Programming Marbel Agents

December 14, 2022

Preface

MARBEL is a rule-based programming language for programming **logic-based 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**. Agents maintain a **database** that they use for reasoning about their environment and for making decisions about what to do next.

The language offers a rich and powerful set of programming constructs and features for writing **agent programs**. The platform developed for running agent programs also provides support for connecting multi-agent systems to environments such as simulators, games, and robots. In principle, there is no limit to the possibilities and a diverse set of environments has been made available that can be used for programming agents (see Chapter 1 for examples).

The MARBEL agent programming language is a successor of the GOALprogramming language. MARBEL simplifies a number of things compared to GOAL. Although the basic language elements are the same, there are a number of important differences. A key difference is that MARBEL agents use a single database instead of a more complicated cognitive state that is used by GOAL agents. There is no difference between knowledge and beliefs any more and MARBEL does not support goals. MARBEL further simplifies things as there is no need any more for specifying action specifications, or for writing percept rules.

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 called the *Tower World* where users can interfere, the *Wumpus World* environment as described in [37], and more challenging environments such as an elevator simulator, the Blocks World for Teams (BW4T) [29], and the real-time STARCRAFT gaming environment.

The language can be downloaded from `https://goalapl.dev`. This website also contains installation instructions for the MARBEL plugin for the Eclipse environment. Please help us to further improve the programming language MARBEL and its development environment by providing your feedback. You can contact me at k.v.hindriks@gmail.com.

In order to make the most out of this programming guide, as the typical advice goes when learning a programming language, the reader is advised to practice using the exercises that are part of it. 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 that you can find here to make the most out of the Eclipse plugin for developing agent programs.

Koen V. Hindriks, Utrecht, November, 2022

Acknowledgements
Getting to where we are now would not have been possible without many others who contributed to the development of the programming languages GOAL and MARBEL and its development environment. I would like to thank everyone who has contributed to this development, either by helping to implement the language, by developing the theoretical foundations, or by contributing to extensions. The list of people who have been involved one way or the other in this 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, Rien Korstanje, Nick Kraayenbrink, John-Jules Ch. Meyer, Peter Novak, Wouter Pasman, M. Birna van Riemsdijk, Tijmen Roberti, Dhirendra Singh, Nick Tinnemeier, Wietske Visser, and Wouter de Vries. Special thanks go in particular to Vincent Koeman, who not only developed Eclipse support for programming agents but also did a large part of the work of developing the language MARBEL itself.

Chapter 1

About Agents and Entities

Before we will see how we can write a simple chat agent program in Chapter 2, we first introduce the basic programming model of **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 what they believe. In this chapter we identify the key components and capabilities of an agent that enable it to effectively interact with its environment.

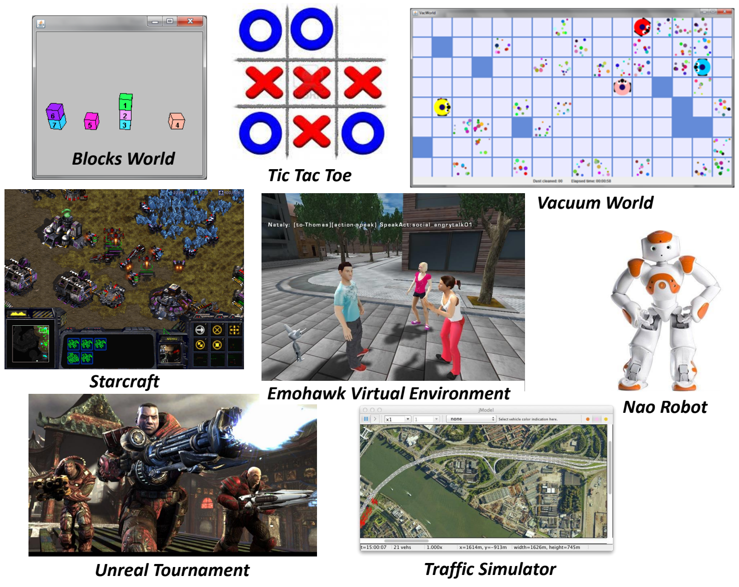


Figure 1.1: Example Environments

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 [37]. 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. The agent receives **percept**s that inform it about what the entity can see. Figure 1.2 illustrates the connection between an agent and an entity. A useful way of thinking about a controllable entity is that it is the body of the agent that controls it. In other words, you can think of the 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 agents as being themselves part of the environment but rather think of agents being connected to an (entity in an) environment.

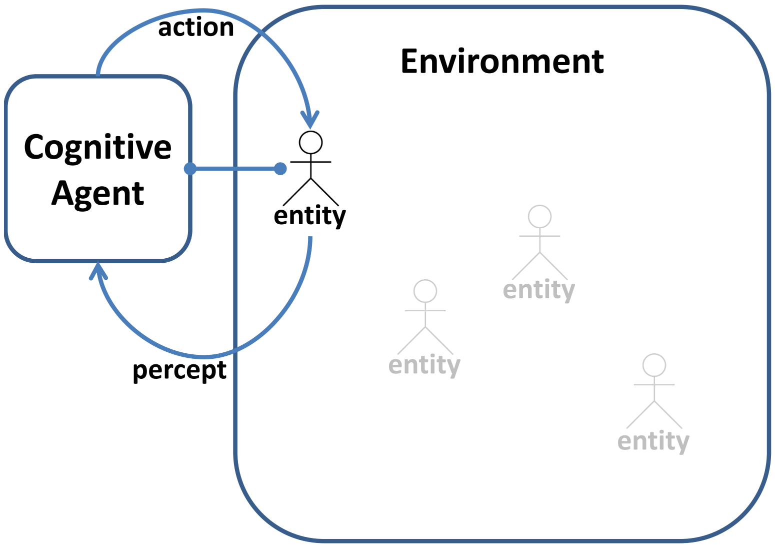


Figure 1.2: Agent Connected to Entity in Environment

It is 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 the 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 Agents

An 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 reconsider what it decides to do next. The second ability is **representing information**. 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.* The third ability is **decision-making**. This enables the agent to *reason about what it should do next* and *select an action to perform next.* These core abilities relate to important areas in artificial intelligence.

We will take much of the first ability of processing events of an agent for granted. Although in practice often, for example, computer vision may be needed to process raw camera images of a robot camera (one type of event, also called a *percept*), we will simply assume that some mechanism is available that transforms such input into symbolic information. In other words, we will take the process of extracting information 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 information 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 more important in this context is to understand 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, hears, etcetera.

For the second ability of representing information, we will 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 a domain. An agent needs a knowledge representation language for representing and reasoning about the environment it interacts with. Marbel agents use a knowledge representation language for representing percepts, beliefs, and messages that agents exchange. In the next chapter we will see how the language Prolog can be used to represent the state of an agent.

The third ability of an agent to make decisions is related to the area of 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.

1.3 Summary

The purpose of this chapter was to provide a brief introduction into **agents** that interact with an **environment** and to identify the key abilities that an agent needs to have to be able to interact effectively with its environment. An agent is connected to a **controllable entity** from which it receives percepts about the environment and which it can tell or direct what to do.
An agent has three core abilities to effectively interact with its environment:

- **event processing** which enables it to update its state using the events it receives;
- **representing information** which enables it to represent and reason about its environment;
- **decision-making** which enables it to select an **action** to perform next.

1.4 Notes

Although we have hinted at the possibility to construct systems of multiple 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.

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 describe 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 decisions that an agent that controls those entities might make.

Exercise 1.5.3

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 1.5.4

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

Chapter 2

First Agent Steps: "Hello World"

In this chapter, we write our first *agent program* in the programming language MARBEL. 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 walk through of the most important elements of the MARBEL language. We show how the "Hello World" agent can be made to print "Hello, world!" exactly 10 times by using module parameters to set a goal. Finally, we rewrite this agent and turn it into a simple script printing agent that, different from the simple "Hello, world!" agent, is connected to an environment. At the end of the chapter, you should have a basic understanding of the type of agent programs that you can write in MARBEL.

2.1 The MARBEL Agent Programming Language

MARBEL 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 and correspond with different core capabilities identified in the previous chapter. First, a **multi-agent system (mas)** file (`.mas2g`) is used for launching an agent system, connecting agents to entities in an environment, and for processing events by subscribing to channels. Second, knowledge representation files are used for representing information and to create the initial database of an agent. In our case we will use Prolog (`.pl` files). Third, **module** files (`.mod2g`) that contain rules are used for programming the decision-making capability 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. 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 running a multi-agent system.*

We assume you have started the MARBEL 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:

[1]See the User Guide for the Eclipse plugin [31]. A new mas file is automatically created when you create a new MARBEL project in Eclipse. This file is instantiated with a minimal template which you will need to complete.

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

The **launch** `helloWorldAgent` code snippet in the launch policy section is called a *launch rule*. 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 therefore must have a corresponding agent definition in the same mas file. A launch policy tells the platform *when* to create and launch an agent. The simple launch rule used here (we will later see other types of launch rules) can be read as an instruction to launch an agent using the agent definition for the `helloWorldAgent` agent whenever the mas is started.

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`. Do so by adding the following code *before* the launch section in the mas file:

```
define helloWorldAgent {

}
```

An agent definition specifies a **name** for the agent and consists of a definition section that specifies which module files should be used for creating the agent. An agent definition specifies a *type* of agent. It can be used, however, to create multiple agents by creating multiple instances of the agent definition. In our simple example we will only create a single agent instance. We will also use a single agent definition. Multiple agent definitions are allowed. But they make most sense, for example, when there are different types of entities in an environment that can be controlled. Then a different agent type can be used to connect an agent of that type to a corresponding type of entity in the environment. Note that even when we only have a single agent definition we need to include it 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.

In order to complete the agent definition above, we need to add at least one **use clause** that refers to a module file that will be used to create the agent. Complete the definition and make it look exactly like:

```
% Simple Hello World agent that prints "Hello, world!" once.
define helloWorldAgent {
   use helloWorld for decisions.
}
```

The definition above tells the system to use a module called `helloWorld.mod2g` when creating an agent called `helloWorldAgent` as the main module for making decisions. Note that the `mod2g` extension does not need to be included here. The decision 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 program is a set of modules.* The most important module that an agent uses to decide *what to do next* is called a decision module. Also note that comments can be added using the `%` sign.

We have not created the module file referenced yet and will do so now. Save the mas file and create a module file named `helloWorld.mod2g`; you may want to check out the 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 counter.

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 `counter` should be used to initialize the agent’s database. 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 a missing reference to the `counter` file. Do so by creating an empty `counter.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 MARBEL program, we can use an empty Prolog file which is only needed 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:

```
[helloWorldAgent] Hello, world!
agent 'helloWorldAgent' terminated successfully.
```

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 use clause in the module file.

2.3 Module Parameters for Setting Goals

We will now program another agent that is created from a different module file and extend the capabilities of our earlier `helloWorldAgent`. Our aim will be to write an agent that is able to print “Hello, world!”, for example, 10 times. This is not particularly exciting but will allow us to illustrate in more detail how queries in a rule are evaluated and how actions can be used for updating an agent’s database.

To get started, within the same project, create a new, second mas file and name it `MultipleLines`. As we have seen, a mas file must have at least one agent definition and a non-empty launch policy section. We will again call our agent `helloWorldAgent` but use a different module called `helloWorldMultipleLines` as the main decision module. To create this module, copy-paste the `helloWorld.mod2g` file and name it `helloWorldMultipleLines.mod2g`. Also add the following agent definition to the mas file and save the mas:

```
define helloWorldAgent {
   use helloWorldMultipleLines(10) for decisions.
}
```

Apart from the different module name used, there is another important difference with the agent definition for our previous agent. The module in our current agent definition *has a parameter*. A module can have one or more parameters. Parameters are instantiated with the parameter value everywhere in program section of a module. That is, the parameter is instantiated in all the rules in which it occurs with the parameter value. In our agent definition the parameter value is set to `10`. Although parameters can be used for various purposes, here we think of the parameter as *setting a goal*. The idea is that the parameter of the `helloWorldMultipleLines` module sets a goal to print the "Hello, world!" as many times as the the parameter indicates. In other words, "Hello, world!" should be printed 10 times in our example. There are other ways to set a target for an agent but parameters of modules provide one useful way for representing a goal.

To complete our mas file we still need to complete the launch policy. Because we want to create our agent immediately when the mas is run, as before, the launch policy section consists of a single instruction to **launch** our agent `helloWorldAgent`. Make sure the launch policy in the mas file looks as follows:

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

2.4 An “Hello World” Agent That Can Count

In order to complete the task of printing multiple lines our agent needs to keep track of progress. An “Hello World” agent that needs to output multiple lines needs to keep track of the number of lines that it already printed. We will introduce a counter to keep track of the number of times the agent has outputted something to the console. To represent the counter we introduce a predicate `printedText/1` with a single argument. An important requirement to be able to use this predicate is that we should **declare** it. We can do this by adding a declaration to the Prolog file `counter.pl` that we already created as follows:

```
% declare predicate that is used in a module but not defined here (by a Prolog rule).
:- dynamic printedText/1.
```

This time we want our agent to base its decision to print “Hello, world!” on its goal to print this line 10 times. We can do this by modifying the query used in the rule in the module that our previous agent used. **Rule**s of a MARBEL agent are of the form **if** ... **then** .... Rules for making decisions of this form can capture the key reason for doing an action in the query of the rule. Our “Hello World” agent, for example, should print “Hello, world!” if it has printed fewer than 10 lines.

Now open and make the following changes in the module file `helloWorldMultipleLines.mod2g`:

- Replace the exit condition **exit**=***always*** with **exit**=***noaction***; we want our agent to execute the module more than once and only terminate when there is nothing left to do.
- Rename the module to match the name of the module file and add a parameter `Lines`.
- Replace the query **true** with the query `printedText(Count), Count < Lines` in the action rule of the module .

You should now have a module that looks like:

```
use counter.

% terminate the agent if there is nothing more to do.
exit=noaction.

module helloWorldMultipleLines(Lines) {
   if printedText(Count), Count < Lines then print("Hello, world!").
}
```

Queries of rules are evaluated on the database of an agent. We should therefore make sure that the database contains facts of the form `printedText(Count)` with the variable `Count` instantiated with some number. To get started, we introduce a new module that we will use to initialize the agent’s database with the fact `printedText(0)`. We will add a simple rule for inserting this fact in the module’s program section. Now create a new module and name it `initCounter` and make sure it looks exactly like:

```
use counter. % contains a declaration of the printedText/1 predicate.

% initializes the printedText/1 counter to 0 in the database.
module initCounter {
   if true then insert( printedText(0) ).
}
```

We use the `counter.pl` file again to indicate the module uses Prolog, as before, but this time also because we use the `printedText/1` predicate which needs to be declared by a use clause at the beginning of the module. The module's program section consists of a single rule to **`insert`** the fact `printedText(0)` into the agent's database. We will use this module as an ***`init`*** module. An ***`init`*** module is the first module to be executed when the agent is launched. That means that the action rule in this module will be applied before any other rules are applied. As a result, the fact is inserted into the initial database that is created when the agent has just been launched.

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 `MultipleLines` mas file:

```
% Simple Hello World agent that outputs "Hello, world!" 10 times to the console.
define helloWorldAgent {
   use initCounter for init.
   use helloWorldMultipleLines(10) for decisions.
}
```

Let's see what happens if we run this version of our agent. Launch the mas file *in debug mode* (check the User Guide for how to do this), and pause the agent after a second or so. Now inspect the agent's 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 changed the exit condition in the module `helloWorldMultipleLines`. The other part which explains that this module does not terminate is that the agent is always able to perform the print action (and the ***`noaction`***exit condition never applies). The point is that the query will always succeed because the database is initialized with `printedText(0)` and is never updated. The agent, in other words, does not yet keep track of how many times it performed the print action. You can confirm this by inspecting the agent's database in the Introspector (check the User Guide) to see that the initial database of the agent has not been changed. The agent thus will continue inferring that it printed 0 lines.

Exercise 2.4.1

Replace the action in the module `initCounter` with an action that inserts the fact `printedText(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 Updating an Agent's Database

As we saw, the agent does not update its database every time that it performs 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 not for making decisions but for updating its database. The idea is that every time that an agent decided to perform an action by executing the module it uses for ***`decisions`***, it will call the update module. The main function of this update module thus is to update the agent's database after performing an action.

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.

% updates the printedText/1 counter in the database and increases it by one.
module updateCounter {
   if printedText(Count), NewCount is Count + 1
      then delete( printedText(Count) ) + insert( printedText(NewCount) ).
}
```

As before, the use clause is needed to declare the predicate `printedText/1` that is used in the module. The query of the rule is used to retrieve the fact that we need to remove from the database. The comma '`,`' is Prolog notation for "and". As a result, after evaluating `printedText(Count)`, an increase of the current counter by 1 is computed and bound to `NewCount`. The **`delete`** action removes information from the database. The **`insert`** action is performed to insert the updated information into the database. The **`+`** operator is used for combining one or more actions; actions combined by **`+`** are performed in the order in which they appear.

The `NewCount` and `Count` in the rule are Prolog variables. You can tell 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 database (when querying that database). For example, given that the database of the agent contains the following fact:

```
printedText(1246).
```

performing the query of the rule would bind 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 printedText(1246), 1247 is 1246 + 1
      then delete( printedText(1246) ) + insert( printedText(1247) ) .
```

Finally, we need to add the module `updateCounter` to the agent definition so the agent will use it. We do so by adding another use clause that tells the agent to use it as an ***`updates`*** module, to make sure it looks like:

```
% Simple Hello World agent that prints "Hello, world!" message 10 times.
define helloWorldAgent {
   use initCounter for init.
   use updateCounter for updates.
   use helloWorldMultipleLines(10) for decisions.
}

launchpolicy{
   launch helloWorldAgent.
}
```

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

Exercise 2.5.1

What happens if we change the number of lines that the agent wants to print in the mas file (by changing the parameter of the `helloWorldMultipleLines` module)? For example, change the value of the parameter from 10 to 5 and run the mas again.

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 an action or 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.* MARBEL 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 `MultipleLines.mas2g` and give this file the name `ScriptAgent.mas2g`. Add the following use clause at the start of this new mas file:

```
% launches a window that can be used by the agent to print text to.
use "HelloWorldEnvironment-1.3.0.jar".
```

The next step is to make sure that our “Hello World” agent is connected to this environment. An environment makes available one or more *controllable entities* and agents can be connected to these entities. Once connected to an entity the agent controls the actions that the entity will perform. When the `HelloWorldEnvironment` is launched it makes available one entity. When this happens the agent platform is informed that an entity has been created. An agent then can be connected to that entity using a **launch rule** in the 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 $\star$ matches with any entity. When triggered, the agent `helloWorldAgent` is created using the agent definition, and connected to the entity.

The main thing that remains is to start using the new `printText` service made available by the environment. To do so, do the following:

- copy and rename the `helloWorldMultipleLines` module to `printScript`, and update the name of the module inside the file,
- replace the **`print`** action in the module’s rule with `printText`, and
- replace the reference to `helloWorldMultipleLines(10)` in the `ScriptAgent` mas file with `printScript(11)` (why we use 11 here will become clear below).

The module `printScript` should look like:

```
use counter.

exit = noaction.

module printScript(Lines) {
   if printedText(Count), Count < Lines then printText("Hello, world!").
}
```

Let's run the new mas file. But before you do so make sure that you checked the 'Percepts processed' in the MARBEL logging preferences (Window → Preferences → MARBEL → Logging). Launch the mas again (in debug mode) and run it. By adding the environment and connecting the agent to an entity, the agent now printed something to a window that was created when the mas was started. In the agent's console, moreover, we now also see:

```
[helloWorldAgent] percepts processed, removed [] and added [].
```

This means that the agent now also engages in percept processing but does not yet receive anything from the environment. The agent will receive percepts if it subscribes to the corresponding **channels**. For each type of percept, a channel is available that the agent can subscribe to. Which channels are available depends on the environment. The `HelloWorldEnvironment` provides two channels: one for `lastPrintedText/1` and one for `printedText/1` percepts. An agent can subscribe to these channels by adding the following code below the use clauses for modules in the agent definition in the mas file:

```
        replace lastPrintedText/1.
        replace printedText/1.
```

Add these lines to the mas file and run the mas file again in debug mode. When you now inspect the agent's console again, you should be able to see the following:

```
[helloWorldAgent] processing percepts...
[helloWorldAgent] inserted 'lastPrintedText(Hello, world!)' and
               deleted 'lastPrintedText()'.
...
[helloWorldAgent] deleted 'printedText(1)'.
[helloWorldAgent] inserted 'printedText(2)'.
```

This means that by subscribing to the percept channels, the agent's database is now automatically updated with new information from the environment! As you can see, the environment keeps track itself of how many times something has been printed, i.e., how often the `printText` action was performed, and informs the agent of this by means of the 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!" 11 times the environment informed the agent it printed it only 10 times... What happened? Recall that the update 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 updating the database incorrectly inserted `printedText(11)` into the agent's database!

We can learn an important lesson from this: *whenever possible use percept information from the environment to update an agent's database.* To follow up on this insight we will use percepts to update the agent's database instead of the rule in the `updateCounter` module we used before. We simply do this by removing the use clause for this module in the `ScriptAgent` mas file! We also remove the clause for the init module as the environment also provides this initial information. After removing both use clauses, run the mas again in debug mode and check the database of the agent to verify that the agent printed "Hello, world!" 11 times.

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 different lines. 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 line 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 lines 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. 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 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 a second use clause to the `printScript` module:

```
use script.
```

We need one more modification before we can use the script. We need to include `script/2` in the query of the rule in the module. In order to figure out which line in the script we need to print, the agent should query its database. The idea is that the agent retrieves the number of lines printed, adds 1 to that number to obtain the next line number, and retrieve the corresponding text from the database using the `script/2` 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. A variable `Text` is used to retrieve the correct script line from the database 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 printedText(Count), Count < Lines, NextLine is Count+1, script(NextLine, Text)
   then printText(Text).
```

Run the script agent to verify that the script is printed. We can now also explain why we used 11 to instantiate the `printScipt` module’s paramteter: the script has 11 lines.

Chapter 3

The Order of Things

To be able to make the right decisions, an agent needs to reason about the state of the environment it interacts with. For this purpose, an agent maintains a database. This database is also used for keeping track of the current state of the environment. In this chapter we will look in more detail at how we can use **Prolog** for reasoning about and keeping track of the state of environments. The **Blocks World** environment is used as the running example to illustrate the concepts introduced.

An agent needs a **strategy** for achieving its goal in life. We discuss a simple strategy for solving Blocks World problems. We show how this strategy can be implemented by a straightforward translation into **action rules** in a Marbel agent program. Action rules are rules that consist of a query and an action. The queries are Prolog queries evaluated on the agent's database. One of the main jobs of a Marbel programmer is to come up with a set of Prolog rules that can be used to specify the queries that are needed to implement action rules. The actions in action rules in this chapter are **environment actions** for changing the state of the environment.

3.1 The Blocks World Environment

As a running example in this chapter, we will use one of the most famous environments in Artificial Intelligence known as the **Blocks World** [37, 41, 46]. In the Blocks World, there is a single robotic gripper entity that can move and stack cube-shaped blocks on a table. The gripper has access to the full configuration of blocks on the table. It can "see" for each block whether it is sitting on the table or on top of another block, and, if so, which block it is sitting. The goal is to move blocks from an *initial state* to a *goal state*, as illustrated in Figures 3.1 and 3.2. For example, the block numbered 6 should sit on top of block 4 according to the goal state instead of sitting on 7 as it does in the initial state. Note that in our version of the Blocks World all blocks are numbered. A block with a particular number thus is sitting on top of a block labelled with another number (if not on the table). In our simple visualization of the Blocks World in Figures 3.1 and 3.2 neither the table nor the gripper is visible (in the Tower World that we will use in the next Chapter 4 we will see a gripper moving while holding a block).

In one of its most simple (and most common) forms, a block can sit directly on top of at most one other block. A block thus is part of a stack of blocks and either located on top of a single other block, or it is sitting directly on the table. To simplify things a bit further, we will assume that the table is large enough to place all the blocks that are present in a state directly on the table. Another way of saying this is that there is always room on the table to put a block on it. The table thus always has a *clear* spot for a block. The robotic gripper can perform a single action of moving one block (onto another block or the table). The gripper thus is limited to moving at most one block at a time and cannot move more than one block simultaneously let alone that it can move a stack of blocks. The gripper action of moving a block, moreover, can only be performed if there is no other block sitting on top of it. Moving a block onto another block only can be performed if there are no other blocks sitting on either of these blocks. If these conditions are

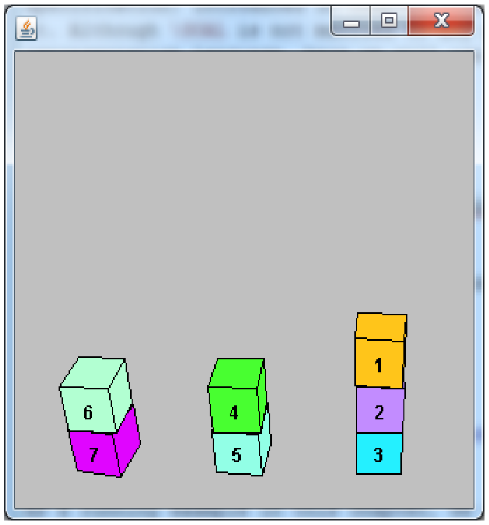


Figure 3.1: Initial State

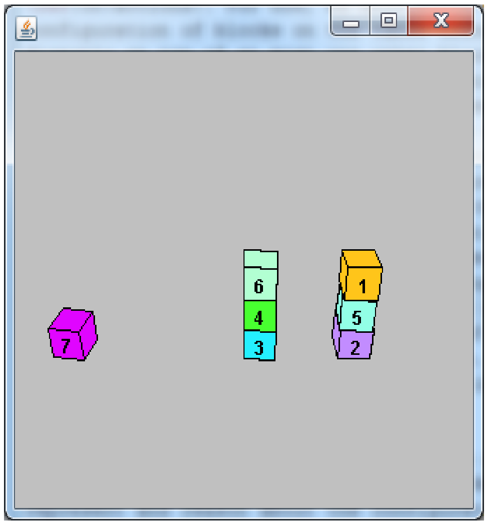


Figure 3.2: Goal State

met, a move action will be successful. If there is no other block sitting on a block, we also say that that block is *clear*. Note that moving a block to the table only requires that the block is clear, as we assume that the table is always clear. These are some of the basic "laws" or constraints of the Blocks World that we will need to take into account.[1]

A *Blocks World Problem* is the problem of how to move blocks from an initial state to a goal state, taking the constraints of the Blocks World into account. To solve a Blocks World Problem we need to figure out how we can reach a goal state by moving one block at a time onto another block or the table. Figures 3.1 and 3.2 illustrate an example Blocks World Problem. We do not care about the exact positioning of a stack of blocks on the table. We do not care, for example, whether the stack of blocks on top of block 3 is located to the left or to the right of the stack of blocks on top of block 2 in the goal state of Figure 3.2. The Blocks World is a toy problem domain, in both senses of the word. That does not mean that they are simple. Even Blocks World Problems can be surprisingly hard to solve optimally. An *optimal solution* of a Blocks World Problem uses the minimal number of moves to reach a goal state.

[1]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].

Blocks World Problems Can Be Hard

In this chapter, we will develop a *good*, but not an *optimal*, strategy for solving Blocks World Problems. The *performance* of an agent that solves Blocks World problems is measured by the number of moves it uses to reach a goal state from an initial state or configuration. 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 get 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* [41]. 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 [41].

It will be useful to briefly discuss how we can use a list of numbers for compactly representing a configuration of blocks in a state. The idea is that we can use the Nth position in or index into the list to indicate on top of which other block block N is sitting. In case block N is sitting on the table, we will put a 0 in the list at index N. For example, the list $[2, 3, 0, 5, 0, 7, 0]$ represents the initial state in Figure 3.1. The first index indicates that block 1 is sitting on top of block 2. The second index indicates that block 2 is sitting on top of block 3. The third index indicates that block 3 is sitting on the table. Etcetera. Please check for yourself that the list $[5, 0, 0, 3, 2, 4, 0]$ represents the goal state in Figure 3.2. Although lists of numbers are a compact way of representing block configurations, we need to be a little careful because not all lists of numbers represent feasible block configurations. The list $[5, 4, 1, 0, 3]$, for example, requires block 1 to sit on 5, 5 to sit on 3, and 3 to sit on 1. This is not a feasible configuration as it requires block 1 to be part of a stack twice! A list of numbers should not have cycles that require a block to be above itself...

Environment Properties of the Blocks World

The Blocks World is **fully observable** as the gripper can always "see" the complete configuration of blocks. As the only goal in life of an agent trying to solve a Blocks World Problem consists of moving blocks, an agent connected to the gripper will be able to see everything that is relevant for its decision-making.
The effects of the single move action performed by the gripper are completely predictable. The move either is successful, if the block that is moved and the target location are clear, or fails (and nothing changes). This means that the Blocks World is **deterministic**.
The only changes that can happen in the Blocks World are those that are the result of performing a move action by the gripper. The Blocks World thus is a **static** environment, where an agent connected to the gripper has full control.
The Blocks World is **discrete** rather than continuous because there is only a *single* action that can make changes to *finite* configurations of blocks. In our version of the Blocks there are no continuous items (such as water flowing from a tap) or continuous changes (such as a bath tub slowly getting filled by a running tap).
Finally, the Blocks World is a **single agent** environment as there is only a single entity (the robotic gripper) that can make changes to the environment. It is also is the only entity that can be controlled by an agent program.

3.2 Creating a Blocks World mas

Let's get started by creating a **mas project** for the Blocks World. Creating such a project always starts by creating a mas file. In this mas file, we need to specify which *environment connector* we want to use, we need to provide at least one *agent definition*, and specify a policy for launching an agent. We use the following program file called `BlocksWorld.mas2g`:

```
use "blocksworld-1.3.0.jar"
   with start=[2, 3, 0, 5, 0, 7, 0].

define stackBuilderAgt {
   use stackBuilder([5, 0, 0, 3, 2, 4, 0]) for decisions.
   use mapBWnames for updates.

   % percept processing
   replace on/2.
}

launchpolicy {
   when * launch stackBuilderAgt.
}
```

Figure 3.3: mas file for connecting an agent to the Blocks World gripper

At the beginning of a mas file a single *environment use clause* specifies the file that will be used as connector. In our case, we use the `blocksworld-1.3.0-jar` file. This Blocks World connector will also launch the Blocks World simulator (a window displaying the current blocks configuration). The connector reference following the `use` command in a mas file can be any path name to an environment connector file relative to the location of the mas file itself in the file system where a connector file can be found. A connector file should be a `jar` file.

An environment use clause can have an additional *with clause.* A with clause is used to set one or more *initialization parameters* of an environment. The values used for setting these parameters should follow the **with** keyword. In our mas file, the value of a parameter named `start` is set to a representation of the environment's initial configuration of blocks, using the list representation we discussed above. This parameter is passed onto the connector when the environment connector is launched. Which parameters can be set depends on the particular environment used.

The environment use clause is followed by a single agent definition in our mas file. 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. As we have seen already in Chapter 2, a use clause in an agent definition can have three use cases: it can be used either **for *decisions***, **for *updates***, or **for *init***ialization purposes. An agent definition must have at least one module that is used **for *decisions*** or **for *updates***. In this case, our agent definition has two use clauses. The first use case specifies that the `stackBuilder` module will be used **for *decisions***. This module is the module that will be used to make the agent's decisions on what to do next. As in Chapter 2, this module has a parameter used to *set the agent's goal.* The list that sets the goal specifies the desired block configuration. The second use case specifies that a module called `mapBWnames` will be used **for *updates***. As we will see below, the `stackBuilderAgt` agent will use this module for preprocessing the percepts it receives from the environment.

Percepts an agent wants to receive from the environment also need to be specified in its definition by means of channel clauses. The channel clause in our definition specifies that the wants to receive `on/2` percepts and that the channel handler **replace** should be used for processing these percepts. This handler replaces all previously received percepts of the same type with the new ones (if any) that are received each agent cycle.

Finally, the mas file contains a **launch policy** section with one **launch rule** for launching an agent. This rule will launch an agent named `stackBuilderAgt` using the corresponding agent definition when the gripper entity becomes available in the Blocks World environment. The agent name in a launch rule must match with a name used for defining an agent.

3.3 Representing and Reasoning about the Blocks World

After defining a mas project, the next step in developing and writing a Marbel agent program is to design the Prolog facts and rules that an agent can use for reasoning about its environment. This is needed next as we will need to provide a use clause referencing a Prolog file in any module that the agent uses. We will use SWI Prolog [43] for writing a Prolog program (facts and rules).

An important task is to select suitable labels or predicates for talking about the environment. In general, this task need not be finished in one go. Several iterations may be needed in which new labels are added or modified during the design of an agent program. This groundwork is important as the action rules that an agent uses for decision-making depend on it.

To get started, the percepts that are received from an environment typically provide a useful starting point. One of the most basic concepts in the Blocks World is that a block is sitting *on top of* another block or is *on* the table. The Blocks World environment provides the predicate `on/2` to represent this:

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

Facts of the form `on(X,Y)` express that *block* `X` *is (directly) on top of* `Y`. For example, `on(1,2)` can be used to represent the fact that block 1 sits on block 2 and `on(2,0)` can be used to represent that block 2 is on the table. In Figure 3.1, `on(1,2)` is the case and `on(2,0)` and `on(1,3)` do not hold. `on(1,3)` 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 to realize that only *blocks* can be on top of something else in the Blocks World. Whenever `on(X, Y)` holds `X` thus must be a block. `Y` however does not need to be a block but may also refer to the table. As we have discussed above, in the Blocks World 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/2` in "the right way".

As a rule of thumb, it is a good idea to introduce predicates that correspond with the most *basic concepts* in a domain first. More complicated concepts then may be *defined* in terms of the more basic concepts. We will use this strategy here as well.[2] 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 only blocks can sit on something else. Therefore, if `on(X,Y)` holds it must be that `X` is a block. We can use this to define the `block` predicate by the following Prolog rule:

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

For this definition to work and to identify all blocks, the agent must have all the facts about which blocks are sitting on top of another block or the table in a Blocks World configuration. As the Blocks World is fully observable, we may safely assume this is the case.

The `block/1` predicate can be used to define another useful predicate `clear/1`. We will use `clear(X)` to infer that a block `Y` can be moved on top of `X` (as long as `Y` is not bound to the same value as `X`). That is, if a block is clear, another block can be moved on top of it. The table is always clear in this sense, as in our version of the Blocks World the table always has room to place a block on it. We can capture this by the following Prolog fact and rule:

[2] 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.

```
clear(0).                                % 0 represents the table
clear(X) :- block(X), not( on(_, X) ).
```

Floundering The rule for defining the `clear/1` predicate can be used to illustrate an important issue that needs to be taken into account when writing Prolog code. The issue is that variables in a negated fact need to be sufficiently instantiated. Reversing the conjuncts in the rule that defines our `clear/1` predicate, for example, would cause a problem. If the rule

```
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 at least one and at most a finite number of blocks, however, there must at least be one block that is clear (only one block would be clear if all blocks were stacked on each other in a single stack). What goes wrong in the rule above 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 rule for the `clear/1` 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 available in the agent's database.

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.

An agent can only keep track of the complete state of its environment if that state is *fully observable.* The lesson here is that properties of the environment need to be carefully taken into account when defining Prolog rules for reasoning about the environment. 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)` might need to be revised or discarded.

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, 0).                        % 0 represents the 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 when `X` sits on the table, i.e., when `on(X,0)` holds. This part of our definition 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 when `[X|T]` is a tower according to our rules, this does *not* mean that block `X` is clear. In other words, any (sub)stack of blocks that is part of a larger stack is a tower. We can, for example, derive `tower([2,3])` 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 will work well.

Declaring Predicates that are Used but not Defined It is important to *declare* all predicates that are *used but not defined* in a Prolog file are **declared** so that the agent knows which predicates it can use in its program. A predicate is used if it occurs anywhere in the agent program in a query (in an action rule) or in the body of a rule (in a Prolog file). A predicate is not defined if it does not occur in the head of a Prolog rule. A predicate can be declared by means of the `:- dynamic` directive. In our code thus far we have used only one predicate that has not been defined: `on/2`. We can declare it by the statement `:- dynamic on/2.`

To conclude, we can combine all Prolog facts and rules as well as the declaration introduced above in a single Prolog file that we can use in the modules we will code. Because we will use this Prolog program later, create a new file and name it `blocksworld.pl`.

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

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

% the table is always clear.
clear(0).
% a block is clear if nothing is sitting on top of it.
clear(X) :- block(X), not( on(_, X) ).

% a tower is any stack of blocks that sits on the table.
tower([X]) :- on(X, 0).
tower([X, Y| T]) :- on(X, Y), tower([Y| T]).
```

3.4 Preprocessing Percept Information

We have assumed that Blocks World configurations can be represented by a list of numbers. But what kind of percepts does the environment provide? More specifically, what kind of naming convention for blocks does the environment use?

Let's try to find out more by running a Blocks World mas. Because we only want to find out more about the environment's naming convention for blocks, we first write some dummy code for the two modules `stackBuilder` and `mapBWnames` which are used in the agent definition:

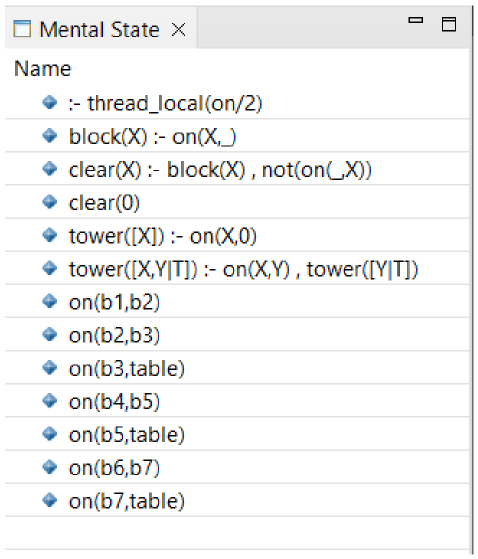
```
% stackBuilder module
use blocksworld.

module stackBuilder(Goal) {
   if true then insert(true).
}
```

```
% mapBWnames module
use blocksworld.
```

```
module mapBWnames {
   if true then insert(true).
}
```

Each of the modules uses the Prolog file `blocksworld.pl` as indicated by the *use clauses* at the start of the module file. A use clause has the form **use** *id*. where *id* is the name of a Prolog file. Note that the file's extension should not be included in a use clause. In our case we thus simply write **use** `blocksworld.` Note that a use clause must be followed by a dot. As can be seen in Figure 3.4, the Prolog file is added to the agent's database. Use clauses are not the only means available for initializing an agent. In Chapter 2 we saw that rules in an ***init*** module can also be used for initializing the state of an agent.

Note that for the `stackBuilder` module we also need to specify a parameter. As we are using this parameter to pass goal state information to this module, we name the parameter `Goal`. Obviously, the action rules of both modules will do nothing useful. (The **insert**(**true**) action does not change an agent's database and thus can be viewed as performing a 'skip' action.) But they are sufficient for our purpose to run the mas file. If we launch our mas project with these module files in *Debug Mode* in the Eclipse editor, the agent and environment are initially paused and we can inspect the agent's state:



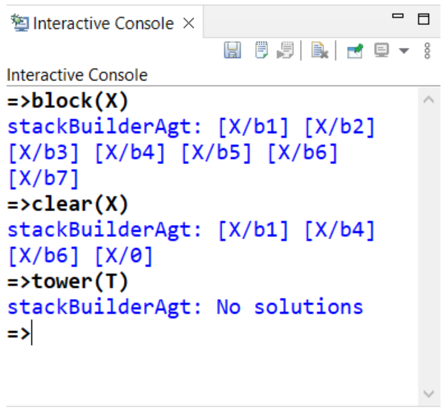
```
Name
:- thread_local(on/2)
block(X) :- on(X,_)
clear(X) :- block(X) , not(on(_,X))
clear(0)
tower([X]) :- on(X,0)
tower([X,Y|T]) :- on(X,Y) , tower([Y|T])
on(b1,b2)
on(b2,b3)
on(b3,table)
on(b4,b5)
on(b5,table)
on(b6,b7)
on(b7,table)
```

Figure 3.4: contents of the Blocks World agent's database after initial launch

The state of the agent in Figure 3.4 shows that the initial database of the agent contains all Prolog facts and rules that we introduced above. That is the good news. The bad news, however, is that it also shows that the percepts received on the `on/2` channel use slightly different names for blocks than we have assumed thus far. Instead of simple numbers, we get facts like `on(b3,table)` which refer to block 3 by `b3` and instead of 0 use `table` to refer to the table. Evaluating a few queries in the Interactive Console highlights some further issues, as can be seen in Figure 3.5. The `block(X)` query still works but now returns the block labels as they are received from the environment. The `clear(X)` query does the same for blocks that are clear but also says that `0` is clear, which we use to represent the table. Because of this, the `tower(T)` query returns no solutions.

Exercise 3.4.1

Why exactly does the `tower(T)` query return no solutions? Provide a detailed explanation.



```
=>block(X)
stackBuilderAgt: [X/b1] [X/b2]
[X/b3] [X/b4] [X/b5] [X/b6]
[X/b7]
=>clear(X)
stackBuilderAgt: [X/b1] [X/b4]
[X/b6] [X/0]
=>tower(T)
stackBuilderAgt: No solutions
=>
```

Figure 3.5: results of several queries in the Interactive Console

We need to fix this mismatch between the numbers that we have used thus far and the labels used by the environment. We will write a Prolog rule for mapping environment labels to the numbers that our agent expects it needs to work with and use this in the update module `mapBWnames` to fix the issue. Later we will also need to map numbers back to labels so we also introduce a rule for that here. We first introduce the Prolog fact and rules:

```
label_to_number(table, 0).
label_to_number(Label, Num) :-
    not( Label=table ), atom_concat(b, Atom, Label), atom_number(Atom, Num).

number_to_label(Num, Label) :- atom_concat(b, Num, Label).
```

The first Prolog rule uses the built-in Prolog operators ***atom_concat***/3 and ***atom_number***/2. To illustrate the first operator, we have, for example, that the query ***atom_concat***`(b, 1, b1)` succeeds. This predicate thus can be used to effectively remove all the `b`'s in the block labels used by the environment. The other operator maps atoms to numbers, which is required to make sure that the right data type is used (integers instead of the atom type that is specific to Prolog). It is possible to use most of the built-in operators of the SWI Prolog system [43] in a MARBEL agent program.

Exercise 3.4.2

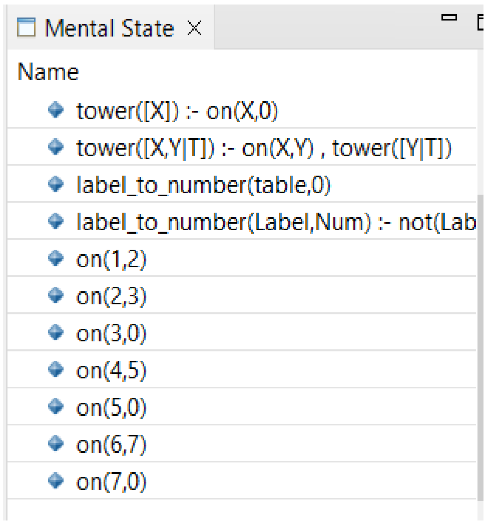
Why can't we use the `label_to_number/2` predicate too to compute the `Label` when we only have the number `Num`? *Hint*: Try evaluating the `atom_concat(b, Num, Label)` query without instantiating the `Num` variable in Prolog.

The purpose of the `mapBWnames` module is to preprocess the percept information and transform it into the number representation that we have assumed so far. To achieve what we want, we need to use a new type of action rule of the form **forall** `<query>` **do** `<action>`:

```
use blocksworld.

module mapBWnames {
   forall on(L1, L2), label_to_number(L1, N1), label_to_number(L2, N2)
      do delete( on(L1, L2) ) + insert( on(N1, N2) ).
}
```

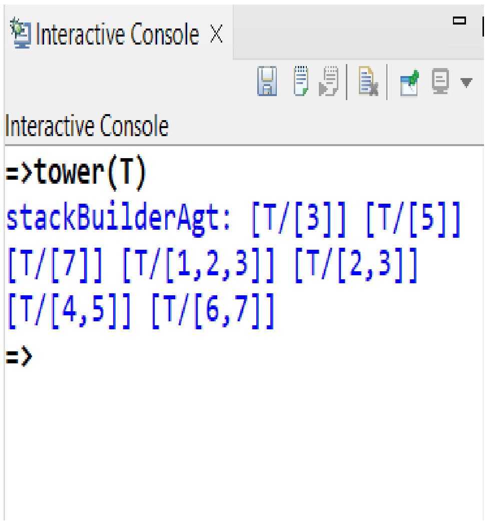
Figure 3.6: new module file for the `mapBWnames` module

The reason why we need a different type of action rule is that action rules of the form **if** `<query>` **then** `<action>` that we have used before only are applied for one of the possible bindings that the query `<query>` returns. In this case, to fix our issue, we need to delete all `on/2` facts received from the environment and replace each of these with a new `on/2` fact that uses numbers for representing a Blocks World configuration. In other words, we want to perform the `<action>` for all bindings that are returned by the query used in the `mapBWnames` module's rule. This is exactly what the **forall** `<query>` **do** `<action>` rule does. This rule is evaluated by first computing all the bindings for `<query>`, then instantiate the action template `<action>` with each of these bindings, and then perform all of the resulting actions. You can check by yourself by stepping through the agent program with the new `mapBWnames` module (also make sure to add the Prolog fact and rule for predicate `label_to_number/2` to the Prolog file) that this yields exactly the database that we were looking for, as can be seen in Figures 3.7 and 3.8.



```
tower([X]) :- on(X,0)
tower([X,Y|T]) :- on(X,Y) , tower([Y|T])
label_to_number(table,0)
label_to_number(Label,Num) :- not(Lab
on(1,2)
on(2,3)
on(3,0)
on(4,5)
on(5,0)
on(6,7)
on(7,0)
```

Figure 3.7: Database



```
=>tower(T)
stackBuilderAgt: [T/[3]] [T/[5]]
[T/[7]] [T/[1,2,3]] [T/[2,3]]
[T/[4,5]] [T/[6,7]]
=>
```

Figure 3.8: Interactive Console

3.5 A Strategy for Solving Blocks World Problems

Our agent needs a strategy for solving Blocks World problems. Although we have seen that optimally solving such problems is hard, it is possible to specify simple strategies that also perform very well. The strategy that we will implement is based on two key concepts: *constructive moves* and *misplaced blocks*. A move of one block on top of something else is said to be *constructive* if it moves the block into the position we want it to be. This means that after making such a move we will never have to move the block again to achieve the goal state. If, for example, we already have part of a tower that we want, where block 2 is on the table and 5 sits on 2, then moving block 1 on top of 5 would be a constructive move (check out the goal we specified in the mas file). A block is *misplaced* if it sits on top of a tower but we don't want such a tower with that block on top of it (it does not match with the goal we have). In the initial state in Figure 3.1, all blocks except for blocks 3 and 7 are misplaced.

It should be clear from this discussion that we also need to be able to check which blocks should be on which other blocks in the goal state to identify constructive moves and misplaced blocks. It is moreover useful and necessary to also be able to identify towers in the goal state. Recall that the goal state is passed into the `stackBuilder` module by means of the parameter `Goal`. To reason about this goal state, we will introduce two new tertiary predicates `on/3` and `tower/3` and use these to define constructive moves and misplaced blocks:

```
% block X sits on top of Y in the goal state Goal.
on(Goal, X, Y) :- nth1(X, Goal, Y).

% tower(Goal, T) holds if T is a tower in the goal state Goal.
tower(Goal, [X]) :- on(Goal, X, 0).
tower(Goal, [X,Y|T]) :- on(Goal, X, Y), tower(Goal, [Y|T]).

% moving block X on top of Y is constructive.
constructive(Goal, X, Y) :- tower(Goal, [X,Y|T]), tower([Y|T]).

% block X is misplaced.
misplaced(Goal, X) :- tower([X|T]), not( tower(Goal, [X|T]) ).
```

For defining `on/3`, we use the built-in Prolog predicate ***nth1***`(Index, List, Element)` which succeeds if the element in the list `List` at index `Index` unifies with `Element`. Here we exploit our representation of block configurations by means of a list of numbers. As a result, we have that `on(Goal, X, Y)` succeeds if the Xth element in the list `Goal` is `Y`. In other words, block `X` sits on `Y` in the goal state. The `tower/3` predicate is defined analogous to the `tower/2` predicate but now also carries the goal state as its first parameter.

The Prolog rule for `constructive/3` states that moving `X` on top of `Y` is constructive if we already have a tower `[Y|T]` and we want tower `[X,Y|T]`. For example, if we already have tower `[5, 2]` and we want `[1, 5, 2]`, then moving block 1 on top of 5 is constructive. The Prolog rule for `misplaced/2` states that a block `X` on top of a tower `[X|T]` that we have but don't want is misplaced.

3.5.1 Constructive Moves and Moving Misplaced Blocks to the Table

Using our definitions above, we can now specify a simple strategy: move misplaced blocks to the table, and, if possible, make a constructive move. Clearly, such a strategy will achieve the goal state, as, in the worst case, it will first move all (misplaced) blocks to the table, and then start building the towers we want by means of making constructive moves. It is also not difficult to translate this into two action rules which we can use to program the `stackBuilder` module. Apart from the fact that we need to check that making a move is possible, we also should not forget to map the numbers we use for representing blocks back to the labels the environment recognizes. Moreover, we check that a block not already sits on the table to prevent trying to move such a block if it is still misplaced again to the table. A final subtle twist is that we have to delete the fact `on(X, Z)` that block `X` sits on `Z` which is no longer true after moving block `X` (this is no longer updated by percept processing!). We thus obtain the following action rule for handling misplaced blocks:

```
if misplaced(Goal, X), clear(X), not( on(X, 0) ), on(X, Z), number_to_label(X, L)
        then move(L, table) + delete( on(X, Z) ).
```

and the following action rule for making constructive moves:

```
if constructive(Goal, X, Y), clear(X), clear(Y), on(X, Z),
      number_to_label(X, Lx), number_to_label(Y, Ly)
then move(Lx, Ly) + delete( on(X, Z) ).
```

Exercise 3.5.1

Why does the agent need to delete the fact `on(X, Z)` that a block `X` sits no longer on `Z` but does the agent not need to add the new fact that `X` sits on the table (after applying the rule for misplaced blocks) or sits on `Y` (after applying the rule for constructive moves)?

Now we have the action rules that implement the key elements of our strategy for solving Blocks World Problems, the only question that remains is in which order we should place these rules in the `stackBuilder` module. The issue is not that order matters for successfully achieving the goal state or not. The question is rather which order yields the best performance. By testing both cases: putting the rule for handling misplaced blocks first, or putting the rule for making constructive moves first, we find find that we should put the rule for constructive moves first. In that case the agent only needs 7 moves to achieve the goal state, whereas it needs 9 otherwise.

Exercise 3.5.2

Why is putting the rule for constructive moves before the rule for handling misplaced blocks more efficient than vice versa?

By putting everything together, we get the following `stackBuilder` module:

```
use blocksworld.

exit=noaction.

module stackBuilder(Goal) {
   if constructive(Goal, X, Y), clear(X), clear(Y), on(X, Z),
        number_to_label(X, Lx), number_to_label(Y, Ly)
   then move(Lx, Ly) + delete( on(X, Z) ).

   if misplaced(Goal, X), clear(X), not( on(X, 0) ), on(X, Z), number_to_label(X, L)
          then move(L, table) + delete( on(X, Z) ).
}
```

3.6 Tracing the Execution of the Blocks World agent

Now that we have created all files that we need in our mas project, including the environment file as well, in a single project folder, we are ready to execute the mas again. You can run the mas (see the User Manual [31] on how to do this) but the result is too fast gone again to see what happened. It is more useful to start the mas in Debug Mode (again, check the User Manual on how to do that). If you launch the mas in Debug Mode, the system is initially paused. This allows you to inspect the state (database) 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 `stackBuilderAgt` and that an environment process called `blocksworld-1.3.0` has been launched. Both agent and environment should be paused.

It is useful to **trace** the behaviour of the agent that we have created once in detail to understand what happens. First inspect the state of the agent. You should see the contents of our Prolog file as well as the initial percepts that we received from the environment. The `on/2` facts in the database of the agent represent the initial Blocks World state in Figure 3.1.

If you continue stepping through the agent program, you will see that the `mapBWnames` module is entered. By stepping further you see that the facts in the database are replaced by the rule in this module by `on/2` facts that use numbers for blocks instead of labels such as `b1`. After completing the application of the **forall... do...** rule, the agent steps into the `stackBuilder` module for making decisions. You will find that the first action rule in this module is not applicable. There is no constructive move that the agent can make. The second action rule is applicable and will select to move block `1` to the table. If you continue stepping you will find that 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 action rule for misplaced blocks again and move block `2` to the table. This move makes it possible to move block 4 onto block 3. In the third execution cycle the agent will now apply the first action 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 6 onto block 4.

Finally, in cycle 7, the goal has been achieved and the agent finds that there are no actions selected any more in its decision module. At that point, no new cycle is started but the agent is terminated because of the exit condition ***noaction*** that we added in the `stackBuilder` module.

3.7 Looking Only Once

The Blocks World environment has very specific properties that give the agent full control over it. Because all changes must be the result of actions that the agent performs (the Blocks World is static and single agent) and the effects of actions are completely predictable (the Blocks World is deterministic), the agent can compute the next state that results from performing an action itself. In other words, it does not have to look at the environment state and can compute this state without receiving any percepts. We are going to illustrate this by making a few small changes to our agent program.

The first change that we make is to change the type of the `mapBWnames` module. We are going to make this an ***init*** instead of an ***updates*** module. We can do so by replacing ***updates*** with ***init*** in the mas file. The idea is that we only have to look once and only have to preprocess the initial percepts that we receive from the environment.

The second change that we need to make is to make sure the agent now itself inserts a new fact after making a change to the blocks configuration by moving a block. We will add an **insert**action with the relevant fact to both of the action rules in the `stackBuilder` module, as follows:

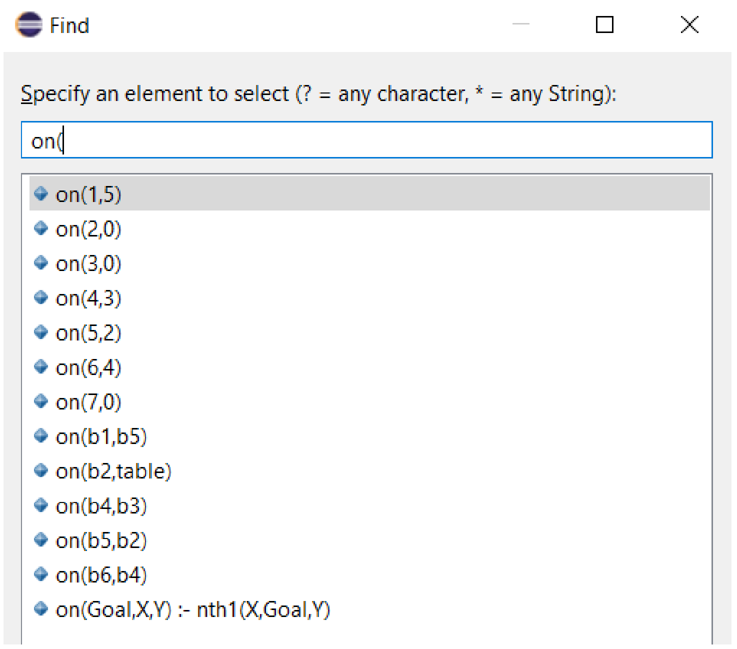
```
use blocksworld.

exit=noaction.

module stackBuilder(Goal) {
   if constructive(Goal, X, Y), clear(X), clear(Y), on(X, Z),
        number_to_label(X, Lx), number_to_label(Y, Ly)
   then move(Lx, Ly) + delete( on(X, Z) ) + insert( on(X, Y) ).

   if misplaced(Goal, X), clear(X), not( on(X, 0) ), on(X, Z), number_to_label(X, L)
         then move(L, table) + delete( on(X, Z) ) + insert( on(X, 0) ).
}
```

After making these changes, run the mas again in Debug Mode and inspect the database of the agent. Select a fact in the database, use the Find function (CTRL+F) and search for `on(` in the search field. You should see the facts as displayed in Figure 3.9.

The facts include the facts that we expect to see but also some new facts that the agent received from the environment. Note that these facts correspond with (some of) the moves that the agent made. For example, we have `on(b2,table)` which corresponds with the action the agent performed of moving `b2` to the `table`. This is true for all facts received from the environment. But for some actions we do not find such corresponding facts.

Exercise 3.7.1

There is no fact `on(b1,table)` corresponding to the first action of moving `b1` to the `table`. We also have no fact `on(b7,table)`. Why are these facts missing in the agent's database?



Specify an element to select (? = any character, * = any String):

```
on(
```

```
on(1,5)
on(2,0)
on(3,0)
on(4,3)
on(5,2)
on(6,4)
on(7,0)
on(b1,b5)
on(b2,table)
on(b4,b3)
on(b5,b2)
on(b6,b4)
on(Goal,X,Y) :- nth1(X,Goal,Y)
```

Figure 3.9: `on/2` facts in the Blocks World agent's database

3.8 Summary

A lot of the work of writing an agent program goes into writing code for representing and reasoning about the environment the agent is interacting with. It is important to build on or re-use the predicates that are received from the environment. For the Blocks World, we defined what blocks are and when blocks are clear by means of the `on/2` predicate used by the environment. Using the `on/2` predicate we also defined the useful concept of a tower. Because it is important to be able to reason about the current as well as the goal state, we also introduced analogous versions of these rules for defining these concepts for the goal state. We did so by adding the `Goal` parameter used for passing the goal state to a module to these definitions.

We have designed a simple strategy that is quite effective for solving Blocks World Problems. This strategy can be implemented by only two action rules in the decision module of the agent. Even with only two rules we have seen that the order of action rules in a module can make a difference. For our Blocks World agent, the order mattered because of performance. In later chapters we will see that order also matters for being successful. In particular, we will see how we can also exploit rule order when writing action rules.

The most important use of a module for an agent is for making decisions on what to do next. Modules can be used for other purposes too, though. In this chapter, we have seen how modules (either an ***init*** or an ***updates*** module) can be used for *preprocessing* the percepts that an agent receives. A new rule of the form **`forall`** `<query>` **`do`** `<action>` was introduced for updating many facts with a single rule.

By tracing agent program we are able to inspect the state (its database) of an agent. An agent queries and updates its state at runtime which is somewhat similar to how a Java method operates on the state of an object. Agent programming, however, works with databases and therefore can also be viewed as *programming with databases*. As we are using Prolog, the content of an agent's database state is specified as a Prolog program.

Finally, the properties of an environment make a difference for how we can write an agent program. Although it seems more natural to update an agent's state in each execution cycle with the percepts it receives, a Blocks World agent does not need to do so because of the specific properties of the Blocks World as we have seen in this chapter. It only needs to look once...

3.9 Exercises

```
:- dynamic on/2.
block(X) :- on(X,_).
clear(X) :- block(X), not( on(_, X) ).
clear(0).
tower([X]) :- on(X, 0).
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).
on(1,4)
on(2,0)
on(3,5)
on(4,2)
on(5,1)
```

Figure 3.10: An Example Database and Goal

3.9.1

Consider the database of Figure 3.10. Which of the following queries succeed? Provide all possible instantiations of variables, or otherwise a conjunct that fails.

1. `above(X,4), tower([4,X|T])`
2. `above(1,X)`
3. `clear(X), on(X,Y), not(Y=0)`
4. `above(X,Y), tower([X,Y|T])`

3.9.2

Partial Blocks World States 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.
Note that in this version of a Blocks World Problem the goal state may not refer to all blocks; in that case the agent program developed in this chapter does not work. The reason is that the query `tower(Goal, [X,Y|T)), tower([Y|T])` for expressing that block `X` is *misplaced* only works for blocks that are part of the goal state. Blocks that are not part of the goal state may, however, still be *in the way*. This means that the block prevents moving another block that is misplaced, and therefore needs to be moved itself. Provide an example of a Blocks World state and goal where this problem occurs.
The definition of the `above/2` predicate in Figure 3.10 can help resolve this issue. Work out a solution. Provide a 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!

3.9.3

A Blocks World Agent Without Tower Predicates In this exercise we investigate how we can write an agent program for solving Blocks World Problems that uses a somewhat simplified Prolog program. The key insight here is that we do not need to reason about towers if we can start building stacks from a configuration where all blocks are on the table. The idea is thus to first move all blocks to the table and then start building stacks.

a. The first step is to remove all the Prolog code that depends on the `tower/2` and `tower/3` predicates. After removing the Prolog code, the compile errors for our mas project will also indicate that we need to remove the action rules in the `stackBuilder` module.

b. The second step is to add an action rule for moving all blocks to the table. We can simply do so by replacing the `misplaced/2` predicate with the `on(X, _)` in the action rule that we used before for handling misplaced blocks. Run the agent in Debug Mode to check that by using this single action rule in the `stackBuilder` module the agent will move all blocks to the table.

c. Now that we have an agent that moves all blocks to the table, the main question that remains is how to build the stacks that we want without destroying them again. Let's get started by creating an action rule that will only put a block on top of another rule if that will make progress towards the goal. We needed the `tower` predicates before to ensure this, but assuming that we start from a configuration where all blocks are on the table, the `on/2` and `on/3` predicates will be sufficient. The first thing to check is that we want a block `X` to be on another block `Y` and that this is not already the case. Then we need to make sure that the block `Y` where we want to put another block on is already in position, i.e. check that `on(Goal, Y, W), on(Y, W)` succeeds (why do we use `W` as variable name?). Finally, as before, we need to check that the move is possible (check for the relevant `clear/1` preconditions), retrieve the current position of block `X`, and map the numbers back to the labels recognized by the environment. For this we can use exactly the same code we used before in the action rule for making constructive moves. Write this action rule and add it after the first rule in the `stackBuilder` module. Run the agent again. What happens?

d. The good news is that if the first rule in the `stackBuilder` module that we just added is *not applicable*, then we know that all blocks must be on the table (why?). The bad news, however, is that by simply adding our action rule for building stacks after the first rule, the agent will also immediately destroy the stack we just build again. To solve this problem, we will create a new module for building stacks and use this module as an action in a new action rule we will put in the `stackBuilder` module instead. Create a new module and call it `buildStacks`. As this module also needs to know which stacks it should build, add a parameter `Goal` in the module after the module name. Now move the rule we just wrote to the program section of the `buildStacks` module, and add a use clause for the Prolog file `blocksworld`. The last thing to do is to make sure that the module is being executed as long as we need to reach the goal state. That is, as long as there are still constructive moves to be made, the agent should stay inside the `buildStacks` module. We can make sure it does by adding the exit condition **`exit`**`=`***`noaction`*** to the `buildStacks` module.

e. The next step is to start using the new `buildStacks` module. We do this by adding the action rule **`if true then`** `buildStacks(Goal)` to the `stackBuilder` module after the first rule. We also need to add the use clause **`use`** `buildStacks` after the first use clause in this module (check what happens if we would not do this). Now run the agent again and check it solves the Blocks World Problem.

f. How does the performance of this new agent compare to the agent we wrote in this chapter? Rank order in terms of performance the agent of this chapter that has the action rule for making constructive moves at the top of the `stackBuilder` module, the one that has the rule for moving misplaced blocks to the table at the top of the module, and our new agent.

Chapter 4

Change is on the Horizon

In this chapter we will also revisit the order of action rules. Although we already have seen that the order in which action rules are put can make a difference performance-wise in Chapter 3, here we will look at how we can *exploit* rule order when writing a module for decision-making.

Another important theme of this chapter will be the actions that an agent can perform. Different from the actions that we have seen before, we will look at actions that take time to complete.

4.1 The Tower World Environment

In the Blocks World we could treat the `move` action as if its effects are achieved immediately. The `move` action is completed *instantaneously*. We can no longer treat actions as if they take no time and complete instantenously in the Tower World (see Figure 4.1) that we introduce here.

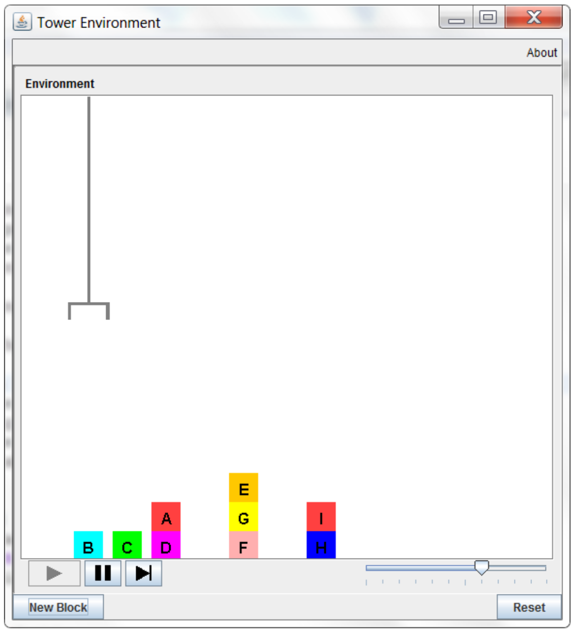


Figure 4.1: The Tower World Interface

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 a significant amount of time. The movement of the gripper is made visible and you can see the gripper moving towards a block to pick it up or move somewhere to put the block it is holding down. Both of these actions are **durative** in this environment, like most actions are in most environments. There are three actions that the gripper entity can perform in the Tower World. The action `pickup(X)`, with a single parameter for indicating which block `X` to pick up with the gripper, can be performed when `X` is clear, and the gripper is not already holding a block. The action `putdown(X,Y)`, where parameter `X` should be a block and `Y` can either be a block name or the `table`, will put down `X` if the gripper holds that block and `Y` clear. Finally, the special `nil` action which does not have any parameters will move the gripper back to the left-upper most corner in the window displaying the environment. The idea is that when an agent has nothing to do any more to achieve its goal it should move the gripper back into its initial starting position. This then will provide an indication that the agent (thinks it) is ready.

The gripper is not the only agent that can make changes. A user (you!) can also move blocks by dropping and dragging them using a mouse to interact with the environment interface. This means that an agent that controls the gripper does not have full control like it hardly ever has in most environments. As a consequence, the effects of actions are not guaranteed. In the Tower World actions may *fail* because another agent can interfere. Even though the effects of performing an action without interference are completely predictable and the Tower World environment is **deterministic**, the actions the gripper performs in the Tower World may not always succeed to put a block at a particular position. We also say that the Tower World environment is **dynamic** in the sense that things can change without the agent itself doing anything (in contrast with being static). Finally, like most environments the Tower World is a **multi-agent** environment. In such environments more than one agent can make changes and effect the environment. In the Tower World the user is this second agent that can also interact with the environment and move blocks. This introduces some new challenges that the Blocks World agent that we developed in Chapter 3 is not able to handle.

Apart from these key differences the Tower World is 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. The gripper, moreover, also has access to the full state and can "see" the complete blocks configuration. In other words, the Tower World is **fully observable**. There are also only a finite set of blocks in this environment and a limited number of actions available to make the gripper do something. Although the fact that actions take time has consequences (they now can fail), otherwise this fact is not relevant for the agent's decision-making. We can thus still classify the Tower World as a **discrete** environment. Finally, the aim is still to move blocks from a given block configuration to achieve another blocks configuration that represents the goal state.

The fact that actions take time gives another agent time to interfere. In the Tower World a user acting as a second agent 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 into the gripper. An agent connected to the gripper in the Tower World does not have full control. While performing an action of picking up a block, for example, it may no longer be feasible to do so because the block may no longer be clear. It may also be that a block the gripper is holding can no longer be put into position. If a block becomes obstructed in some way or is moved to another position, moving the gripper to a particular location thus may no longer make sense. If that happens, the agent should *reconsider* what it is doing. 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. The agent will need to observe the changes in the Tower World that can happen due to actions of the second agent. It thus will have to look at all times at what happens in its environment and cannot disregard percept information as could be done in the Blocks World (cf. Section 3.7).

4.2 Creating a Tower World mas

Let's get started again by creating a mas project for the Tower World. As before, we need to specify an environment connector, an agent definition, and a policy for launching an agent. We create a mas file called `TowerWorld.mas2g` (see Figure 4.2). The environment use clause has no with clause because the Tower World has no parameters for initializing the environment. The Tower World is different from the Blocks World in that it requires a user to add blocks to the environment using the environment's graphical interface. All blocks that are added are put on the table initially. In the agent definition, we use a module parameter to pass a goal state to the `towerBuilder` module that is used for decision-making as we have done before. Note that we use lists of numbers again to represent a configuration of blocks. The launch policy can also be copied from the Blocks World mas file we created in Chapter 3 except for the name of the agent which we changed to `towerBuilderAgt`.

```
use "tower-1.4.1.jar".

define towerBuilderAgt {
   use towerBuilder([2, 3, 0, 5, 6, 0]) for decisions.
   use mapTWnames for updates.

   % percept processing
   replace holding/1.
   replace on/2.
}

launchpolicy {
   when * launch towerBuilderAgt.
}
```

Figure 4.2: mas file for connecting an agent to the Tower World gripper

The more interesting part concerns the new percept that is received from the environment.[1] Using a **`replace`** handler, the agent subscribes to the channel for receiving `holding/1` predicates. The `holding/1` predicate provides the agent with the ability to monitor its gripper. It informs the agent whether it is holding a block or not at any time. Note that a Marbel agent is able to monitor progress of actions because it does not block on an action that it sends to the environment.[2] While an action is being performed by an entity such as the gripper in the Tower World, the agent therefore can monitor progress of the action it is trying to perform by means of the percepts it receives from the environment. Like the Blocks World, the Tower World is full observability: Every time percepts are received they provide *correct and complete* information about the state, i.e., the configuration of blocks. Because of this we can make use of the Closed World Assumption to infer whether the gripper is holding a block or not. In other words, if the query `holding(_)` does not succeed, we can conclude that the gripper is not holding a block.

4.3 Representing and Reasoning about the Tower World

Most of the groundwork for reasoning about the Tower World has already been done in Chapter 3 and we can re-use most of the Prolog code that we created for the Blocks World. We create a copy of the `blocksworld.pl` file into the Tower World mas project which we call

[1]The Tower World provides a third percept `block/1` which we do not use because we can infer when something is a block rather than the table.

[2]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.

```
% declaration of the holding/1 and on/2 predicates.
:- dynamic holding/1, on/2.

% block X sits on top of Y in the goal state Goal.
on(Goal, X, Y) :- nth1(X, Goal, Y).

% only blocks can be on top of something else;
% if the gripper is holding something, that also must be a block.
block(X) :- on(X, _) ; holding(X).

% the table is always clear.
clear(0).
% a block is clear if nothing sits on top of it, and it is not held by the gripper.
clear(X) :- block(X), not( on(_, X) ), not( holding(X) ).

% a tower is any stack of blocks that sits on the table.
tower([X]) :- on(X, 0).
tower([X, Y| T]) :- on(X, Y), tower([Y| T]).

% tower(Goal, T) holds if T is a tower in the goal state Goal.
tower(Goal, [X]) :- on(Goal, X, 0).
tower(Goal, [X,Y|T]) :- on(Goal, X, Y), tower(Goal, [Y|T]).

% moving block X on top of Y is constructive.
constructive(Goal, X, Y) :- tower(Goal, [X,Y|T]), tower([Y|T]).

% block X is misplaced.
misplaced(Goal, X) :- tower([X|T]), not( tower(Goal, [X|T]) ).

% helper predicates.
label_to_number(table, 0).
label_to_number(Label, Num) :-
    not( Label=table ), atom(Label), char_code(Label, Code), Num is Code-96.

number_to_label(0, table) :- !.
number_to_label(Num, Label) :- integer(Num), Code is Num+96, char_code(Label, Code).
```

Figure 4.3: towerKnowledge.pl

towerKnowledge.pl. We need to make a few changes because the gripper in the Tower World can also be holding a block. First, we add holding/1 to the list of declarations at the start of the file. Because a block is still a block when it is held by the gripper, we also add a clause to account for that situation in the rule for the block/1 predicate. The rule for the clear/1 predicate needs to be changed for a similar reason because a block that is held by the gripper is not clear (we cannot put a block on it). We add a clause to exclude this case in the rule.

Because the naming convention in the Tower World is different again from that of the Blocks World (it uses small case letters a, b, etc. to reference blocks), we have to change the code for mapping labels to numbers and vice versa. We do so by using the built-in char_code/2 predicate instead of the atom_concat/3 predicate. We need to take some care to handle the different Prolog data types that the char_code/2 predicate expects appropriately. As a first parameter, this predicate expects a term of type atom. As a second parameter, the predicate expects a term of type integer. We add some checks to our rules to make sure they are only applied when the right data types are fed into the predicates label_to_number/2 and number_to_label. Because we will not remove the percepts received from the environment, we will also need to handle the case where 0 needs to be mapped back to the table label; the cut ! operator from Prolog is used to prevent that the rules for number_to_label return any alternative bindings when Num is 0.

Otherwise, we simply re-use most of the code for, e.g., the tower/1 and tower/2 predicates.

4.3.1 Preprocessing Percepts Again

As before in Section 3.4, we need to handle the naming convention that the Tower World uses which is different from the numbers our agent is able to work with. Getting this right for the Tower World, however, is more involved than for the Blocks World because it takes time before changes are realized and the corresponding percepts are received from the environment. There are two different percepts to handle, moreover, and trying to devise an update strategy to keep track of changes in the environment instead of relying on the information received from the environment itself through percepts is difficult. We therefore choose to implement a different approach that does not remove the percepts received from the environment but just adds their corresponding counterparts that use numbers to refer to blocks instead of alphabetic characters. Adding the facts with numbers instead of labels is relatively easy given our prior work on the mapping predicate `label_to_number`. We also need to remove these facts again, however, when the environment changes. To do so we exploit the fact that the Tower World is fully observable. Because the agent always "sees" all there is to see in the Tower World, we can remove a fact with numbers if we do not find the corresponding fact with labels any more in the database. For example, initially we have `on(b,table)`. The corresponding fact `on(2,0)` will then be added to the database. However, suppose now that block `b` is moved onto `c`. The **`replace`** channel handler for `on/2` will then update the database and remove the fact `on(b,table)` and insert the fact `on(b,c)`. Because we no longer can find the `on(b,table)` in the database any more, we can then conclude that we need to remove the corresponding fact `on(2,0)` from the database too. Of course, we should also add `on(2,3)`. A similar logic can be applied to the `holding/1` predicate. To summarize this discussion, we provide the code for this approach of preprocessing percepts in the `mapTWnames` module:

```
use towerKnowledge.

module mapTWnames {
   forall on(L1, L2), label_to_number(L1, N1), label_to_number(L2, N2)
      do insert( on(N1, N2) ).
   forall on(N1, N2), number_to_label(N1, L1), number_to_label(N2, L2), not(on(L1,L2))
      do delete( on(N1, N2) ).

   forall holding(L), label_to_number(L, N)
      do insert( holding(N) ).
   forall holding(N), number_to_label(N, L), not( holding(L) )
      do delete( holding(N) ).
}
```

Figure 4.4: the `mapTWnames` module

4.4 Actions in the MARBEL Language

Before we move on to design a strategy for our Tower World Agent, we take a moment to look in more detail at the actions that an agent performs. An **action** is defined by its **name** `id`, and the number of **parameters** of the form $(t_1, \ldots, t_n)$ it has. 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. Only actions that are **fully instantiated** should be performed by an agent. That is, all free variables in parameters of an action must have been fully instantiated with ground terms. This is a general requirement that applies to all actions. For a `move(X, Y)` action this means, for example, that the variables `X` and `Y` need to be instantiated with ground terms that refer either to a block or to the table before it can be performed. Actions must be fully instantiated to ensure that it is clear which action exactly should be performed. What, for example, does it mean to perform the action `move(X,table)` when the variable `X` has not been instantiated? There seems to be no reasonable option to perform actions that are not fully instantiated.

Actions made available by an environment are called **environment actions**. If an agent is connected to an environment, such 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 that 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.

An agent performs environment 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 being able to perform it in the environment hold. Preconditions specify whether it is possible to perform an action. A precondition can be viewed as a query φ that is evaluated against an agent's database to check whether it succeeds or not. When an agent performs an action it is useful to check first whether the preconditions for being able to perform the action hold. Actions are **enabled** when this is the case.

4.4.1 Instantaneous and Durative Actions

Performing an environment action such as the `move` action means that the action is sent to the environment that the agent is connected to. The environment then 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 **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. In the case of an instantaneous action, we can then conclude that its effects are realized immediately when the action is performed. Actions that take time are called durative actions. The effects of durative actions are established only after some time has passed. Moreover, these effects are not guaranteed because of other events that may take place while the action is being performed. All actions available in the Tower World are durative.

As a rule of thumb, you should assume that executing an environment action takes time and an environment action 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 very different 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. 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, or a block is removed from the gripper. In all of these cases, the effects of environment actions can only be perceived by an agent if it receives the correponding information via percepts from the environment.

Another issue that we need to consider when programming an agent that performs durative actions 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 `move` action in StarCraft. 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 StarCraft, 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.

4.4.2 Built-in Actions Available in the MARBEL Language

We now briefly discuss the built-in actions that are available in the Marbel programming language. Some of these actions we already have been using while others are new. They include actions for changing the database of an agent, for printing messages to the console, logging information to a file, for making an agent go to sleep, for setting and cancelling timers, and for communicating information with other agents. We discuss these actions here except for the `send` action that is used for communicating messages to other agents, which is discussed in Chapter **??**.

Modifying an Agent's Database There are two built-in actions available for modifying an agent's database. First, the action:

```
insert(<update>)
```

can be used to insert `<update>` into the agent's database. `<update>` should be a conjunction of Prolog literals. The **`insert`** action can always be performed. The **`insert`** action *adds* all positive literals that occur in `<update>` to the database of the agent and *removes* all facts that occur in negative literals in `<update>` from the database. For example, **`insert`**`(`**`not`**`(on(1,2)), on(1,0))` removes `on(1,2)` from the database and adds `on(1,0)`. As a result of performing an **`insert`**`(<update>)` action the query `<update>` will succeed.

Second, the action:

```
delete(<update>)
```

can be used to delete `<update>` from the agent's database. The **`delete`** action can be viewed as the inverse of the **`insert`** action: Instead of adding literals that occur positively in `<update>` it removes such literals, and instead of removing negative literals it adds those 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 a program easier to read. It is considered best practice to *use **`insert`** only for adding facts and to use **`delete`** only for removing facts.* For example, instead of writing **`insert`**`(`**`not`**`(on(1,2)), on(1,0))`, it is better to use a combination of the two actions **`insert`**`(on(1,0))` and **`delete`**`(on(1,2))`. Multiple actions can be combined by the + operator. Using this operator we can write:

```
insert(on(1,0)) + delete(on(1,2))
```

which would have exactly the same effect as the single action **insert**(**not**(on(1,2)), on(1,0)). 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 from left to right.

One thing to note about the **delete**(<update>) action is that it removes *all* positive literals that occur in <update>. The **delete** action thus does not support *minimal change* in the sense that no more information would be removed than is strictly necessary to make sure that the query <update> does not succeed 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 make the query p, q fail.

Printing a Message We have used the **print** action already in Chapter 1. The action:

```
print(<term>)
```

prints the Prolog term <term> to the standard output console. Here, <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. Because logging is platform dependent, more information on this action can be found in the User Manual [31].

Sleeping The action:

```
sleep(<term>)
```

sleeps the agent for <term> milliseconds, i.e., it freezes the agent for that amount of time. After the sleep, the agent will continue executing as if any other action just had been performed.

Timers The action:

```
starttimer(<name>,<interval>,<duration>)
```

can be used to start a timer. If a timer with the given <name> already exists, it will be replaced. A timer in Marbel will generate a percept on each <interval> milliseconds, starting from the call of the action until (and including) <duration> milliseconds have elapsed. Such percepts will be of the form:

```
timer(<name>,<elapsed>)
```

where *elapsed* is the number of milliseconds of the current interval; *elapsed* will thus always be a multitude of *interval* (and it therefore does not make much sense to not make *duration* a multitude of *interval* either). Note that if the cycles of an agent take longer than *interval* milliseconds, it is possible to receive multiple timer/2 percepts at once. Moreover, if a timer did not reach an interval, a timer percept will not be present for it. For example, calling **starttimer**(example,1000,10000) will generate a percept every second, thus creating a sequence of timer(example,1000), timer(example,2000), ..., timer(example,10000) percepts.

The action:

```
canceltimer(<name>)
```

can be used to cancel a timer (i.e., make it stop generating percepts) before its duration has ended. If a timer does not exist or has already ended, a warning will be printed.

Like all other actions, the built-in actions that we discussed here must have been fully instantiated 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. Finally, all built-in actions that have been discussed here can for almost all purposes be treated as instantaneous actions as the time they take to execute is typically negligible compared to the speed things change in an agent's environment.

4.5 A Strategy for the Tower World

To specify an effective strategy for our Tower World agent, we will first pretend there is no other agent that can interfere with what we are trying to achieve. To get started we then first need to think about when the agent should pick up a block. In the Tower World, a good reason for picking up a block (and thus to hold it) is that a constructive move can be made with the block. As discussed in the previous section we also need to check whether the preconditions for picking up a block are satisfied too: The block should be clear and the gripper should not be holding a block already. Furthermore, we should avoid making redundant attempts to move a block. For this we need to check that the block we want to move the block we intend to pick up is clear, and that the block we want to pick up is not already where it should be. And, last but not least, we should map the number for the block back to the label expected by the environment. We add the following rule to the `towerBuilder` module for picking up a block:

```
if constructive(Goal, X, Y), clear(X), clear(Y), not( holding(_) ), not( on(X, Y) ),
   number_to_label(X, L)
then pickup(L).
```

As you can check by running the agent (don't forget to add a sufficient number of blocks using the New Block button on the interface!), using this rule the agent moves the gripper to pick up block `b` in the environment. The next thing we should see to is to then put the block the gripper is holding down again on the block we want it to be. The following rule will achieve this:

```
if constructive(Goal, X, Y), clear(Y), holding(X),
     number_to_label(X, Lx), number_to_label(Y, Ly)
then putdown(Lx, Ly).
```

Note that in this case we do not need to check that `X` is already on `Y` because we already check that the gripper is holding `X`. Because putting a block down should be prioritized over picking up a block, we put our new rule before the previous rule in the `towerBuilder` module.

As you can check again, the agent will now build the towers that it should build. That means that without interference the agent is able to achieve its goal. However, if we now start interfering by moving some blocks to positions where they are misplaced, the agent is at a loss and the gripper stops moving. Clearly, the agent does not know how to handle misplaced blocks and is not able to respond to actions of a user that interfere with the agent's goal. We could get away with this because thus far we have exploited the fact that all blocks are initially put on the table. In that case the agent can reach the goal state by only making constructive moves (compare exercise 3.9 in Chapter 3).

In the remainder of this section, we complete the design of an agent program that is able to respond appropriately to actions of another agent in the multi-agent Tower World. The first issue that we will fix is to introduce another rule that instead of picking up a block when a constructive move can be made picks up a block when it is misplaced. We can almost verbatim copy paste the rule for the constructive move case above and only need to replace the `constructive/3` predicate with the `misplaced/2` predicate and replace the remaining `Y` variable which no longer is instantiated (we need to prevent floundering!) with a `0` representing the table:

```
if misplaced(Goal, X), clear(X), not( holding(_) ), not( on(X, 0) ),
   number_to_label(X, L)
then pickup(L).
```

Because we should prioritize constructive moves over moving misplaced blocks (as we have seen in the Blocks World, that is more efficient), we put our new rule last in the `towerBuilder` module.

As you can check again, as long as we only move a block in a position where it is misplaced when the gripper is empty (not holding a block), the agent is able to continue to make progress up and until it needs to put down a misplaced block again with which it can not make any constructive move. We need to add another rule to the module for handling this case. The most simple rule for doing so is:

```
if holding(X), number_to_label(X, L) then putdown(L, table).
```

This rule is surprisingly simple compared to the other rules we have introduced so far. This is in part due to the fact that there are no preconditions for putting a block on the table (as by definition the table is always clear) and the fact that a block that is held cannot already be in a position we want it to be. Moreover, we already have another rule for putting down a block to make a constructive move, so we do not have to check for that either.

Exercise 4.5.1

Before you read any further, think about where you would put the rule introduced above in the `towerBuilder` module. Why would you put it there?

We will now put everything together and try to simplify. But first, we need to decide where to put the last rule we defined. First of all, if the agent is holding a block, it has to put it down somewhere (the gripper can hold at most one block). Now, if the agent holds a block, it should, first, try to put it in position and make a constructive move, but, if that is not possible, put it somewhere on the table.

Now let's put everything together to see if we can simplify our code:

```
use towerKnowledge.

module towerBuilder(Goal) {
   % put down a block on another block if a constructive move can be made with it.
   if constructive(Goal, X, Y), clear(Y), holding(X),
        number_to_label(X, Lx), number_to_label(Y, Ly)
   then putdown(Lx, Ly).

   % otherwise, when holding a block, put it down on the table.
   if holding(X), number_to_label(X, L) then putdown(L, table).

   % NONE OF THE RULES ABOVE IS APPLICABLE AT THIS POINT:
   % THIS MEANS THE AGENT IS NOT HOLDING A BLOCK!
   % if a constructive move can be made with a block pick it up.
   if constructive(Goal, X, Y), clear(X), clear(Y), not( holding(_) ), not( on(X, Y) ),
      number_to_label(X, L)
   then pickup(L).

   % if a block is misplaced pick it up.
   if misplaced(Goal, X), clear(X), not( holding(_) ), not( on(X, 0) ),
      number_to_label(X, L)
   then pickup(L).
}
```

Thus far we have reasoned about which things should get priority to decide in which order we should put the action rules that we wrote. But we can further exploit rule order to simplify our code. First of all, we can derive information from the fact that the first applicable rule is applied in a module for decision-making (our `towerBuilder` module is such a module). This means that we would only consider applying the third rule when the first two rules are not applicable. But clearly, by inspecting the second rule, this means that the agent can not be holding a block if that rule is not applicable. The first simplification that we can make, therefore, is that we do not need to check that the agent is not holding a block in the third and fourth rule. We thus can remove this check from the queries of these rules.

The second simplification of our code that we can make uses a new feature of the programming language: **nested rules**. Instead of an action a rule can also have a rule section as head. Such rules are of the form

```
if <query> then {
   ...
}
```

where the dots ... can be replaced by one or more action rules. By pulling the common parts of the two first rules out of these rules and renaming the single variable used in the second rule, we arrive at the following code for the towerBuilder module:

```
use towerKnowledge.

module towerBuilder(Goal) {
   % if the agent is holding a block, put it down somewhere.
   if holding(X), number_to_label(X, Lx) then {
       % put down block X on block Y if a constructive move can be made with it.
       if constructive(Goal, X, Y), clear(Y), number_to_label(Y, Ly)
            then putdown(Lx, Ly).

       % otherwise, put it down on the table.
       if true then putdown(Lx, table).
   }

   % if a constructive move can be made with a block pick it up.
   if constructive(Goal, X,Y), clear(X), clear(Y), not(on(X,Y)), number_to_label(X, L)
       then pickup(L).

   % if a block is misplaced pick it up.
   if misplaced(Goal, X), clear(X), not(on(X, 0)), number_to_label(X, L)
       then pickup(L).

   % move the gripper back to its home position when done.
   if true then nil.
}
```

Note that we also added a finishing touch by adding a new rule for moving the gripper back to its initial position using the nil action when nothing else can be done. It now is up to you to test the agent and verify that it is operating correctly, with which we conclude the design of our agent for the multi-agent Tower World.

Chapter 5

Getting Things Done (Together)

Thus far we have looked at developing single agent programs, even though there might be other (human) agents interfering with the goals of the agent. In this Chapter we will look at how multiple agents can communicate and coordinate to achieve their joint goal. We will also look in more detail at channels and how they can be used for processing percepts from the environment and messages that agents exchange between themselves. Another theme discussed in this chapter is how to call modules from other modules and how variants of the linear rule evaluation order can be used to explore the grid in the Vacuum World that is used as running example in this chapter.

5.1 The Vacuum World

The Vacuum World is a grid world consisting of cells that can be dusty and, if so, should be cleaned by one or more VacBots that are present in the world (see Figure 5.1).

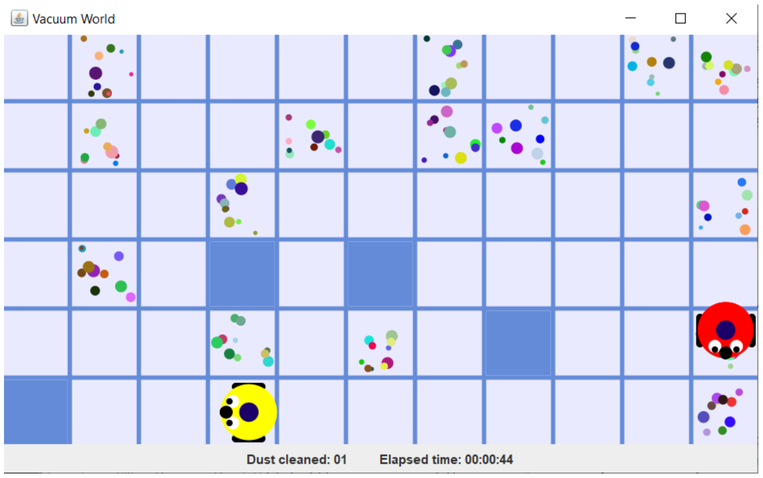
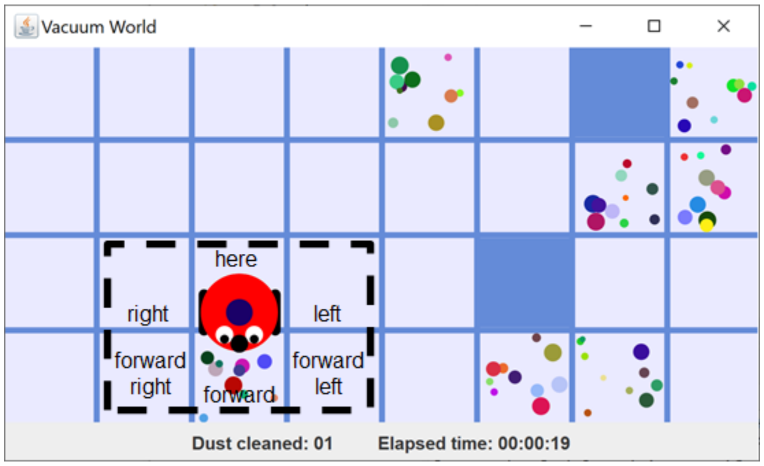


Figure 5.1: The Vacuum World

Initially, a VacBot is randomly positioned somewhere in the grid facing in one of the four directions north, east, south, or west. VacBots cannot occupy the same cell. VacBots move in a continuous fashion from cell to cell and can perform a `move(Dir)` action where `Dir` is indicates whether the move is to a cell to the `north`, `east`, `south`, or `west`, as long as such a movement is not obstructed by obstacles or another VacBot. If they are facing in a different direction then the direction they are instructed to move to, a VacBot will first turn to change the direction it is facing in and then move a forward move in the right direction. Turning and moving takes time. At the boundaries of the grid, obstacles prevent VacBots from moving of the grid. In case a VacBot bumps into an obstacle, the move it tries to make fails and the VacBot remains in the cell it already is. Besides moving, a VacBot can always perform a `clean` action in the cell it is in. If there is dust, a cleaning action will be performed until all dust has been removed. If there is no dust, a clean action does nothing and finishes immediately. There are two performance measures displayed at the bottom of the Vacuum World simulator window indicating how many dusty cells have been cleaned and how much time has been used thus far. The field of vision of a VacBot is limited and depends on the direction it is facing. A VacBot can see the contents of the cell it is in, and can look to the left, right, forward, and see the contents of the cells to the left and right of the the cell directly in front of it. In total, a VacBot thus "sees" the contents of six cells (see Figure 5.2).



Dust cleaned: 01 Elapsed time: 00:00:19

Figure 5.2: Field of Vision in the Vacuum World

The environment can be configured in different ways. The number of VacBots that are present, the size of the grid, and whether dust is (re)generated or not can be varied. In case the generation of dust is activated, dust and dirt may (re)appear in the vacuum world in cells which the agent has cleaned before. In that case the VacBots do not have full control over what happens in the environment.

The Vacuum World is **partially observable** because of the limited field of vision of a Vacbot. VacBots thus will have to explore the grid to locate the cells with dust. In this chapter, we will disable (re)generation of dust to make sure that VacBots are able to fully clean the grid, which will be the main goal in life of our agents. Because in this case the environment does not change if an agent does not do anything, this version of the Vacuum World is **static**. The effects of actions that VacBots are predictable and therefore the Vacuum World is **deterministic**. Even though the amount of dust in a cell can vary and it takes time to clean dust, we can treat the Vacuum World as a **discrete** environment because what really counts for the decision-making of an agent are the finite number of cells in the grid that generate at most a finite number of percepts and action possibilities. Initially, we will focus on developing a **single agent** controlling a single VacBot that is able to clean the grid. In the second part of this chapter, we discuss a **multi-agent** version of the Vacuum World where multiple VacBots need to coordinate their effort to clean the grid to do this more efficiently than a single VacBot.

Agent's Goal in Life The main purpose of VacBot's in the Vacuum World is to clean dust (as efficiently as possible). In this chapter we will not discuss how to optimize performance of such bots but focus on the job of cleaning all dust and dirt present in the Vacuum World grid. We specify the agent's main goal in life by the following requirements. First and foremost, the agent(s) should (1) make sure that *all dust has been cleaned*. Thereafter, we require agents (2) to return to their original starting position when done, (3) and self-terminate when back at that position. This will make it clear to us that the agent(s) has (have) finished its (their) job.

5.2 Creating a mas for the Vacuum World

We first create a mas file `VacuumWorld.mas2g`. The environment use clause has a with with several parameters for initializing the Vacuum World. The `level` parameter can be used to indicate how many VacBots will be present in the Vacuum World. The Vacuum World grid size is increased if more VacBots are present in the World. When this parameter is used, also only a limited number of obstacles are randomly placed inside the grid (2x the number of VacBots present).[1] The `generation` parameter is set to `"no"` to prevent dust from being generated by the environment. The speed parameter is useful to speed up the movement of the VacBots to be able to quickly get a sense of how the VacBots perform in the Vacuum World.

```
use "vacuumworld-1.3.2.jar"
   with level = "1", generation = "no", speed = "1000".

define vacuumBotAgt {
   use vacuumBot for decisions.
   use vacuumInit for init.
   use vacuumUpdate for updates.

   % channels
   update location/2.
   update direction/1.
   replace square/2.
   replace task/1.
}

launchpolicy {
   when * launch vacuumBotAgt.
}
```

Figure 5.3: mas file for connecting an agent to a VacBot in the Vacuum World

The agent definition defines an agent called `vacuumBotAgt` which is connected to a VacBot and launched by the launch rule in the launch policy section. The agent uses three modules for initializing and updating the agent's database (state), and for decision-making. The module for updating the database will be used to keep track of the progress that the agent(s) make.

5.2.1 Channels and Types of Percepts

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). By connecting to an entity an agent obtains access to the capabilities of that entity. An agent can see what the entity can see in its environment and control the actions that an entity can perform. An agent that is connected to an entity receives percepts to "see" what the entity sees and can request the entity to perform actions.

The connectors that we use for connecting agents to environments distinguishes three types of percepts [3, 4]: percepts that are **send once** when the agent first connects to the environment, percepts that are **send always**, and percepts that are **send on change** when a feature of the environment changes. Each of these percept types should be handled differently and the Marbel language provides different channel handlers for processing these different percept types.

[1]Even with only two obstacles inside the grid a corner cell might not be reachable for a VacBot and it may be impossible to clean the full grid; we will assume that this never happens, or, in other words, disregard this very special case.

Send Once For percepts that are received only once, a programmer needs to decide whether these should be added to the database of the agent when it is connected to the environment or not. 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 (BW4T) and StarCraft environments. For each percept of this type that the agent should store in its database, a channel should be added to the agent definition. The **`add`** handler should be used to process send once percepts. This handler simply adds the percepts received on the channel to the agent's database. This handler will also be useful for processing messages, as we will see below. An example that we will use below for messages received about obstacles is:

```
add obstacle/2.
```

Send Always We have already handled percepts that are sent always in previous chapters. A percept that is received always when the environment feature is present requires an agent to both add information to its database when it is available and remove it again when it is no longer available. New information that becomes available should be added whereas old information should be removed. That is what the **`replace`** handler does and send always percepts therefore should be processed using this handler. The `square/2` percept in the Vacuum World provides an example. The percept `square(RelDir, Content)` provides the agent with information about the contents of cells in its field of vision. The contents of a cell can be either `empty`, `dust`, `obstacle`, and `vac` (for VacBot). The `RelDir` parameter of this percept specifies a relative direction of view. The agent will always receive six percepts with the values `left`, `here`, `right`, `forwardLeft`, `forward`, `forwardRight` for the parameter `RelDir`. The **`replace`** handler removes the previously received six percepts from the database and replaces these with the six new percepts it receives from the entity. We have included two channels for send always percepts in the agent definition:

```
replace square/2.
replace task/1.
```

The `task(Act)` percept indicates what the entity is doing, where `Act` can be `turn` for when a VacBot is turning, `move` for when it is moving, `clean` for when it is cleaning, or `none` if it is doing nothing. A somewhat peculiar feature of the Vacuum World is that percepts are only updated when a VacBot is doing nothing. This will be important for processing percepts as well as for the agent's decision-making (we should make sure that the VacBot controlled is only asked to do something new when it is not doing anything at that moment, to also make sure new updated percepts are received).

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. In the BW4T, for example, a percept `state(traveling)` is received when a robot starts moving and a percept `state(arrived)` is received when the robot arrives at its destination. In between when the robot is moving, no `state/1` percept is received. This type of percept should only be removed from the agent's database when an update for the percept is received. This is achieved by means of the **`update`** handler. The `location/1` and the `direction/1` channels use the **`update`** handler for updating these percepts:

```
update location/1.
update direction/1.
```

The `location(X, Y)` percept indicates that the VacBot is at coordinates `(X, Y)` in the grid. The left-upper corner has coordinates `(0, 0)`. The cell to the right of it has coordinates `(1,0)` and the cell one below the cell at `(0,0)` has coordinates`(0,1)`. The `direction(Dir)` percept indicates the (absolute) direction which the VacBot is facing; the `Dir` parameter can be either `north`, `east`, `south`, or `west`.

5.3 Representing and Reasoning about the Vacuum World

We create a Prolog file `vacuumKnowledge.pl` for reasoning about the Vacuum World. At the start of this file amongst other predicates the four channel percepts that we used in our agent definition are declared (channel percepts are used but never defined because they are fixed by the environment). Agents need to keep track of the contents of cells in the grid. If for all cells inside the grid an agent knows that the cell is either clean (`empty`) or is an obstacle, then it can conclude that the job is done. This is why two predicates `clean/2` and `obstacle/2` are declared in the Prolog file. Because initially the agent also does not know the exact boundaries of the grid, an important part of the exploration task of agents will be to determine the grid boundaries. This is why a predicate `gridBoundary/2` is declared. We use the fact `gridBoundary(Dir, XorY)` for representing the `X` coordinate boundary for the directions east and west and the `Y` coordinate boundary for the directions north and south of the grid. Only if agents know these boundaries can they conclude the grid has been fully cleaned. Finally, because agents are randomly placed on the grid initially, to be able to return to their original location we declare a predicate `initialLocation/2` so agents can store the location where they started in the grid.

```
:- dynamic clean/2, direction/1, gridBoundary/2, initialLocation/2, location/2,
   obstacle/2, square/2, task/1.

% bot can move north, east, south, or west if that direction is not blocked
canMove(Dir) :-
   (Dir = north ; Dir = east; Dir = south; Dir = west),
   location(X, Y, Dir), not( obstacle(X, Y) ).

done :-
   % agent is back at initial location.
   location(X, Y), initialLocation(X, Y),
   % compute boundaries of grid.
   gridBoundary(north, N), gridBoundary(east, E),
   gridBoundary(south, S), gridBoundary(west, W),
   % check whether all cells within boundaries are clean or obstacles.
   N1 is N+1, E1 is E-1, S1 is S-1, W1 is W+1,
   forall( (between(W1,E1,X), between(N1,S1,Y)), ( clean(X,Y) ; obstacle(X,Y) ) ).

% the coordinates of the cell to the north of the current location are X, Ynew, etc.
location(X, Ynew, north) :- location(X, Y), Ynew is Y-1.
location(Xnew, Ynew, northeast) :- location(X, Y), Xnew is X+1, Ynew is Y-1.
location(Xnew, Y, east) :- location(X, Y), Xnew is X+1.
location(Xnew, Ynew, southheast) :- location(X, Y), Xnew is X+1, Ynew is Y+1.
location(X, Ynew, south) :- location(X, Y), Ynew is Y+1.
location(Xnew, Ynew, southhwest) :- location(X, Y), Xnew is X-1, Ynew is Y+1.
location(Xnew, Y, west) :- location(X, Y), Xnew is X-1.
location(Xnew, Ynew, northwest) :- location(X, Y), Xnew is X-1, Ynew is Y-1.
```

The two main things that an agent needs to be able to reason about are (1) whether a move can be made and is not obstructed by an obstacle (or other VacBot), and (2) whether it is done, i.e. whether the grid is clean and the agent is back at its original location. The rule for `canMove(Dir)` first checks whether the direction `Dir` is a viable direction to move in (a VacBot cannot move diagonally), then computes the coordinates of the cell the move towards `Dir` would move the VacBot to, and finally checks whether that cell is not an obstacle. To compute the coordinates of the cell in a particular direction relative to the current location of a VacBot, the agent uses the rules for `location(X, Y, Dir)`. The left-upper corner has coordinates `(0, 0)`. Moving to the east increases the `X` coordinate, and moving to the south increases the `Y` coordinate. To compute the coordinates of the cell directly to the north of the current location thus requires subtracting 1 from the `Y` coordinate. Similar rules are defined for all directions. We also included rules for directions on a diagonal (e.g., `northeast`) which are useful when we update the agent's database with `square(RelDir, Content)` facts which also include relative directions such as `forwardLeft`.

To be able to conclude that the agent is `done`, the agent needs to be able to reason about the grid boundaries. The predicate `gridBoundary/2` is introduced for this purpose. The boundaries to the north (`Y` coordinate -1) and west are given (`X` coordinate -1). This is not true for the eastern and southern boundaries. We introduce somewhat ad hoc rules to compute where these boundaries are located by counting the number of obstacles we find in a row on the same `X` (eastern) and `Y` (southern) coordinate. We look for 5 obstacles in a row which will be sufficient for the levels 1 and 2 of the Vacuum World.

```
% northern boundary is given.
gridBoundary(north, -1).
% if there are 5 obstacles in a row located on the same X>0 coordinate,
% then we conclude we have an eastern boundary.
gridBoundary(east, X) :-
        obstacle(X, Y),
        Yplus1 is Y+1, obstacle(X, Yplus1),
        Yplus2 is Y+2, obstacle(X, Yplus2),
        Yplus3 is Y+3, obstacle(X, Yplus3),
        Yplus4 is Y+4, obstacle(X, Yplus4),
        X>0.
% if there are 5 obstacles in a row located on the same Y>0 coordinate,
% then we conclude we have a southern boundary.
gridBoundary(south, Y) :-
        obstacle(X, Y),
        Xplus1 is X+1, obstacle(Xplus1, Y),
        Xplus2 is X+2, obstacle(Xplus2, Y),
        Xplus3 is X+3, obstacle(Xplus3, Y),
        Xplus4 is X+4, obstacle(Xplus4, Y),
        Y>0.
% western boundary is given.
gridBoundary(west, -1).
```

5.3.1 Initializing and Updating the Agent's Database

Part of what is required by the agent's goal in life is to return to its initial starting position when the grid has been cleaned completely. To be able to meet this requirement and return back to their starting position, agents need to store the initial position of the robot they are connected to. Because robots initially are randomly placed on the grid, an agent needs to receive this starting position from the environment via the `location/2` channel. For storing this initial location of a robot, we use the special predicate `initialLocation`. Storing this location only needs to happen once, and must be done immediately when the agent is launched. For this purpose, a module for initializing the agent's database can be used:

```
use vacuumKnowledge.

module vacuumInit {
   if location(X, Y) then insert( initialLocation(X, Y) ).
}
```

To be able to conclude that the grid has been cleaned the agent needs to keep track of which cells are clean and which cells are obstacles. We introduced the `clean/2` and `obstacle/2` predicates to store this kind of information. Whether a cell is clean or an obstacle can be inferred from the `square/2` percepts that agents receive from the environment by combining it with the location of the VacBot. The idea is to first compute the absolute direction in the grid from the relative direction obtained from the `square(RelDir, Content)` fact, and then to compute the coordinates `(X, Y)` of the cell from this absolute direction. Using these coordinates, if the `Content` parameter is instantiated with `empty` we insert the fact `clean(X, Y)` in the agent's database, and if `Content` is instantiated with `obstacle` we insert the fact `obstacle(X, Y)`. The `square(here, Content)` fact does not require us to compute the coordinates of the cell which we can simply retrieve from the current `location(X, Y)` of the VacBot. Because we want to update the database with all six `square/2` facts that the agent receives, we use **`forall`**... **`do`**... rules to process all of these facts. As these updates need to happen each time new information is received from the environment, we use a module for updating the database and in this module check whether new information will have been updated by the `task(none)` fact in line with what we discussed above:

```
use vacuumKnowledge.

% Update database using square percepts.
module vacuumUpdate {
   if task(none) then {
      forall square(here, empty), location(X, Y)
         do insert( clean(X, Y) ).

      forall square(RelDir,empty), absDirection(RelDir,AbsDir), location(X,Y,AbsDir)
         do insert( clean(X, Y) ).

      forall square(RelDir,obstacle), absDirection(RelDir,AbsDir), location(X,Y,AbsDir)
         do insert( obstacle(X, Y) ).
   }
}
```

The final missing piece of the puzzle is how to compute an absolute direction (`north`, etc.) from a relative direction (`left`, etc.) from the current absolute direction `direction(Dir)` the VacBot is facing towards. We use a very simple approach by explicitly coding all the information. We only provide the rules for the relative direction `left` here. Rules for other relative directions are similar.

```
% when VacBot is facing north, its left is to the west, etc.
absDirection(left, west) :- direction(north).
absDirection(left, north) :- direction(east).
absDirection(left, east) :- direction(south).
absDirection(left, south) :- direction(west).
```

5.4 A Single Agent Strategy for the Vacuum World

We implement a basic strategy for our single agent. Most important, and therefore our first rule, will be to terminate the agent by the **`exit-module`**when the agent is `done`. The second highest priority, and therefore our second rule, is to perform the `clean` action when a VacBot stands on dust. Third, the agent will greedily move towards dust when possible. If we can move to a cell with dust, this also means that we have not been there yet because in that case the agent would have applied the second rule in that cell first. Moving towards dust thus also moves a VacBot to a new square. Finally, if the agent can do nothing is, the idea is to explore the grid by making a random move. The agent will, moreover, only select a new action when the VacBot it controls is doing nothing (`task(none)` succeeds).

```
use vacuumKnowledge.

order=linearrandom.

module vacuumBot {
   % quite when we're done.
   if done then exit-module.

   if task(none) then {
      % clean when we're standing on dust.
      if square(here, dust) then clean.

      % if we can move towards dust, do it (greedy).
      if square(RelDir, dust), absDirection(RelDir, AbsDir), canMove(AbsDir)
         then move(AbsDir).

      % otherwise, make a random move.
      if canMove(Dir) then move(Dir).
   }
}
```

In a default decision-making module, an action rule would be applied with the first binding obtained by evaluating the query of that rule. This would result in the agent to always make the same move when applying the fourth rule for exploring the grid. We can change this by selecting a slightly different mechanism for evaluating rules by the **`order`** option of a rule. For the `vacuumBot` module, we set the **`order`** option to **`linearrandom`**. This option still evaluates the rules in linear order as they appear in the module, but randomly selects one of the bindings obtained by evaluating the query to instantiate the action with. In particular, this means that the last rule performs a random move action out of the possible moves at a location.

5.4.1 Rule Evaluation Order

An action rule **`if`** `<query>` **`then`** `<action>` is evaluated by first evaluating the query `<query>` and computing all bindings for that query. The instantiations of the `<action>` with those bindings are called the *action options*. A rule may generate multiple action options. In the example of Figure 5.2, the query in the action rule **`if`** `canMove(Dir)` **`then`** `move(Dir)` computes four different bindings and four action options `move(north)`, `move(east)`, `move(south)`, and `move(west)`. 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, action rules that appear in a module for decision-making (**`for`** ***`decisions`***) 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 the example of Figure 5.2, the agent can move towards a cell with dust and the rule **`if`** `square(RelDir, dust), absDirection(RelDir, AbsDir), canMove(AbsDir)` **`then`** `move(AbsDir)` is applicable with only one instantiation: `move(south)`. This is the default **linear order** style of rule evaluation.

As we have seen, the rule evaluation order of modules for initialization (**`for`** ***`init`***) and for updating (**`for`** ***`updates`***) is different. Instead of applying only the first applicable rule, in these types of modules all rules are evaluated in order and all of the rules that are applicable are applied. 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. This option can be specified after the use clauses in a module. We can, for example, add **`order=random`** in our `vacuumBot` module 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 one of the actions in the set `move(south)`, `move(north)`, `move(east)`, `move(south)`, or `move(west)`. Note that the action `move(south)` appears twice as the two last rules in the module both generate this action. As a result, the agent will non-deterministically execute its program and may execute differently each time it is run.

The options that are available for the **`order`** option are the **`linear`**, **`linearall`**, **`linearrandom`**, **`random`**, **`linearallrandom`**, and **`randomall`** options. We have already discussed the first four of these options. The difference between the **`linearallrandom`**order and the **`linearrandom`**order is the same as that of the **`linearall`**and the **`linear`**order: the first evaluates all rules in a module and applies all applicable rules in a linear order while the second will only apply the first one that is applicable. Both rule evaluation orders will randomly select one of the applicable instantiations of a rule. 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.

5.5 Summary

In this chapter, we discussed a single agent for the Vacuum World. We introduced and discussed various new concepts, including how channel handlers relate to percept types and how the rule evaluation order of a module can be used to change the way rules are evaluated and applied when executing a module. We illustrated how the **`linearrandom`** option can be used to make an agent explore its environment.

The chapter does not yet discuss a 2-agent system for the Vacuum World, and in that respect is still incomplete.

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.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)

[26] 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)

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

[28] 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)

[29] 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

[30] 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 Multiagent Systems, AAMAS '16, pp. 1237–1246. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2016)

[31] Koeman, V.J., Pasman, W., Hindriks, K.V.: !! REPLACE ME !!-Eclipse User Manual. `https://goalapl.dev/GOALUserManual.pdf` (2021)

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

[33] 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)

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

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

[36] 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)

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

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

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

[40] 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)

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

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

[43] |http://www.swi-prolog.org| (2018)

[44] `http://www.swi-prolog.org/pldoc/man?section=builtin` (2018)

[45] Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: Swi-prolog. Theory and Practice of Logic Programming **12**, 67–96 (2012)

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