automated testing framework was first developed and reported on in [32].

# Bibliography


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

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

[3] 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

[4] 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)

[5] 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)

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

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

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

[9] 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)

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

[11] Boutilier, C.: A unified model of qualitative belief change: a dynamical systems perspective. Artificial Intelligence **98**(1?2), 281 – 316 (1998). DOI http://dx.doi.org/10.1016/S0004-3702(97)00066-0. URL http://www.sciencedirect.com/science/article/pii/S0004370297000660

[12] 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)

[13] 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)

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

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

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

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

[18] 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

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

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

[21] Hanks, S., Pollack, M.E., Cohen, P.R.: Benchmarks, test beds, controlled experimentation, and the design of agent architectures. AI Magazine **14**(4), 17–42 (1993). URL http://dl.acm.org/citation.cfm?id=187082.187085

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

[23] 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)

[24] 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)

[25] 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

[26] 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)

[27] 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)

[28] Hindriks, K.V., Pasman, W., Koeman, V.J.: GOAL-Eclipse User Manual. http://ii.tudelft.nl/trac/goal/wiki/WikiStart#Documentation (2016)

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

[30] 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)

[31] 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

[32] Koeman, V.J., Hindriks, K.V., Jonker, C.M.: Automating failure detection in cognitive agent programs. In: Proceedings of the 2016 International Conference on Autonomous Agents and Multi-agent Systems, AAMAS '16. International Foundation for Autonomous Agents and Multiagent Systems (2016). Pending publication

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

[34] 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)

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

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

[37] 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)

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

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

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

[41] 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)

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

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

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

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

[46] Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: Swi-prolog. Theory and Practice of Logic Programming **12**, 67–96 (2012)

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