Programming Multi-Agent Systems in AgentSpeak using Jason

Rafael H. Bordini
University of Durham, UK

Jomi Fred Hübner
University of Blumenau, Brazil

Michael Wooldridge
University of Liverpool, UK
Programming Multi-Agent Systems in AgentSpeak using \textit{Jason}
Wiley Series in Agent Technology

Series Editor: Michael Wooldridge, University of Liverpool, UK

The ‘Wiley Series in Agent Technology’ is a series of comprehensive practical guides and cutting-edge research titles on new developments in agent technologies. The series focuses on all aspects of developing agent-based applications, drawing from the Internet, Telecommunications, and Artificial Intelligence communities with a strong applications/technologies focus.

The books will provide timely, accurate and reliable information about the state of the art to researchers and developers in the Telecommunications and Computing sectors.

Titles in the series:
Padgham/Winikoff: Developing Intelligent Agent Systems 0470861207 (June 2004)
Bellifemine/Caire/Greenwood: Developing Multi-Agent Systems with JADE 978-0-470-05747-6 (February 2007)
Programming Multi-Agent Systems in AgentSpeak using *Jason*

Rafael H. Bordini  
*University of Durham, UK*

Jomi Fred Hübner  
*University of Blumenau, Brazil*

Michael Wooldridge  
*University of Liverpool, UK*
To Idahyr (in memoriam), Maria, Ricardo, Lizete, Roberto, Renato, Rubens, Liliane and Thays.  (RHB)

To Ilze, Morgana and Thales.  (JFH)

To Lily May and Thomas Llewelyn.  (MW)
# Contents

Preface xi

Acknowledgements xvii

1 Introduction 1
1.1 Autonomous Agents ........................................ 1
1.2 Characteristics of Agents .................................... 2
1.3 Multi-Agent Systems ........................................ 5
1.4 Hello World! .................................................. 7

2 The BDI Agent Model 15
2.1 Agent-Oriented Programming ................................. 15
2.2 Practical Reasoning ............................................ 17
2.3 A Computational Model of BDI Practical Reasoning .......... 20
2.4 The Procedural Reasoning System ............................ 22
2.5 Agent Communication ......................................... 25

3 The Jason Agent Programming Language 31
3.1 Beliefs ......................................................... 32
3.2 Goals ........................................................ 40
3.3 Plans ........................................................ 41
3.4 Example: A Complete Agent Program ........................ 58
3.5 Exercises ...................................................... 65

4 Jason Interpreter 67
4.1 The Reasoning Cycle .......................................... 67
4.2 Plan Failure .................................................... 86
4.3 Interpreter Configuration and Execution Modes ............... 93
4.4 Pre-Defined Plan Annotations .................................. 97
4.5 Exercises ...................................................... 98
# CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Environments</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>Support for Defining Simulated Environments</td>
<td>102</td>
</tr>
<tr>
<td>5.2</td>
<td>Example: Running a System of Multiple Situated Agents</td>
<td>109</td>
</tr>
<tr>
<td>5.3</td>
<td>Exercises</td>
<td>114</td>
</tr>
<tr>
<td>6</td>
<td>Communication and Interaction</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>Available Performatives</td>
<td>118</td>
</tr>
<tr>
<td>6.2</td>
<td>Informal Semantics of Receiving Messages</td>
<td>119</td>
</tr>
<tr>
<td>6.3</td>
<td>Example: Contract Net Protocol</td>
<td>130</td>
</tr>
<tr>
<td>6.4</td>
<td>Exercises</td>
<td>135</td>
</tr>
<tr>
<td>7</td>
<td>User-Defined Components</td>
<td></td>
</tr>
<tr>
<td>7.1</td>
<td>Defining New Internal Actions</td>
<td>140</td>
</tr>
<tr>
<td>7.2</td>
<td>Customising the Agent Class</td>
<td>144</td>
</tr>
<tr>
<td>7.3</td>
<td>Customising the Overall Architecture</td>
<td>151</td>
</tr>
<tr>
<td>7.4</td>
<td>Customising the Belief Base</td>
<td>155</td>
</tr>
<tr>
<td>7.5</td>
<td>Pre-Processing Directives</td>
<td>160</td>
</tr>
<tr>
<td>7.6</td>
<td>Exercises</td>
<td>162</td>
</tr>
<tr>
<td>8</td>
<td>Advanced Goal-Based Programming</td>
<td></td>
</tr>
<tr>
<td>8.1</td>
<td>BDI Programming</td>
<td>166</td>
</tr>
<tr>
<td>8.2</td>
<td>Declarative (Achievement) Goal Patterns</td>
<td>169</td>
</tr>
<tr>
<td>8.3</td>
<td>Commitment Strategy Patterns</td>
<td>172</td>
</tr>
<tr>
<td>8.4</td>
<td>Other Useful Patterns</td>
<td>175</td>
</tr>
<tr>
<td>8.5</td>
<td>Pre-Processing Directives for Plan Patterns</td>
<td>176</td>
</tr>
<tr>
<td>9</td>
<td>Case Studies</td>
<td></td>
</tr>
<tr>
<td>9.1</td>
<td>Case Study I: Gold Miners</td>
<td>180</td>
</tr>
<tr>
<td>9.2</td>
<td>Case Study II: Electronic Bookstore</td>
<td>194</td>
</tr>
<tr>
<td>10</td>
<td>Formal Semantics</td>
<td></td>
</tr>
<tr>
<td>10.1</td>
<td>Semantic Rules</td>
<td>207</td>
</tr>
<tr>
<td>10.2</td>
<td>Semantics of Message Exchange in a Multi-Agent System</td>
<td>212</td>
</tr>
<tr>
<td>10.3</td>
<td>Semantic Rules for Receiving Messages</td>
<td>215</td>
</tr>
<tr>
<td>10.4</td>
<td>Semantics of the BDI Modalities for AgentSpeak</td>
<td>221</td>
</tr>
<tr>
<td>11</td>
<td>Conclusions</td>
<td></td>
</tr>
<tr>
<td>11.1</td>
<td><em>Jason</em> and Agent-Oriented Programming</td>
<td>225</td>
</tr>
<tr>
<td>11.2</td>
<td>Ongoing Work and Related Research</td>
<td>227</td>
</tr>
<tr>
<td>11.3</td>
<td>General Advice on Programming Style and Practice</td>
<td>230</td>
</tr>
</tbody>
</table>
A typical day:

You are waiting for the bus to take you to the airport: it is late, and you begin to be concerned about whether, if it is much later, you will have enough time to make your flight. You decide to abandon your plan of travelling by bus, and flag down a taxi instead. The taxi driver does not know which terminal your flight departs from; neither do you. As you approach the airport, you telephone the information line number printed on your ticket, and you are told that the flight departs from terminal one. The taxi drops you off at terminal one, and you enter the terminal, to be confronted by long queues at the check-in counters. You realise that, since you have no check-in baggage, you can use an express check-in counter, and within a minute, an attendant is checking you in. The attendant asks you whether you packed the bag yourself. No, you reply, but immediately point out to her that the bag was packed by your partner. You are given a boarding card and told to proceed through security; you do not know where the security gate is, so you ask. Proceeding through security, the metal detector goes off; you realise your mobile phone is in your pocket, so you remove it, but the detector still goes off. You remove your belt, then your wallet, and you are allowed through. In the departure lounge, you look for a cafe, and buy a sandwich. While you are eating your sandwich, you open your laptop, and quickly check through your email, deleting a dozen or so spam messages, scanning the genuine mail for anything important. While doing this, you keep half an eye on the departure board; your flight is called, so you finish your sandwich, shutdown your laptop, and board.

There does not seem to be anything magical about this scenario, does there? Yet an enormous range of human skills are in evidence here, which are worth identifying. Here are some examples:
• We start the day with some goal in mind (be on a particular flight at the airport at a particular time), and we have some idea of how we will achieve this – we have a plan of action to get us to the airport, involving a bus. Yet the plan is only partially elaborated – we do not know, for example, exactly what time the bus will turn up, exactly where it will drop us, how much it will cost, or where exactly at the airport the check-in desks or security gate are. However, we are able to carry out such thinly specified plans, filling in the details as required on the fly.

• When things go wrong, as they do in this scenario, we seem to be able to recover more-or-less seamlessly. The bus does not turn up: but we do not do a system crash, and we do not give up and go home. We quickly develop an alternative plan (flag down a taxi), and carry it out.

• While we are executing our plan we realise that we need some information – the terminal number – for our current plan to be successful. We perform an action (calling the airport information number) that will furnish us with this information.

• Throughout the day, we cooperate and coordinate our activities with other people – asking questions, answering their questions, solving problems together. We are able to anticipate their needs, and act to accommodate them. When the taxi driver tells us that she does not know which terminal our flight departs from, we work with her to find out. When the check-in attendant asks us if we packed the bag ourselves and we answer ‘no’, we anticipate that this will lead to further questions, and so we anticipate these, pro-actively providing the information we know the attendant wants. When we are going through the metal detector, we work with the security personnel to find out what is setting off the alarm.

• We interleave multiple activities, concurrently, each of them attempting to achieve one of our goals. We eat our sandwich, while reading email, while watching the departure board for details of our flight departure.

Still, most people would not regard this scenario as anything magic: it is everyday; positively routine, hardly worth mentioning. For researchers in artificial intelligence, however, the scenario is positively dripping with magic. The kinds of skills we describe above, which seem so everyday to us, are notoriously, frustratingly, irritatingly hard to deploy in computer systems. We do not get computers to do things for us by giving them some high-level goal, and simply letting them get on with it: we have to tell them what to do by giving them a precise, tediously detailed list of instructions, called a ‘program’, which the computer blindly executes. When computer systems encounter unexpected scenarios, and things do
not go as we anticipated, they do not seamlessly recover and develop alternative courses of action. We get an uncaught exception at best; a complete system failure at worst. Computers do not cooperate with us, anticipate our needs, and coordinate their activities around us. Indeed, computers often seem sullen, uncooperative and indeed positively unhelpful. Nor can we communicate them in any high-level way: everything that they do seems to have to be communicated to them by us selecting items from menus, clicking on a button or dragging something across a window – the idea that the computer anticipates our needs is definitely laughable.

This book is all about writing computer programs that have some flavour of the kind of skills illustrated above. We should put in an important disclaimer at this point: we do not claim that our programs come close to the abilities evident in the scenario. However, after you have finished reading this book, you should have a reasonable understanding of what the main issues are, and what sorts of techniques are currently available for programming software with these kinds of skills.

The book describes a programming language called ‘AgentSpeak’, and specifically, an implementation of the AgentSpeak language called Jason. The basic idea behind programming in AgentSpeak is to define the know-how of a program, in the form of plans. By ‘know-how’, we mean knowledge about how to do things. In the above scenario, for example, our know-how relates to travelling to the airport, and for this we have two plans: one involving a bus, the other involving a taxi. Programming in AgentSpeak thus involves encoding such plans, and AgentSpeak provides a rich, high-level language that allows programmers to capture this ‘procedural knowledge’ in a transparent manner. Plans in AgentSpeak are also used to characterise responses to events. In this way, we can build programs that not only systematically try to accomplish goals by making use of the know-how we provide, but they can also be responsive to their changing environment. The resulting programming paradigm incorporates aspects of conventional, so-called procedural programming, as embodied in languages like Pascal and C, as well as a kind of declarative, deductive style of programming, as embodied in Prolog. Ultimately, though, while these analogies may be useful, the style of programming in AgentSpeak is fundamentally different from either of these paradigms, as you will see in the remainder of the book. These differences are summed up in the following way: we refer to computer programs in AgentSpeak as agents. This is because they are active pieces of software, not dumbly providing services for us or other pieces of software, but constantly trying to achieve goals in their environment, making use of their know-how.

Another important aspect of AgentSpeak and Jason in particular is that it is designed with cooperation in mind. Thus, Jason comes with a rich environment that will enable agents to communicate and coordinate with one another in a high-level way. Communication in Jason is also rather different from the communication you may be familiar with from other programming languages. It is
not concerned with low-level issues (getting a byte from \( A \) to \( B \)); nor is it concerned with service provision via methods or the like. The focus is on knowledge level communication: it addresses issues such as how an agent can communicate its beliefs to another, how an agent can delegate one of its goals, and how an agent can communicate its own know-how.

The book you are holding provides a practical introduction to AgentSpeak and Jason; it gives a detailed overview of how to make the most of the Jason language, hints and tips on good AgentSpeak programming style, a detailed overview of the Jason programming environment, a formal semantics for the language, and some pointers to where the language came from, as well as where it is going. Whether you are reading this book in order to build agent systems, or just curious about a new style of programming, we hope you will find it instructive, useful and thought provoking. We conclude with advice that is by now traditional for programming books. While you might understand how the language works in principle from reading the book, there is ultimately no substitute for having a go yourself. Jason is freely available: so why not download it, try, learn and enjoy. Links to the software download page and teaching resources (slides, solved exercises, etc.) are available at http://jason.sf.net/jBook.

Structure of this Book

- The first two chapters of the book provide essentially background material, giving an introduction to agents and multi-agent systems, and a short overview of the BDI model of agency, which underpins AgentSpeak. If you know something about agents, and want to go straight into the language, then you can skip these chapters.

- Chapter 3 is probably the most important chapter in the book: it gives a detailed introduction to the AgentSpeak language as implemented in the Jason system. If you want to know how to program in AgentSpeak, you need to read this chapter.

- Chapter 4 describes how the Jason interpreter works: if you want to understand exactly how your programs will behave, and optimise your code, then you need to read this chapter to understand how the interpreter deals with them.

- Chapter 5 describes Jason’s implementation of communication.

- Chapter 6 describes the support available for developing (in Java) simulated environments where agents are to be situated.
• Chapter 7 shows how you can replace parts of *Jason* with your own Java code. If you want your agent to be embedded in some system, or to talk with some legacy system, for example, then you need to read this chapter.

• Chapter 8 describes various ‘programming patterns’ that can be used when programming with AgentSpeak using *Jason*. Roughly, this chapter is about programming style.

• Chapter 9 gives some working case studies.

• Chapter 10 presents formal semantics for AgentSpeak.

• Chapter 11 presents some conclusions, and ongoing research issues.

Thus, if you just want to understand the basic principles of AgentSpeak, then Chapters 3 and 4 are the most important. If you want to do serious system development, then Chapters 3 – 9 are the most important. If you want to do research on AgentSpeak itself, then Chapters 10 and 11 are also important.
Acknowledgements

A significant body of research has helped to shape Jason in its current form. Such research was done by the authors of this book in collaboration with many colleagues, including: Michael Fisher, Willem Visser, Álvaro Moreira, Renata Vieira, Natasha Alechina, Brian Logan, Viviana Mascardi, Davide Ancona, Antônio Carlos da Rocha Costa and Fabio Okuyama. The programming of the Jason platform was done by Jomi Hübner and Rafael Bordini with much appreciated support from Joyce Martins, who helped us improve the grammar used for the parsing of AgentSpeak code. With the many extensions to the language that we did over the last few years, Joyce has been a continued source of help: many thanks to her! Many students have helped in various different ways, and we would like to thank specially some of the first students to be involved with implementations of AgentSpeak interpreters and using them, for their brave work: especially Rodrigo Machado, but also Daniel Basso, Rafael Jannone and Denise de Oliveira. The undergraduate students in the Multi-Agent Systems module at Durham University and final year project students at Durham and FURB using Jason also contributed enormously in this development, including Alisson Appio, Daniel Dalcastagne, George Millard, Marios Richards, Daniel Tallentire, Karlyson Vargas and Daniel Wilkinson. Equally, students in various courses/projects in Brazil, Netherlands, England, Italy, Portugal and Australia, as well as subscribers to Jason email lists, should be thanked for the courage in using Jason during a time it was evolving very quickly; their experience certainly helped us a lot. Many others have also helped us, and the Jason manual lists some more names; many thanks to all those who helped (e.g. by pointing out bugs). Finally, we would like to thank Patricia Shaw and Berndt Farwer who read drafts of some chapters of this book, and of course Birgit, Richard and Sarah from Wiley for all the help during the (longer than hoped) period it took us to write this book.
1

Introduction

This book is all about a computer programming language called AgentSpeak, and a particular implementation of AgentSpeak called Jason. The AgentSpeak language is intended for developing multi-agent systems. Before we start to investigate how to program with AgentSpeak, it seems appropriate to try to understand in more detail what multi-agent systems are, and some of the ideas that underpin the language.

1.1 Autonomous Agents

To better understand what we mean by the terms ‘agent’ and ‘multi-agent systems’, let us consider how agents relate to other types of software. Start by considering functional programs, which are possibly the simplest type of software from the point of view of software development and software engineering. A functional program takes some input, chews over this input, and then on the basis of this, produces some output and halts. A compiler is an example of such a program: the input is some source code (e.g. a .java file), and the output is bytecode (.class files), object code or machine code. When we learn how to program, the types of program we typically construct are of this type: the sorts of exercises we set to programmers in an introductory Java class are things like ‘read a list of numbers and print the average’. Functional programs are so called because, mathematically, we can think of them as functions $f : I \rightarrow O$ from some domain $I$ of possible inputs (source code programs, in our ‘compiler’ example) to some range $O$ of possible outputs (bytecode, object code, etc). We have a range of well-established techniques for developing such programs; the point is that, from the standpoint of software development, they are typically straightforward to engineer.
Unfortunately, many programs do not have this simple input – compute – output operational structure. In particular, many of the systems we need to build in practice have a ‘reactive’ flavour, in the sense that they have to maintain a long-term, ongoing interaction with their environment; they do not simply compute some function of an input and then terminate:

Reactive systems are systems that cannot adequately be described by the relational or functional view. The relational view regards programs as functions ... from an initial state to a terminal state. Typically, the main role of reactive systems is to maintain an interaction with their environment, and therefore must be described (and specified) in terms of their on-going behavior ... Every concurrent system ... must be studied by behavioral means. This is because each individual module in a concurrent system is a reactive subsystem, interacting with its own environment which consists of the other modules. [77]

Examples of such programs include computer operating systems, process control systems, online banking systems, web servers, and the like. It is, sadly, a well-known fact that, from the software development point of view, such reactive systems are much harder to correctly and efficiently engineer than functional systems.

A still more complex class of systems is a subset of reactive systems that we will call agents. An agent is a reactive system that exhibits some degree of autonomy in the sense that we delegate some task to it, and the system itself determines how best to achieve this task. We call such systems ‘agents’ because we think of them as being active, purposeful producers of actions: they are sent out into their environment to achieve goals for us, and we want them to actively pursue these goals, figuring out for themselves how best to accomplish these goals, rather than having to be told in low-level detail how to do it. We can imagine such agents being delegated a task like booking a holiday for us, or bidding on our behalf in an online auction, or cleaning our office space for us, if they are robotic agents.

1.2 Characteristics of Agents

Let us try to be a little more precise about what sorts of properties we are thinking of when we talk about agents. We consider agents to be systems that are situated in some environment. By this, we mean that agents are capable of sensing their environment (via sensors), and have a repertoire of possible actions that they can perform (via effectors or actuators) in order to modify their environment. The key question facing the agent is how to go from sensor input to action output: how to decide what to do based on the information obtained via sensors. This leads to the
view of an agent as shown in Figure 1.1. As we will see, in AgentSpeak, deciding what to do is achieved by manipulating *plans*.

The environment that an agent occupies may be physical (in the case of robots inhabiting the physical world) or a software environment (in the case of a software agent inhabiting a computer operating system or network). We think of decisions about what action to perform being translated into actual actions via some mechanism external to the agent; usually, this is achieved via some sort of API. In almost all realistic applications, agents have at best *partial* control over their environment. Thus, while they can perform actions that change their environment, they cannot in general completely control it. Very often this is because there will be other agents in the environment, who exhibit control over their part of the environment.

Apart from being situated in an environment, what other properties do we expect a rational agent to have? Wooldridge and Jennings [104] argued that agents should have the following properties:

- autonomy;
- proactiveness;
- reactivity; and
- social ability.
Autonomy

It is important to realise that autonomy is a very broad spectrum. At one end of the spectrum, we have computer programs such as conventional word processors and spreadsheets, which exhibit little or no autonomy. Everything that happens with such an application happens because you make it happen – you select a menu item, or click on an icon, for example. Such programs, by and large, do not take the initiative in any sense. At the other end of the autonomy spectrum are you and us. You are completely autonomous. You can ultimately choose to believe what you want, and do what you want – although society typically constrains your autonomy in various ways, preventing you from doing certain things, for the sake of you and your peers. You have your own goals, your own agenda, and autonomy means they really are yours: nobody and nothing explicitly dictates them to you. (Of course, you might argue that society tries to shape our beliefs and goals, but that is another story.) In this book, we are interested in computer programs that lie somewhere between these two extremes. Roughly speaking, we want to be able to delegate goals to agents, which then decide how best to act in order to achieve these goals. Thus, our agent’s ability to construct goals is ultimately bounded by the goals that we delegate. Moreover, the way in which our agents will act to accomplish their goals will be bounded by the plans which we give to an agent, which define the ways in which an agent can act to achieve goals and sub-goals. One of the key ideas in AgentSpeak is that of an agent putting together these plans on the fly in order to construct more complex overall plans to achieve our goals.

At its simplest, then, autonomy means nothing more than being able to operate independently in order to achieve the goals we delegate to an agent. Thus, at the very least, an autonomous agent makes independent decisions about how to achieve its delegated goals – its decisions (and hence its actions) are under its own control, and are not driven by others.

Proactiveness

Proactiveness means being able to exhibit goal-directed behaviour. If an agent has been delegated a particular goal, then we expect the agent to try to achieve this goal. Proactiveness rules out entirely passive agents, who never try to do anything. Thus, we do not usually think of an object, in the sense of Java, as being an agent: such an object is essentially passive until something invokes a method on it, i.e. tells it what to do. Similar comments apply to web services.

Reactiveness

Being reactive means being responsive to changes in the environment. In everyday life, plans rarely run smoothly. They are frequently thwarted, accidentally or
deliberately. When we become aware that our plans have gone wrong, we respond, choosing an alternative course of action. Some of these responses are at the level of ‘reflexes’ – you feel your hand burning, so you pull it away from the fire. However, some responses require more deliberation – the bus has not turned up, so how am I going to get to the airport? Designing a system which simply responds to environmental stimuli in a reflexive way is not hard – we can implement such a system as a lookup table, which simply maps environment states directly to actions. Similarly, developing a purely goal-driven system is not hard. (After all, this is ultimately what conventional computer programs are: they are just pieces of code designed to achieve certain goals.) However, implementing a system that achieves an effective balance between goal-directed and reactive behaviour turns out to be hard. This is one of the key design objectives of AgentSpeak.

Social Ability

Every day, millions of computers across the world routinely exchange information with humans and other computers. In this sense, building computer systems that have some kind of social ability is not hard. However, the ability to exchange bytes is not social ability in the sense that we mean it. We are talking about the ability of agents to cooperate and coordinate activities with other agents, in order to accomplish our goals. As we will see later, in order to realise this kind of social ability, it is useful to have agents that can communicate not just in terms of exchanging bytes or by invoking methods on one another, but that can communicate at the knowledge level. That is, we want agents to be able to communicate their beliefs, goals and plans to one another.

1.3 Multi-Agent Systems

So far, we have talked about agents occupying an environment in isolation. In practice, ‘single agent systems’ are rare. The more common case is for agents to inhabit an environment which contains other agents, giving a multi-agent system [103]. Figure 1.2 gives an overview of a multi-agent system. At the bottom of the figure, we see the shared environment that the agents occupy; each agent has a ‘sphere of influence’ in this environment, i.e. a portion of the environment that they are able to control or partially control. It may be that an agent has the unique ability to control part of its environment, but more generally, and more problematically, we have the possibility that the spheres of influence overlap: that is, the environment is jointly controlled. This makes life for our agents more complicated, because to achieve an outcome in the environment that our agent desires, it will have to take into account how the other agents with some control are likely to act.
Above the environment, we see the agents themselves, which stand in various organisational relationships to one another (for example, one agent may be the peer of another, or may have line authority over another). Finally, these agents will have some knowledge of each other, though it may be the case that an agent does not have complete knowledge of the other agents in the system.

**Programming Languages for Agents and Multi-Agent Systems**

We now have some idea of what kinds of properties we are thinking of in our agents. So, suppose we want to program these things: what do these properties tell us about the kinds of programming language or environment that we might use for programming autonomous agents? We can identify the following requirements:
The language should support delegation at the level of goals. As we noted earlier, when we delegate a task to an agent, we do not generally want to do this by giving the agent an executable description of what to do. Rather, we want to communicate with it at the level of goals: we should be able to describe our goals to an agent in a high-level way, independent of approaches to achieving these goals.

The language should provide support for goal-directed problem solving. We want our agents to be able to act to achieve our delegated goals, systematically trying to achieve them.

The language should lend itself to the production of systems that are responsive to their environment.

The language should cleanly integrate goal-directed and responsive behaviour.

The language should support knowledge-level communication and cooperation.

These are the main requirements that AgentSpeak and Jason are intended to fulfill. In the following section, we will give a very brief introduction to AgentSpeak, which will give a feel for how some of these features are provided.

1.4 Hello World!

When introducing a new programming language, it has become the tradition to give a short example program, the purpose of which is simply to display the text ‘Hello World!’ to the programmer.¹ For example, here is a ‘Hello World’ program in Java:

```java
public class HelloWorld {
    public static void main( String args[] ) {
        System.out.println( "Hello World!" );
    }
}
```

Trivial though they are, running a ‘Hello World’ program helps to give a programmer confidence with the new language, and very often they give a useful insight

¹There are even web sites devoted to ‘Hello World’ programs: http://www.roesler-ac.de/wolfram/hello.htm is one example, with ‘Hello World’ programs for hundreds of languages.
INTRODUCTION

into the ‘mind set’ of the language. Therefore, we strongly encourage you to try this exercise:

```plaintext
started.

+started <- .print("Hello World!").
```

Let us try to understand a little of what is going on here, although we will save the details for later.

The first thing to understand is that this constitutes the definition of a single agent. This definition will often be saved in a single file, and let us suppose that on our system we have called this file `hello.asl`; the `.asl` extension will be used for all our AgentSpeak programs. Now, the definition of an agent in AgentSpeak consists of two parts:

- the agent’s *initial beliefs* (and possibly *initial goals*); and
- the agent’s *plans*.

The first line defines one initial belief for our agent. (Although there is only one initial belief here, we could have given a list.) AgentSpeak does not have variables as in programming languages such as Java or C; the constructs are specific agent notions, such as beliefs, goals and plans. We need ‘beliefs’ because the intuition is that they represent the information that the agent has currently been able to obtain about its environment. The full stop, ‘.’, is a syntactic separator, much as a semicolon is in Java or C. Therefore, when our agent first starts running, it will have the single belief `started`; intuitively, it has the belief that it has started running. Notice that there is no magic in the use of the term ‘started’; we could just as well have written the program as follows, and we would have obtained exactly the same result:

```plaintext
cogitoErgoSum.

+cogitoErgoSum <- .print("Hello World!").
```

So, while it’s important and helpful for programmers to pick sensible and useful names for beliefs, the computer, of course, could not care less. One rule worth remembering at this stage is that beliefs must start with a lowercase letter.

Next, we have the line

```plaintext
+started <- .print("Hello World!").
```

which defines a plan for the agent — which is in fact the only plan this agent has. Intuitively, we can read this plan as meaning
1.4. HELLO WORLD!

whenever you come to believe ‘started’, print the text ‘Hello World!’.

The plan, like all AgentSpeak plans, comes in three parts. The first part is a triggering event. In this case, the triggering event is simply

+started

The symbol ‘+’ in this context means ‘when you acquire the belief . . . ’, and so overall the triggering condition is ‘when you acquire the belief “started”’. We know from the above discussion that the agent acquires this belief when it starts executing, and so in sum, this plan will be triggered when the agent starts executing.

However, what does it mean, to trigger a plan? The idea is that the trigger of a plan defines the events that it is useful for handling. In this case, the event is the acquisition of a particular new belief. A plan is triggered when events occur which match its trigger condition, and when this happens, the plan becomes ‘active’; it becomes something that the agent ‘considers doing’. However, before an agent selects a plan to become active, it checks that the context of the plan is appropriate.

In the hello world case, the context is in fact empty, which can be understood as meaning ‘this plan is always good’. As we will see later, in the context part of plans, we can define complex conditions which an agent uses to determine whether or not to choose a particular plan for a given event. In particular, an agent can have multiple plans triggered by the same event which deal with this event in different ways: thus an agent can have multiple different responses to events, and can choose between these depending on the situation in which it currently finds itself.

In this case, there is just one plan that can be triggered by the event, and since the context is empty, it is always applicable, and so the AgentSpeak interpreter directly executes the body of that plan. In this case, the body of the plan is very simple, containing a single action:

.print("Hello World!").

As you might guess, the effect of this action is simply to display the text ‘Hello World!’ on the user’s console. Running this example on Jason, the result is that the following gets displayed in the user’s console (see Figure 1.3):

[hello] saying: Hello World!

There are several points to note here. First, although .print(...) looks like a belief, it is in fact an action, as it appears in the plan body; to give the reader some syntactic clues when reading a Jason program, ‘internal’ actions begin with a full stop (actions normally change the environment, but not internal actions). In fact, .print(...) is a pre-defined internal action in Jason: other pre-defined actions
include, for example \texttt{send(...)} and \texttt{broadcast(...)} for agent communication, and \texttt{stopMAS}, to cause a halt to the execution of the multi-agent system. Appendix A describes other internal actions available in \texttt{Jason}.

Don’t be misled by the very simple structure of this example into thinking that plan bodies are just like sub-routines, methods or procedures from conventional programming languages: they are in fact very much more than that. For example, one of the key ideas in AgentSpeak is that we can have a \textit{goal} in a plan body. The idea is that, when the interpreter comes to a goal in a plan body, it tries to find a plan that achieves this goal; as we discussed above, there may be several such plans, which may or may not match the current context. The upshot of this is that it is possible, within AgentSpeak, to invoke code \textit{with reference to the effect of the code}, and the same invocation, in different contexts, results in different code being invoked. This is a very substantial difference to conventional languages like Java or C.

Finally – and this may at first sight seem somewhat perverse to those with a grounding in the theory of computing – if you run this example, you will see that the program does not terminate! We are used to thinking of non-termination (infinite loops and the like) as a bad thing. In the case of AgentSpeak, however, we