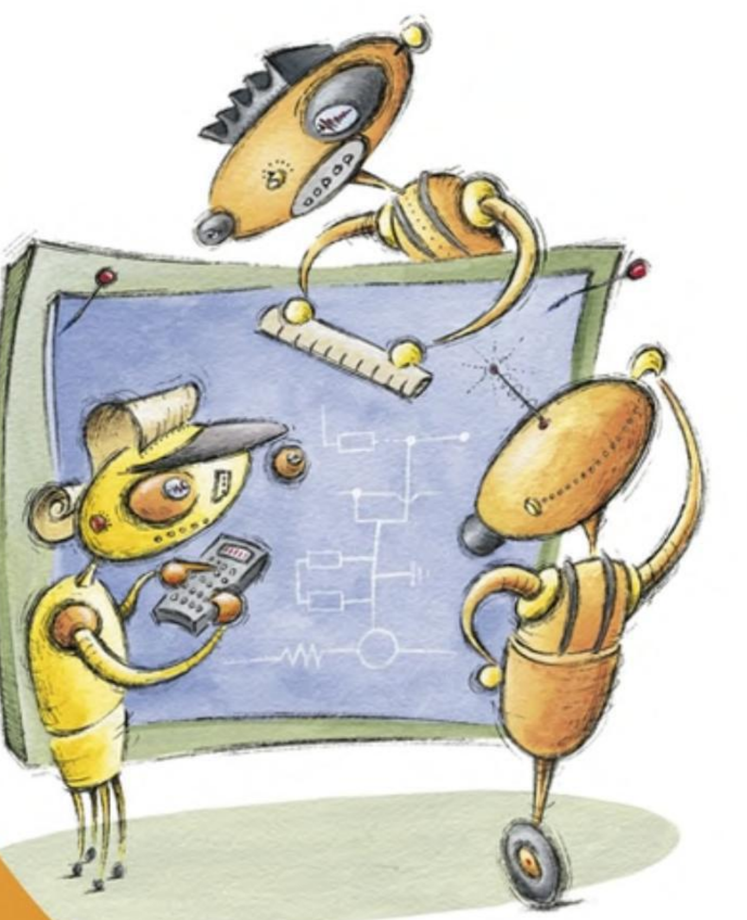

SECOND EDITION

# An Introduction to MultiAgent Systems

MICHAEL WOOLDRIDGE

Copyrighted material

# An Introduction to MultiAgent Systems

Second Edition

Michael Wooldridge

*Department of Computer Science, University of Liverpool*

WILEY

A John Wiley and Sons, Ltd, Publication

Copyrighted material

This edition first published 2009
© 2009 John Wiley & Sons Ltd

*Registered office*
John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, United Kingdom

For details of our global editorial offices, for customer services and for information about how to apply for permission to reuse the copyright material in this book please see our website at www.wiley.com.

The right of the author to be identified as the author of this work has been asserted in accordance with the Copyright, Designs and Patents Act 1988.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, except as permitted by the UK Copyright, Designs and Patents Act 1988, without the prior permission of the publisher.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Designations used by companies to distinguish their products are often claimed as trademarks. All brand names and product names used in this book are trade names, service marks, trademarks or registered trademarks of their respective owners. The publisher is not associated with any product or vendor mentioned in this book. This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold on the understanding that the publisher is not engaged in rendering professional services. If professional advice or other expert assistance is required, the services of a competent professional should be sought.

*Library of Congress Cataloging-in-Publication Data*

Wooldridge, Michael J., 1966-
An introduction to multiagent systems / Michael Wooldridge. – 2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-470-51946-2 (pbk.)
1. Intelligent agents (Computer software) I. Title.
QA76.76.I58W65 2009
006.3–dc22
2009004188

A catalogue record for this book is available from the British Library.

ISBN 9780470519462 (pbk.)

Set in 10/12pt Palatino by Sunrise Setting Ltd, Torquay, UK.
Printed in Great Britain by Bell & Bain, Glasgow.

Copyrighted material

# Contents

Copyrighted material

Copyrighted material

Copyrighted material

Copyrighted material

Copyrighted material

Copyrighted material

Table 1: A summary of notation.

| | |
|---|---|
| Sets | |
| $\{a, b, c\}$ | the set containing elements $a$, $b$, and $c$ |
| $\emptyset$ | the empty set (contains nothing) |
| $a \in S$ | $a$ is a member of set $S$, e.g., $a \in \{a, b, c\}$ |
| $\{x \mid P(x)\}$ | set of objects $x$ with property $P$ |
| $S_1 \subseteq S_2$ | set $S_1$ is a subset of set $S_2$, e.g., $\{b\} \subseteq \{a, b, c\}$ |
| $S_1 \cap S_2$ | the intersection of $S_1$ and $S_2$, e.g., $\{a, b\} \cap \{b\} = \{b\}$ |
| $S_1 \cup S_2$ | the union of $S_1$ and $S_2$, e.g., $\{a, b\} \cup \{b, c\} = \{a, b, c\}$ |
| $S_1 \setminus S_2$ | the difference of $S_1$ and $S_2$, e.g., $\{a, b\} \setminus \{b\} = \{a\}$ |
| $\mathbf{2}^S$ | powerset of $S$, e.g., $\mathbf{2}^{\{a,b\}} = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$ |
| $\lvert S \rvert$ | cardinality of $S$ (number of elements it contains) |
| Common sets | |
| $\mathbb{N}$ | the natural numbers: $0, 1, 2, 3, \ldots$ |
| $\mathbb{R}$ | the real numbers |
| $\mathbb{R}_+$ | the positive real numbers |
| Relations and functions | |
| $(a, b)$ | a pair of objects, first element $a$ second element $b$ |
| $S_1 \times S_2$ | Cartesian product (a.k.a. cross product) of $S_1$ and $S_2$ |
| $S_1 \times \cdots \times S_n$ | Cartesian product of sets $S_1, S_2, \ldots, S_n$ |
| $\langle a_1, a_2, \ldots, a_n \rangle$ | tuple consisting of elements $a_1, a_2, \ldots, a_n$ |
| $f : D \to R$ | a function $f$ with domain $D$ and range $R$ |
| $f(x)$ | the value given by function $f$ for input $x$ |
| Permutations | |
| $\Pi(S)$ | the possible permutations of set $S$, e.g., $\Pi(\{a, b\}) = \{(a, b), (b, a)\}$ |
| Logic | |
| $\top$ | the Boolean value for truth |
| $\bot$ | the Boolean value for falsity |
| $\phi, \psi$ | logical formulae |
| $\neg$ | negation ('not'), e.g., $\neg\bot = \top$ |
| $\vee$ | disjunction ('or'), $\phi \vee \psi = \top$ iff either $\phi = \top$ or $\psi = \top$ |
| $\wedge$ | conjunction ('and'), $\phi \wedge \psi = \top$ iff both $\phi = \top$ and $\psi = \top$ |
| $\rightarrow$ | implication ('implies'), $\phi \rightarrow \psi = \top$ iff $\phi = \bot$ or $\psi = \top$ |
| $\leftrightarrow$ | biconditional ('iff'), $\phi \leftrightarrow \psi$ is the same as $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$ |
| $DB$ | a database – a set of logical formulae |
| $DB \vdash \phi$ | logical proof – $\phi$ can be proved from $DB$ |
| $I \models \phi$ | formula $\phi$ is true under interpretation $I$ |

- an introduction to the basic ideas of intelligent autonomous agents (Chapter 2) and then an introduction to the main approaches to building such agents (Chapters 3–5)

- an introduction to the main approaches to building multiagent systems in which agents can communicate and cooperate to solve problems (Chapters 6–10)

Copyrighted material

- an introduction to decision-making in multiagent systems and logical modelling of multiagent systems (Chapters 11–17).

Although the book is not heavily mathematical, there is inevitably *some* mathematics. A coherent course, avoiding the more mathematical sections, would include Chapters 1 to 10. A course on the principles of multiagent systems would include Chapters 1 and 2, and then move on to Chapters 11 to 17.

Complete lecture slides, exercises, and other associated teaching material are available at:

`http://www.csc.liv.ac.uk/~mjw/pubs/imas/`

I welcome additional teaching materials (e.g., tutorial/discussion questions, exam papers and so on), which I will make available on an 'open source' basis – please email them to me at:

`mjw@liv.ac.uk`.

## Chapter structure

Every chapter of the book ends with the following elements:

- A 'class reading' suggestion, which lists one or two key articles from the research literature that may be suitable for class reading in seminar-based courses.
- A 'notes and further reading' section, which provides additional technical comments on the chapter and extensive pointers into the literature for advanced reading. This section is aimed at those who wish to gain a deeper, research-level understanding of the material.
- A 'mind map', which gives a pictorial summary of the main concepts in the chapter, and how these concepts relate to one another. The hope is that the mind maps will be useful as a memory and revision aid.

## What was left out and why

Part of the joy in working in the multiagent systems field is that it takes inspiration from, and in turn contributes to, a very wide range of other disciplines. The field is in part artificial intelligence (AI), part economics, part software engineering, part social sciences, and so on. But this poses a real problem for anyone writing a book on the subject, namely, what to put in and what to leave out. While there is a large research literature on agents, there are not too many models to look at with respect to textbooks on the subject, and so I have had to make some hard choices here. When deciding what to put in/leave out, I have been guided to a great extent by what the 'mainstream' multiagent systems literature regards as important, as evidenced by the volume of published papers on the subject. The second consideration was what might reasonably be (i) taught and (ii) understood in the context of a typical one-semester university course. This largely excluded most abstract theoretical material, which will probably make most students happy (if not their teachers).

Copyrighted material

I deliberately chose to emphasize some aspects of agent work, and give less emphasis to others. In particular, I did not give much emphasis to the following:

**Learning** It goes without saying that learning is an important agent capability. However, machine learning is an enormous area of research in its own right, and consideration of this would take us at something of a tangent to the main concerns of the book. After some agonizing, I therefore decided not to cover learning. There are plenty of references to learning algorithms and techniques in agent systems: see, for example, [Kaelbling, 1993; Stone, 2000; Weiß, 1993; Weiß, 1997; Weiß and Sen, 1996].

**Artificial life** Some sections of this book (in Chapter 5 particularly) are closely related to work carried out in the artificial life, or 'alife', community. However, the work of the alife community is carried out largely independently of that in the 'mainstream' multiagent systems community. By and large, the two communities do not interact with one another. For these reasons, I have chosen not to focus on alife in this book. (Of course, this should not be interpreted as in any way impugning the work of the alife community: it just is not what this book is about.) There are many easily available references to alife on the Web. A useful starting point is [Langton, 1989]; another good reference is [Mitchell, 1996].

**Robotics** As will become evident later, many ideas in the multiagent systems community can trace their heritage to work on autonomous mobile robots. In particular, the agent decision-making architectures discussed in the first part of the book are drawn from this area. However, robotics has its own problems and techniques, distinct from those of the software agent community. In this book, I will focus almost exclusively on software agents, but will give some pointers to the autonomous robots community as appropriate. See [Matarić, 2007; Murphy, 2000] for introductions to autonomous robotics, and [Arkin, 1998; Thrun et al., 2005] for advanced topics.

**Software mobility** As with learning, I believe mobility is a useful agent capability, which is particularly valuable for some applications. But, like learning, I do not view it to be central to the multiagent systems curriculum. In fact, I do touch on mobility, but only briefly: the interested reader will find plenty of references in Chapter 9.

In my opinion, the most important things for students to understand are (i) the 'big picture' of multiagent systems (why it is important, where it came from, what the issues are, and where it is going), and (ii) what the key tools, techniques, and principles are. I see no value in teaching students deep technical issues in (for example) the logical aspects of multiagent systems, or game-theoretic approaches to multiagent systems, if these students cannot understand or articulate why these issues are important, and how they relate to the 'big picture'. Similarly, teaching students how to program agents using a particular programming language or development platform is of severely limited value if these same students have no conception of the deeper issues surrounding the design and deployment of agents. Students who have some sense of the big picture, and have an understanding of the key issues, questions, and techniques, will be well equipped to make sense of the deeper literature if they choose to study it.

Copyrighted material

## Omissions and errors

> Unprovided with original learning, unformed in the habits of thinking, unskilled in the arts of composition, I resolved to write a book.
>
> Edward Gibbon

In writing this book, I tried to set out the main threads of work that make up the multiagent systems field, and to critically assess their relative merits. In doing so, I have tried to be as open-minded and even-handed as time and space permit. However, I will no doubt have unconsciously made my own foolish and ignorant prejudices visible, by way of omissions, oversights, and the like. If you find yourself speechless with rage at something I have omitted – or included, for that matter – then all I can suggest is that you accept my apology, and take solace from the fact that someone else is almost certainly more annoyed with the book than you are.

> Little did I imagine as I looked upon the results of my labours where these sheets of paper might finally take me. Publication is a powerful thing. It can bring a man all manner of unlooked-for events, making friends and enemies of perfect strangers, and much more besides.
>
> Matthew Kneale (*English Passengers*)

I have no doubt that the book contains many errors – some perhaps forgivable, and others unquestionably not. I assure you that this is more depressing for me than it is annoying for you, but I am cheered by the following commentary on Alan Turing's paper *On Computable Numbers* (the paper that introduced Turing machines, and thereby invented much of computer science):

> This is a brilliant paper, but the reader should be warned that many of the technical details are incorrect. ...It may well be found most instructive to read this paper for its general sweep, ignoring the petty technical details.
>
> Martin Davis (*The Undecidable*)

If Turing couldn't get the 'petty technical details' right, then what hope is there for us mere mortals? Nevertheless, comments, corrections, and suggestions for a possible third edition are welcome, and should be sent to the email address given above.

## Web references

It would be very hard to write a book about Web-related issues without giving URLs as references. In many cases, the best possible reference to a subject is a website, and given the speed with which the computing field evolves, many important topics are only documented in the 'conventional' literature very late in the day. But citing web pages as authorities can create big problems for the reader. Companies go bust, sites go dead, people move, research projects finish, and when these things happen, web references become useless. For these reasons, I have therefore attempted to keep web references to a minimum. I have preferred to cite the 'conventional' (i.e., printed) literature over web pages when given a choice. In addition, I have tried to cite only web pages that are likely to be stable and supported for the foreseeable future. The date associated with a web page is the date at which I checked the reference was working.

Copyrighted material

## What has changed since the first edition?

There was a time when I rather arrogantly believed I had read all the key papers in the multiagent systems field, and had a basic working knowledge of all the main research problems and techniques. Well, if that was ever true, then it certainly isn't any more, and hasn't been for nearly two decades: the time has long since passed when any one individual could have a deep understanding of the entire multiagent systems research area. I mentioned above that one of the joys of the multiagent systems area was its diversity, but of course this very diversity makes life hard not just for the student, but also for the textbook author. Since the first edition of this book appeared, literally thousands of research papers and dozens of books on multiagent systems have been published, and there now seems to be a truly dizzying collection of journals, conferences, and workshops specifically devoted to the topic. This makes it very hard indeed to decide what to include in a second edition.

The biggest single change since the first edition is the inclusion of much more material on game-theoretic aspects of multiagent systems. The reason for this is simple: game theory has been a huge growth area not just in multiagent systems research, but in computer science generally, and I felt it important that this explosion of interest was reflected in my coverage. The main changes in this respect are as follows. First, the introductory coverage of basic game-theoretic notions such as Nash equilibrium has been clarified and deepened. (For example, I was cavalier with the distinction between '$>$' and '$\geq$' in the first edition, and it seems these distinctions are quite important in game theory ...) Completely new material has been added on computational social choice theory (voting), and coalitions and coalition formation. The coverage of auctions has been extended considerably, to include combinatorial auctions and the basic principles of mechanism design. Auctions now get a chapter of their own, as does negotiation.

The other main changes are as follows. First, I have clarified and deepened the coverage of argumentation, which now gets a chapter of its own. Given the huge growth of the semantic web since 2001, ontologies and semantic web ontology languages also get a chapter of their own.

Apart from this, the main changes have been polishing (trimming some sections down where they were verbose), and generally updating material. I have also tidied up the figures, and added marginal notes for key concepts. There are not many changes to the chapters on agent architectures, as this has not been a very active research area since the publication of the first edition.

From my point of view, the main failing of the first edition was that I didn't emphasize computational aspects enough; I have tried to do this more systematically in the second edition, particularly in the sections on game-theoretic ideas. Finally, I have never heard anybody complain about a textbook having too many examples, so I have made an effort to include more of these.

Copyrighted material

Finally, I suppose I should acknowledge the British weather. The summer of 2008 was, by common consent, one of the wettest, coldest, greyest British summers in living memory. Had the sun shone, even occasionally, I might have been tempted out of my office, and this second edition would not have seen the light of day.

My personal life in the six years since the first edition of this book has been pretty busy, largely due to the arrival of Lily May Wooldridge on 10 May 2002, and Thomas Llewelyn Wooldridge on 13 August 2005. I am immensely proud and immensely lucky to be the father of such beautiful, happy, funny, and warm-hearted children. And of course Lily, Tom, and I are blessed to have Janine at the heart of our family.

*Mike Wooldridge*
Liverpool
Summer 2008

Copyrighted material

# Part I

# Setting the Scene

Copyrighted material

The aim of Chapter 1 is to sell you the multiagent systems project. If you want to understand a software technology, it helps to understand where the ideas underpinning this technology came from, and what the drivers and key challenges are behind this technology. In this chapter, we will see where the multiagent systems field emerged from in terms of ongoing trends in computing, what the long-term visions are for the multiagent systems field, how the multiagent systems paradigm relates to other trends in software, and what the key issues are in realizing the multiagent systems vision.

Copyrighted material

Chapter 1

# Introduction

The history of computing to date has been marked by five important, and continuing, trends:

- *ubiquity*
- *interconnection*
- *intelligence*
- *delegation*
- *human-orientation*.

UBIQUITY

By ubiquity, I simply mean that the continual reduction in cost of computing capability has made it possible to introduce processing power into places and devices where it would hitherto have been uneconomic, and perhaps even unimaginable. This trend will inevitably continue, making processing capability, and hence intelligence of a sort, ubiquitous.

INTERCONNECTION

While the earliest computer systems were isolated entities, communicating only with their human operators, computer systems today are usually *interconnected*. They are *networked* into large *distributed* systems. The Internet is the obvious example; it is becoming increasingly rare to find computers in use in commercial or academic settings that do not have the capability to access the Internet. Until a comparatively short time ago, distributed and concurrent systems were seen by many as strange and difficult beasts, best avoided. The very visible and very rapid growth of the Internet has (I hope) dispelled this perception forever. Today, and for the future, distributed and concurrent systems are essentially the norm in commercial and industrial computing, leading some researchers and practitioners to revisit the very foundations of computer science, seeking theoretical models that reflect the reality of computing as primarily a process of interaction.

INTELLIGENCE

The third trend is towards ever more *intelligent* systems. By this, I mean that the *complexity* of tasks that we are capable of automating and delegating to computers has also grown steadily. We are gaining a progressively better understanding of how to engineer computer systems to deal with tasks that would have been unthinkable only a short time ago.

Copyrighted material

DELEGATION

The next trend is towards ever-increasing delegation. For example, we routinely delegate to computer systems such safety-critical tasks as piloting aircraft. Indeed, in fly-by-wire aircraft, the judgement of a computer program is trusted over that of experienced pilots. Delegation implies that we *give control* to computer systems.

HUMAN-ORIENTATION

The fifth and final trend is the steady move away from machine-oriented views of human–computer interaction towards concepts and metaphors that more closely reflect the way in which we ourselves understand the world. This trend is evident in every way that we interact with computers. For example, in the earliest days of computers, a user interacted with a computer by setting switches on the machine. The internal operation of the device was in no way hidden from the user – in order to use it successfully, one had to fully understand the internal structure and operation of the device. Such primitive – and unproductive – interfaces gave way to command-line interfaces, where one could interact with the device in terms of an ongoing dialogue, in which the user issued instructions that were then executed. Such interfaces dominated until the 1980s, when they gave way to graphical user interfaces, and the direct manipulation paradigm in which a user controls the device by directly manipulating graphical icons corresponding to objects such as files and programs (the 'desktop' metaphor). Similarly, in the earliest days of computing, programmers had no choice but to program their computers in terms of raw machine code, which implied a detailed understanding of the internal structure and operation of their machines. Subsequent programming paradigms have progressed away from such low-level views: witness the development of assembler languages, through procedural abstraction, to abstract data types, and most recently, objects. Each of these developments has allowed programmers to conceptualize and implement software in terms of higher-level – more human-oriented – abstractions.

GLOBAL COMPUTING

These trends present major challenges for software developers. With respect to ubiquity and interconnection, we do not yet know what techniques might be used to develop systems to exploit ubiquitous processor power. Current software development models have proved woefully inadequate even when dealing with relatively small numbers of processors. What techniques might be needed to deal with systems composed of $10^{10}$ processors? The term *global computing* has been coined to describe such unimaginably large systems.

The trends to increasing delegation and intelligence imply the need to build computer systems that can act effectively on our behalf. This in turn implies two capabilities. The first is the ability of systems to operate *independently*, without our direct intervention. The second is the need for computer systems to be able to act in such a way as to *represent our best interests* while interacting with other humans or systems.

The trend towards interconnection and distribution has, in mainstream computer science, long been recognized as a key challenge, and much of the intellectual energy of the field throughout the past three decades has been directed towards developing software tools and mechanisms that allow us to build distributed systems with greater ease and reliability. However, when coupled with the need for systems that can represent our best interests, distribution poses other fundamental problems. When a computer system acting on our behalf must interact with another computer system that represents the interests of another, it may well be (indeed, it is likely) that these interests are not the same. It becomes necessary to endow such systems with the ability to *cooperate* and *reach agreements* with

Copyrighted material

out what needs to be done and how to do it. Finally, the probe must actually do the actions it has chosen, and must presumably monitor what happens in order to ensure that all goes well. If more things go wrong, the probe will be required to recognize this and respond appropriately. Notice that this is the kind of behaviour that we (humans) find easy: we do it every day – when we miss a flight or have a flat tyre while driving to work. But, as we shall see, it is *very* hard to design computer programs that exhibit this kind of behaviour.

A key air-traffic control system at the main airport of Ruritania suddenly fails, leaving flights in the vicinity of the airport with no air-traffic control support. Fortunately, autonomous air-traffic control systems in nearby airports recognize the failure of their peer, and cooperate to track and deal with all affected flights. The potentially disastrous situation passes without incident.

There are several important issues in this scenario. The first is the ability of systems to take the initiative when circumstances dictate. The second is the ability of agents to cooperate to solve problems that are beyond the capabilities of any individual agent. The kind of cooperation required by this scenario was studied extensively in the Distributed Vehicle Monitoring Testbed (DVMT) project undertaken between 1981 and 1991 (see, for example, [Durfee, 1988]). The DVMT simulates a network of vehicle monitoring agents, where each agent is a problem solver that analyses sensed data in order to identify, locate, and track vehicles moving through space. Each agent is typically associated with a sensor, which has only a partial view of the entire space. The agents must therefore cooperate in order to track the progress of vehicles through the entire sensed space. Air-traffic control systems have been a standard application of agent research since the work of Cammarata and colleagues in the early 1980s [Cammarata et al., 1983]; an example of a multiagent air-traffic control application is the OASIS system implemented for use at Sydney airport in Australia [Ljungberg and Lucas, 1992].

Well, most of us are not involved in either designing control systems for space probes or the design of safety-critical systems such as air-traffic controllers. So let us now consider a vision that is closer to most of our everyday lives.

After the wettest and coldest UK winter on record, you are in desperate need of a last-minute holiday somewhere warm and dry. After specifying your requirements to your personal digital assistant (PDA), it converses with a number of different websites, which sell services such as flights, hotel rooms, and hire cars. After hard negotiation on your behalf with a range of sites, your PDA presents you with a package holiday.

This example is perhaps the closest of the three scenarios to actually being realized. There are many websites that will allow you to search for last-minute holidays, but at the time of writing, to the best of my knowledge, none of them engages in active real-time negotiation in order to assemble a package specifically for you from a range of service providers. There are many basic research problems that need to be solved in order to make such a scenario work, such as the examples that follow.

Copyrighted material

**Autonomous Systems in Space**

Space exploration is proving to be an important application area for autonomous systems. Current unmanned space missions typically require a ground crew of up to 300 staff to continuously monitor flight progress. This ground crew usually makes all the necessary control decisions on behalf of the space probe, and painstakingly transmits these decisions to the probe, where they are then blindly executed. Given the length of typical planetary exploration missions, this procedure is expensive and, if decisions are ever required *quickly*, it is simply not practical. Moreover, in some circumstances, it isn't possible at all. For example, in the first serious feasibility study on interstellar travel, [Bond, 1978] notes that an extremely high degree of autonomy would be required on the proposed mission to Barnard's star, lasting more than 50 years:

> [The control system] must be capable of reacting autonomously in the best way possible to a set of circumstances which is indeterminate at launch. . . . Goals may be implanted in the [system] prior to flight, but rigid seeking of those goals may result in total mission failure; those implanted goals may have to be expanded, contracted, or superseded in the light of unanticipated circumstances.
>
> [Bond, 1978, p. S131]

An extremely clear description of the type of system that this book is all about, written in 1978! Sadly, interstellar travel of the type discussed in [Bond, 1978] is a long way off, if it is ever possible at all. But the idea of autonomy in space flight remains very relevant for real space missions today. Launched from Cape Canaveral on 24 October 1998, NASA's DS1 was the first space probe to have an autonomous, agent-based control system [Muscettola et al., 1998]. The autonomous control system in DS1 was capable of making many important decisions itself. This made the mission more robust, particularly against sudden unexpected problems, and also had the very desirable side effect of reducing overall mission costs. NASA (and other space agencies) are currently looking beyond the autonomy of individual space probes, to having teams of probes cooperate in space exploration missions [Jonsson et al., 2007].

- How do you state your preferences to your agent?
- How can your agent compare different deals from different vendors?
- What algorithms can your agent use to negotiate with other agents (so as to ensure that you are not 'ripped off')?

The ability to negotiate in the style implied by this scenario is potentially very valuable indeed. Every year, for example, the European Commission puts out thousands of contracts to public tender. The bureaucracy associated with managing this process has an enormous cost. The ability to automate the tendering and negotiation process would save enormous sums of money (*taxpayers'* money!). Similar situations arise in government organizations everywhere – a good example is the US military. So the ability to automate the process of

Copyrighted material

software agents reaching mutually acceptable agreements on matters of common interest is not just an abstract concern – it may affect our lives (the amount of tax we pay) in a significant way.

## 1.2 Some Views of the Field

The multiagent systems field is highly interdisciplinary: it takes inspiration from such diverse areas as economics, philosophy, logic, ecology, and the social sciences. It should come as no surprise that there are therefore many different views of what the 'multiagent systems project' is all about.

### 1.2.1 Agents as a paradigm for software engineering

Software engineers have derived a progressively better understanding of the characteristics of complexity in software. It is now widely recognized that interaction is probably the most important single characteristic of complex software. Software architectures that contain many dynamically interacting components, each with their own thread of control, and engaging in complex, coordinated protocols, are typically orders of magnitude more complex to engineer correctly and efficiently than those that simply compute a function of some input through a single thread of control. Unfortunately, it turns out that many (if not most) real-world applications have precisely these characteristics. As a consequence, a major research topic in computer science over at least the past three decades has been the development of tools and techniques to model, understand, and implement systems in which interaction is the norm. Indeed, many researchers now believe that, in the future, computation itself will be understood chiefly as a process of interaction. Just as we can understand many systems as being composed of essentially passive objects, which have a state, and upon which we can perform operations, so we can understand many others as being made up of interacting, semi-autonomous agents. This recognition has led to the growth of interest in agents as a new paradigm for software engineering.

As I noted at the start of this chapter, the trend in computing has been – and will continue to be – towards ever more ubiquitous, interconnected computer systems. The development of software paradigms that are capable of exploiting the potential of such systems is perhaps the greatest challenge in computing at the start of the 21st century. Agents seem a strong candidate for such a paradigm.

It is worth noting that many researchers from other areas of computer science have similar goals to those of the multiagent systems community.

#### Self-interested computation

First, there has been a dramatic increase of interest in the study and application of *economic mechanisms* in computer science. For example, auctions are a well-known type of economic mechanism, used for resource allocation, which have achieved particular prominence in computer science [Cramton et al., 2006; Krishna, 2002]. There are a number of reasons for this rapid growth of interest, but perhaps most fundamentally, it is increasingly recognized

Copyrighted material

that a truly deep understanding of many (perhaps most) distributed and networked systems can only come after acknowledging that they have the characteristics of economic systems, in the following sense. Consider an online auction system, such as eBay [eBay, 2001]. At one level of analysis, this is simply a distributed system: it consists of various nodes, which interact with one another by exchanging data, according to some protocols. Distributed systems have been very widely studied in computer science, and we have a variety of techniques for engineering and analysing them [Ben-Ari, 1990]. However, while this analysis is of course legitimate, and no doubt important, it is surely missing a big and very important part of the picture. The participants in such online auctions are *self interested*. They are acting in the system *strategically*, in order to obtain the best outcome for themselves that they can. For example, the seller is typically trying to maximize selling price, while the buyer is trying to minimize it. Thus, if we only think of such a system as a distributed system, then our ability to predict and understand its behaviour is going to be rather limited. We also need to understand it from an *economic* perspective. In the area of multiagent systems, we take these considerations one stage further, and start to think about the issues that arise when the participants in the system *are themselves computer programs*, acting on behalf of their users or owners.

## The Grid

THE GRID

The long-term vision of the *Grid* involves the development of large-scale open distributed systems, capable of being able to effectively and dynamically deploy and redeploy computational (and other) resources as required, to solve computationally complex problems [Foster and Kesselman, 1999]. To date, research in the architecture of the Grid has focused largely on the development of a software *middleware* with which complex distributed systems (often characterized by large datasets and heavy processing requirements) can be engineered. Comparatively little effort has been devoted to *cooperative problem solving* in the Grid. But issues such as cooperative problem solving are exactly those studied by the multiagent systems community:

MIDDLEWARE

COOPERATIVE PROBLEM SOLVING

> The Grid and agent communities are both pursuing the development of such open distributed systems, albeit from different perspectives. The Grid community has historically focussed on . . . 'brawn': interoperable infrastructure and tools for secure and reliable resource sharing within dynamic and geographically distributed virtual organisations (VOs) [Foster et al., 2001], and applications of the same to various resource federation scenarios. In contrast, those working on agents have focussed on 'brains', i.e. on the development of concepts, methodologies and algorithms for autonomous problem solvers that can act flexibly in uncertain and dynamic environments in order to achieve their objectives.
>
> [Foster et al., 2004]

## Ubiquitous computing

The vision of *ubiquitous computing* is as follows:

> [P]opulations of computing entities – hardware and software – will become an effective part of our environment, performing tasks that support our broad purposes without

Copyrighted material

our continual direction, thus allowing us to be largely unaware of them. The vision arises because the technology begins to lie within our grasp. This tangle of concerns, about future systems of which we have only hazy ideas, will define a new character for computer science over the next half-century. What sense can we make of the tangle, from our present knowledge?

[Milner, 2006]

This vision is clearly connected with the trends that we discussed at the opening of this chapter, and makes obvious reference to cooperation and autonomy. We might expect that the ubiquitous computing and multiagent systems communities will have something to say to one another in the years ahead.

### The semantic web

Tim Berners-Lee, inventor of the worldwide web, suggested that the lack of *semantic markup* on the current worldwide web hinders the ability of computer programs to usefully process information available on web pages. The 'markup' (HTML tags) used on current web pages only provides information about the layout and presentation of the web page. While this information can be used by a program to present a page, these tags give no indication of the *meaning* of the information on the page. This led Berners-Lee to propose the idea of the *Semantic Web*:

SEMANTIC MARKUP

I have a dream for the Web [in which computers] become capable of analysing all the data on the Web – the content, links, and transactions between people and computers. A 'Semantic Web', which should make this possible, has yet to emerge, but when it does, the day-to-day mechanisms of trade, bureaucracy and our daily lives will be handled by machines talking to machines. The 'intelligent agents' [that] people have touted for ages will finally materialise.

[Berners-Lee, 1999, pp. 169–170]

The semantic web vision thus explicitly proposes the use of agents. In later chapters, we will see how the kinds of technologies developed within the semantic web community have been used within multiagent systems.

### Autonomic computing

*Autonomic computing* is described as:

AUTONOMIC COMPUTING

[T]he ability to manage your computing enterprise through hardware and software that automatically and dynamically responds to the requirements of your business. This means self-healing, self-configuring, self-optimising, and self-protecting hardware and software that behaves in accordance to defined service levels and policies. Just like the nervous system responds to the needs of the body, the autonomic computing system responds to the needs of the business.

[Murch, 2004]

Systems that can heal themselves and adapt autonomously to changing circumstances clearly have the character of agent systems.

Copyrighted material

- Classical AI has largely ignored the *social* aspects of agency. I hope you will agree that part of what makes us unique as a species on Earth is not simply our undoubted ability to learn and solve problems, but our ability to communicate, cooperate, and reach agreements with our peers. These kinds of social ability – which we use every day of our lives – are surely just as important to intelligent behaviour as are components of intelligence such as planning and learning, and yet they were not studied in AI until about 1980.

**Isn't it all just economics/game theory?**

Game theory is a mathematical theory that studies interactions among self-interested agents [Binmore, 1992]. It is interesting to note that von Neumann, one of the founders of computer science, was also one of the founders of game theory [von Neumann and Morgenstern, 1944]; Alan Turing, arguably the other great figure in the foundations of computing, was also interested in the formal study of games, and it may be that it was this interest that ultimately led him to write his classic paper *Computing Machinery and Intelligence*, which is commonly regarded as the foundation of AI as a discipline [Turing, 1963]. However, after these beginnings, game theory and computer science went their separate ways for some time. Game theory was largely – though by no means solely – the preserve of economists, who were interested in using it to study and understand interactions among economic entities in the real world.

Recently, the tools and techniques of game theory have found many applications in computational multiagent systems research, particularly when applied to problems such as negotiation. Indeed, at the time of writing, game theory seems to be the predominant theoretical tool in use for the analysis of multiagent systems. An obvious question is therefore whether multiagent systems are properly viewed as a subfield of economics/game theory. There are two points here.

- Many of the solution concepts developed in game theory (such as Nash equilibrium, discussed later) were developed without a view to computation. They tend to be *descriptive* concepts, telling us the properties of an appropriate, optimal solution *without* telling us how to compute a solution. Moreover, it turns out that the problem of computing a solution is often computationally very hard (e.g. NP-complete or worse). Multiagent systems research highlights these problems, and allows us to bring the tools of computer science (e.g. computational complexity theory [Garey and Johnson, 1979; Papadimitriou, 1994]) to bear on them.

- Some researchers question the assumptions that game theory makes in order to reach its conclusions. In particular, debate has arisen in the multiagent systems community with respect to whether or not the notion of a rational agent, as modelled in game theory, is valid and/or useful for understanding human or artificial agent societies.

Note that all this should *not* be construed as a criticism of game theory, which is without doubt a valuable and important tool in multiagent systems, likely to become much more widespread in use over the coming years.

Copyrighted material

**Software Agents in Popular Culture**

Software technologies do not seem the most obvious subject matter for novels or films, but autonomous software agents have a starring role surprisingly often. Part of the reason may be that agents are seen as an 'embodiment' of the artificial intelligence dream, which has long been a subject for story makers. The computer Hal, in the film *2001: A Space Odyssey* is the best-known example. However, the kinds of issues addressed in this book have also made other appearances in film and fiction. One of the earliest mentions that I am aware of was in Douglas Adams' novel *Mostly Harmless*, where he imagines software agents cooperating to try to control a damaged spacecraft:

> Small modules of software – agents – surged through the logical pathways, grouping, consulting, re-grouping. They quickly established that the ship's memory, all the way back to its central mission module, was in tatters.

Some authors like to play on the fact that the word 'agent' has multiple meanings: in the Wachowski brothers' *Matrix* trilogy of films, the character Neo must do battle in a virtual world with 'agents' that are clearly intended to be of both the autonomous software and the secret variety.

Michael Crichton's novel *Prey* is based on the premise of agents, embodied in nano-machines, going (badly!) wrong. He clearly did some research about multiagent systems:

> Basically, you can think of a multiagent environment as something like a chessboard, the agents like chess pieces. The agents interact . . . to achieve a goal. . . . The difference is that nobody is moving the agents. They interact on their own to produce the outcome.

Finally, the main character of the David Lodge novel *Thinks* is an artificial intelligence researcher, who has an affair with a student, who subsequently blackmails him to publish her scientific paper – entitled 'Modelling Learning Behaviours in Autonomous Agents'! I am happy to report that, in my experience at least, this kind of behaviour really is limited to fiction.

**Isn't it all just social science?**

The social sciences are primarily concerned with understanding the behaviour of human societies. Some social scientists are interested in (computational) multiagent systems because they provide an experimental tool with which to model human societies. In addition, an obvious approach to the design of multiagent systems – which are artificial societies – is to look at how human societies function, and try to build the multiagent system in the same way. (An analogy may be drawn here with the methodology of AI, where it is quite common to study how humans achieve a particular kind of intelligent capability, and then to attempt to model this in a computer program.) Is the multiagent systems field therefore simply a subset of the social sciences?

Copyrighted material

Although we can usefully draw insights and analogies from human societies, it does not follow that we should build artificial societies in the same way. It is notoriously hard to model precisely the behaviour of human societies, simply because they are dependent on so many different parameters. Moreover, although it is perfectly legitimate to design a multiagent system by drawing upon and making use of analogies and metaphors from human societies, it does not follow that this is going to be the *best* way to design a multiagent system: there are other tools that we can use equally well (such as game theory – see above).

It seems to me that multiagent systems and the social sciences have a lot to say to each other. Multiagent systems provide a powerful and novel tool for modelling and understanding societies, while the social sciences represent a rich repository of concepts for understanding and building multiagent systems – but they are quite distinct disciplines.

## Notes and Further Reading

There are now many introductions to intelligent agents and multiagent systems. [Ferber, 1999] is an undergraduate textbook, although it was written in the early 1990s, and so (for example) does not mention any issues associated with the Web. A first-rate collection of articles introducing agent and multiagent systems is [Weiß, 1999]. Many of its articles address issues in much more depth than is possible in this book. I would certainly recommend this volume for anyone with a serious interest in agents, and it would make an excellent companion to the present volume for more detailed reading.

Three collections of research articles provide a comprehensive introduction to the field of autonomous rational agents and multiagent systems: Bond and Gasser's 1988 collection, *Readings in Distributed Artificial Intelligence*, introduces almost all the basic problems in the multiagent systems field, and although some of the papers it contains are now rather dated, it remains essential reading [Bond and Gasser, 1988]; Huhns and Singh's more recent collection sets itself the ambitious goal of providing a survey of the whole of the agent field, and succeeds in this respect very well [Huhns and Singh, 1998]. Finally, [Bradshaw, 1997] is a collection of papers on software agents.

For a general introduction to the theory and practice of intelligent agents, see [Wooldridge and Jennings, 1995], which focuses primarily on the theory of agents, but also contains an extensive review of agent architectures and programming languages. A short but thorough roadmap of agent technology was published as [Jennings et al., 1998].

**Class reading: introduction to [Bond and Gasser, 1988].** This article is probably the best survey of the problems and issues associated with multiagent systems research yet published. Most of the issues it addresses are fundamentally still open, and it therefore makes a useful preliminary to the current volume. It may be worth revisiting when the course is complete.

Copyrighted material

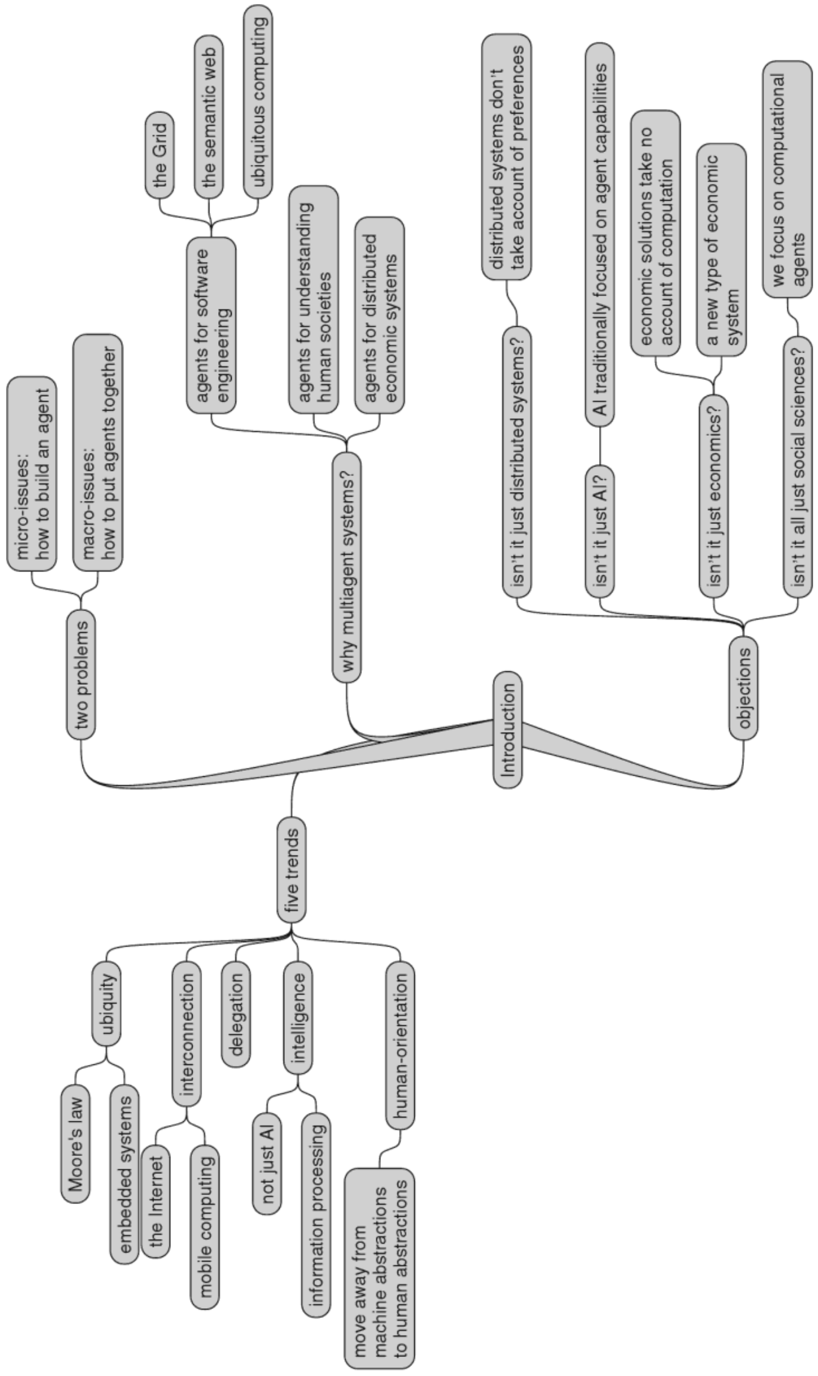
Introduction
two problems
micro-issues: how to build an agent
macro-issues: how to put agents together
why multiagent systems?
agents for software engineering
the Grid
the semantic web
ubiquitous computing
agents for understanding human societies
agents for distributed economic systems
objections
isn't it just distributed systems?
distributed systems don't take account of preferences
isn't it just AI?
AI traditionally focused on agent capabilities
isn't it just economics?
economic solutions take no account of computation
a new type of economic system
isn't it all just social sciences?
we focus on computational agents
five trends
ubiquity
Moore's law
embedded systems
interconnection
the Internet
mobile computing
delegation
intelligence
not just AI
information processing
human-orientation
move away from machine abstractions to human abstractions

Figure 1.1: Mind map for this chapter.

Copyrighted material

Copyrighted material

Chapter 2

# Intelligent Agents

The aim of this chapter is to give you an understanding of what agents are, and some of the issues associated with building them. In later chapters, we will see specific approaches to building agents.

An obvious way to open this chapter would be by presenting a definition of the term *agent*. After all, this is a book about multiagent systems – surely we must all agree on what an agent is? Sadly, there is no universally accepted definition of the term agent, and indeed there is much ongoing debate and controversy on this very subject. Essentially, while there is a general consensus that *autonomy* is central to the notion of agency, there is little agreement beyond this. Part of the difficulty is that various attributes associated with agency are of differing importance for different domains. Thus, for some applications, the ability of agents to *learn* from their experiences is of paramount importance; for other applications, learning is not only unimportant, it is undesirable.[1]

AUTONOMY

Nevertheless, some sort of definition is important – otherwise, there is a danger that the term will lose all meaning. The definition presented here is adapted from [Wooldridge and Jennings, 1995].

> An *agent* is a computer system that is *situated* in some *environment*, and that is capable of *autonomous action* in this environment in order to meet its delegated objectives.

SITUATED AGENT

AUTONOMOUS ACTION

Figure 2.1 gives an abstract view of an agent. In this diagram, we can see the action output generated by the agent in order to affect its environment. In most domains of reasonable complexity, an agent will not have *complete* control over its environment. It will have at best *partial* control, in that it can *influence* it. From the point of view of the agent, this

[1]Michael Georgeff, the main architect of the PRS agent system discussed in later chapters, gives the example of an air-traffic control system he developed; the clients of the system would have been horrified at the prospect of such a system modifying its behaviour at run-time.

Copyrighted material

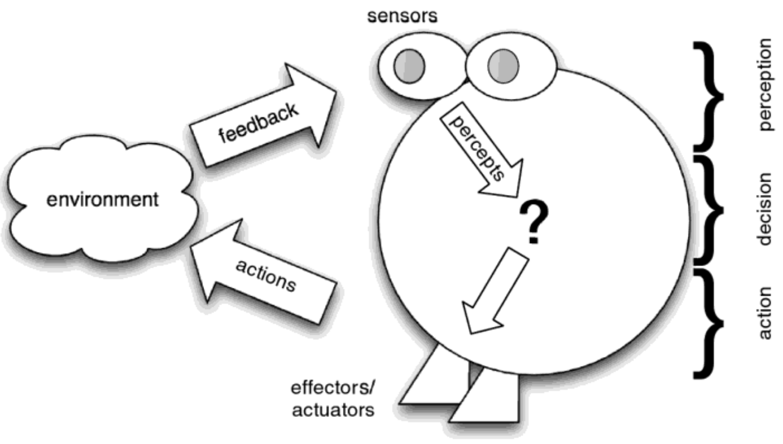

Figure 2.1: An agent in its environment (after [Russell and Norvig, 1995, p. 32]). The agent takes sensory input from the environment, and produces, as output, actions that affect it. The interaction is usually an ongoing, non-terminating one.

means that the same action performed twice in apparently identical circumstances might appear to have entirely different effects, and in particular it may *fail* to have the desired effect. Thus agents in all but the most trivial of environments must be prepared for the possibility of *failure*.

**Varieties of autonomy**

We have been casually using the term 'autonomy' without digging too deeply into what this means. Unfortunately, 'autonomy' is a very loaded word: it conveys a number of different meanings to different people. For our purposes, we can understand autonomy as follows.

The first thing to say is that autonomy is a *spectrum*. At one extreme on this spectrum are fully realized human beings, like you and me. We have as much freedom as anybody does with respect to our beliefs, our goals, and our actions. Of course, we do not have *complete* freedom over beliefs, goals, and actions. For example, I do not believe I could choose to believe that $2 + 2 = 5$; nor could I choose to want to suffer pain. Our genetic makeup, our upbringing, and indeed society itself have effectively conditioned us to restrict our possible choices. For example, millions of years of evolution have conditioned my animal nature to prevent me wanting to suffer pain. There are of course cases where individuals go outside these bounds, and often society may forcibly restrict the behaviour of such individuals, for their own good and that of society itself. Nevertheless, a human in the free world is just about as autonomous as it gets: we can choose our beliefs, our goals, and our actions.

At the other end of the autonomy spectrum, we might think of a software service (such as a public method provided by a Java object). Such services are not autonomous in any meaningful sense: they simply do what they are told. Similarly, applications such as word

Copyrighted material

processors are not usefully thought of as being autonomous – for the most part, they do things only because we tell them to.

However, there is a range of points between these two extremes. For example, we can think of a system that acts in some environment under remote control, through remote supervision (where we monitor the behaviour of an entity, and can intervene if necessary, but otherwise the entity acts under its own direction). The point on the autonomy spectrum that we will largely be interested in is an entity to which we can *delegate* goals in some high-level way (i.e. not just by giving it a fully elaborated program to execute), and then have this entity decide for itself how best to accomplish its goals. The entity here is not quite autonomous in the sense that you and I are: it cannot simply choose what goals to accomplish, except inasmuch as such 'subgoals' are in the furtherance of our delegated goals. Thus 'autonomy', in the sense that we are interested in it, means the ability and requirement to decide how to act so as to accomplish our delegated goals.

Sometimes, we may be cautious about unleashing an agent on the world. We may want to build in some degree of limited autonomy, or more generally, equip the agent with some type of *adjustable autonomy* [Scerri et al., 2003]. The basic idea with adjustable autonomy is that control of decision-making is transferred from the agent to a person whenever certain conditions are met, for example [Scerri et al., 2003, p. 211]:

ADJUSTABLE AUTONOMY

- when the agent believes that the human will make a decision with a substantially higher benefit
- when there is a high degree of uncertainty about the environment
- when the decision might cause harm, or
- when the agent lacks the capability to make the decision itself.

Of course, there is a difficult balance to be struck: an agent that *always* comes back to its user or owner for help with decisions will be unhelpful, while one that *never* seeks assistance will probably also be useless.

## Decisions and actions

Normally, an agent will have a repertoire of actions available to it. This set of possible actions represents the agent's ability to modify its environments. Note that not all actions can be performed in all situations. For example, an action 'lift table' is only applicable in situations where the weight of the table is sufficiently small that the agent *can* lift it. Actions therefore have *preconditions* associated with them, which define the possible situations in which they can be applied.

The key problem facing an agent is that of deciding *which* of its actions it should perform in order to best satisfy its design objectives. *Agent architectures*, of which we shall see many examples later in this book, are really software architectures for decision-making systems that are embedded in an environment. At this point, it is worth pausing to consider some examples of agents (though not, as yet, intelligent agents).

AGENT ARCHITECTURES

Copyrighted material

### Control systems

CONTROL SYSTEMS

First, any *control* system can be viewed as an agent. A simple (and overused) example of such a system is a thermostat. Thermostats have a sensor for detecting room temperature. This sensor is directly embedded within the environment (i.e. the room), and it produces as output one of two signals: one that indicates that the temperature is too low, and another which indicates that the temperature is OK. The actions available to the thermostat are 'heating on' or 'heating off'. The action 'heating on' will generally have the effect of raising the room temperature, but this cannot be a *guaranteed* effect – if the door to the room is open, for example, switching on the heater may have no effect. The (extremely simple) decision-making component of the thermostat implements (often in electromechanical hardware) the following rules:

$$\begin{aligned}\text{too cold} &\longrightarrow \text{heating on},\\ \text{temperature OK} &\longrightarrow \text{heating off}.\end{aligned}$$

More complex environment control systems, of course, have considerably richer decision structures. Examples include autonomous space probes, fly-by-wire aircraft, nuclear reactor control systems, and so on.

### Software demons

Second, most software demons (such as background processes in the Unix operating system), which monitor a software environment and perform actions to modify it, can be viewed as agents. An example is the X Windows program `xbiff`. This utility continually monitors a user's incoming email, and indicates via a GUI icon whether or not they have unread messages. Whereas our thermostat agent in the previous example inhabited a *physical* environment – the physical world – the `xbiff` program inhabits a *software* environment. It obtains information about this environment by carrying out software functions (by executing system programs such as `ls`, for example), and the actions it performs are software actions (changing an icon on the screen, or executing a program). The decision-making component is just as simple as our thermostat example.

PHYSICALLY EMBODIED AGENT

SOFTWARE AGENT

To summarize, agents are simply computer systems that are capable of autonomous action in some environment in order to meet their design objectives. An agent will typically sense its environment (by physical sensors in the case of agents situated in part of the real world, or by software sensors in the case of software agents), and will have available a repertoire of actions that can be executed to modify the environment, which may appear to respond non-deterministically to the execution of these actions.

### Reactive and functional systems

FUNCTIONAL SYSTEM

Originally, software engineering concerned itself with what are known as 'functional' systems. A functional system is one that simply takes some input, performs some computation over this input, and eventually produces some output. Such systems may formally be viewed as functions $f : I \to O$ from a set $I$ of inputs to a set $O$ of outputs. The classic example

Copyrighted material

**Environments**

It is worth pausing at this point to discuss the general properties of the environments that agents may find themselves in. Russell and Norvig suggest the following classification of environment properties [Russell and Norvig, 1995, p. 46].

**Accessible versus inaccessible** An accessible environment is one in which the agent can obtain complete, accurate, up-to-date information about the environment's state. Most real-world environments (including, for example, the everyday physical world and the Internet) are not accessible in this sense.

**Deterministic versus non-deterministic** A deterministic environment is one in which any action has a single guaranteed effect – there is no uncertainty about the state that will result from performing an action.

**Static versus dynamic** A *static* environment is one that can be assumed to remain unchanged except by the performance of actions by the agent. In contrast, a *dynamic* environment is one that has other processes operating on it, and which hence changes in ways beyond the agent's control. The physical world is a highly dynamic environment, as is the Internet.

**Discrete versus continuous** An environment is discrete if there are a fixed, finite number of actions and percepts in it.

In general, the most complex kind of environment is one that is inaccessible, non-deterministic, dynamic, and continuous.

of such a system is a compiler, which can be viewed as a mapping from a set $I$ of legal source programs to a set $O$ of corresponding object or machine-code programs.

Although the internal complexity of a functional system may be great (e.g. in the case of a compiler for a complex programming language such as Ada), functional programs are, in general, regarded as comparatively simple from the standpoint of software development. Unfortunately, many computer systems that we desire to build are not functional in this sense. Rather than simply computing a function of some input and then terminating, many computer systems are *reactive*, in the following sense: REACTIVE SYSTEM

> 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 ongoing behaviour ... [E]very concurrent system ... must be studied by behavioural 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.
>
> [Pnueli, 1986]

Reactive systems are harder to engineer than functional ones. Perhaps the most important reason for this is that an agent engaging in a (conceptually) non-terminating

Copyrighted material

our agent to be able to react to the new situation, in time for the reaction to be of some use. However, we do not want our agent to be *continually* reacting, and hence never focusing on a goal long enough to actually achieve it.

On reflection, it should come as little surprise that achieving a good balance between goal-directed and reactive behaviour is hard. After all, it is comparatively rare to find humans that do this very well. This problem – of effectively integrating goal-directed and reactive behaviour – is one of the key problems facing the agent designer. As we shall see, a great many proposals have been made for how to build agents that can do this – but the problem is essentially still open.

Finally, let us say something about *social ability*, the final component of flexible autonomous action as defined here. In one sense, social ability is trivial: every day, millions of computers across the world routinely exchange information with both humans and other computers. But the ability to exchange bit streams is not really social ability. Consider that, in the human world, comparatively few of our meaningful goals can be achieved without the *cooperation* of other people, who cannot be assumed to *share* our goals – in other words, they are themselves autonomous, with their own agenda to pursue. To achieve our goals in such situations, we must *negotiate* and *cooperate* with others. We may be required to understand and reason about the goals of others, and to perform actions (such as paying them money) that we would not otherwise choose to perform, in order to get them to cooperate with us, and achieve our goals. This type of social ability is much more complex, and much less well understood than simply the ability to exchange binary information. Social ability in general (and topics such as negotiation and cooperation in particular) are dealt with elsewhere in this book, and will not therefore be considered here. In this chapter, we will be concerned with the decision-making of *individual* intelligent agents in environments which may be dynamic, unpredictable, and uncertain, but do not contain other agents.

## 2.2 Agents and Objects

Programmers familiar with object-oriented languages such as Java, C++, or Smalltalk sometimes fail to see anything novel in the idea of agents. When one stops to consider the relative properties of agents and objects, this is perhaps not surprising.

> There is a tendency . . . to think of objects as 'actors' and endow them with human-like intentions and abilities. It's tempting to think about objects 'deciding' what to do about a situation, [and] 'asking' other objects for information. . . . Objects are not passive containers for state and behaviour, but are said to be the agents of a program's activity.
>
> [NeXT Computer Inc., 1993, p. 7]

Objects are defined as computational entities that *encapsulate* some state, are able to perform actions, or *methods*, on this state, and communicate by message passing. While there are obvious similarities, there are also significant differences between agents and objects. The first is in the degree to which agents and objects are autonomous. Recall that the defining characteristic of object-oriented programming is the principle of encapsulation – the idea that objects can have control over their own internal state. In programming languages like

Copyrighted material

Java, we can declare instance variables (and methods) to be `private`, meaning that they are only accessible from within the object. (We can of course also declare them `public`, meaning that they can be accessed from anywhere, and indeed we must do this for methods so that they can be used by other objects. But the use of `public` instance variables is usually considered poor programming style.) In this way, an object can be thought of as exhibiting autonomy over its state: it has control over it. But an object does not exhibit control over its *behaviour*. That is, if a method `m` is made available for other objects to invoke, then they can do so whenever they wish – once an object has made a method `public`, then it subsequently has no control over whether or not that method is executed. Of course, an object *must* make methods available to other objects, or else we would be unable to build a system out of them. This is not normally an issue, because if we build a system, then we design the objects that go in it, and they can thus be assumed to share a 'common goal'. But in many types of multiagent system (in particular, those that contain agents built by different organizations or individuals), no such common goal can be assumed. It cannot be taken for granted that an agent $i$ will execute an action (method) $a$ just because another agent $j$ wants it to – $a$ may not be in the best interests of $i$. We thus do not think of agents as invoking methods upon one another, but rather as *requesting* actions to be performed. If $j$ requests $i$ to perform $a$, then $i$ may perform the action or it may not. The locus of control with respect to the decision about whether to execute an action is thus different in agent and object systems. In the object-oriented case, the decision lies with the object that invokes the method. In the agent case, the decision lies with the agent that receives the request. This distinction between objects and agents has been nicely summarized in the following slogan.

---

Objects do it for free; agents do it because they want to.

---

Of course, there is nothing to stop us implementing agents using object-oriented techniques. For example, we can build some kind of decision-making about whether to execute a method into the method itself, and in this way achieve a stronger kind of autonomy for our objects. The point is that autonomy of this kind is not a component of the basic object-oriented model.

The second important distinction between object and agent systems is with respect to the notion of flexible (reactive, proactive, social) autonomous behaviour. The standard object model has nothing whatsoever to say about how to build systems that integrate these types of behaviour. Again, one could object that we can build object-oriented programs that *do* integrate these types of behaviour, but this argument misses the point, which is that the standard object-oriented programming model has nothing to do with these types of behaviour.

The third important distinction between the standard object model and our view of agent systems is that agents are each considered to have their own thread of control – in the standard object model, there is a single thread of control in the system. Of course, a lot of work has recently been devoted to *concurrency* in object-oriented programming. For example, the Java language provides built-in constructs for multithreaded programming. There are also many programming languages available (most of them admittedly prototypes) that were specifically designed to allow concurrent object-based programming. But such

Copyrighted material

languages do not capture the idea of agents as *autonomous* entities. Perhaps the closest that the object-oriented community comes is in the idea of *active objects*.

> An active object is one that encompasses its own thread of control. . . . Active objects are generally autonomous, meaning that they can exhibit some behaviour without being operated upon by another object. Passive objects, on the other hand, can only undergo a state change when explicitly acted upon.
>
> [Booch, 1994, p. 91]

Thus active objects are essentially agents that do not necessarily have the ability to exhibit flexible autonomous behaviour.

To summarize, the traditional view of an object and our view of an agent have at least three distinctions:

- Agents embody a stronger notion of autonomy than objects, and, in particular, they decide for themselves whether or not to perform an action on request from another agent.
- Agents are capable of flexible (reactive, proactive, social) behaviour, and the standard object model has nothing to say about such types of behaviour.
- A multiagent system is inherently multithreaded, in that each agent is assumed to have at least one thread of control.

## 2.3 Agents and Expert Systems

Expert systems were the most important AI technology of the 1980s [Hayes-Roth et al., 1983]. An expert system is one that is capable of solving problems or giving advice in some knowledge-rich domain [Jackson, 1986]. A classic example of an expert system is MYCIN, which was intended to assist physicians in the treatment of blood infections in humans. MYCIN worked by a process of interacting with a user in order to present the system with a number of (symbolically represented) facts, which the system then used to derive some conclusion. MYCIN acted very much as a *consultant*: it did not operate directly on humans, or indeed any other environment. Thus perhaps the most important distinction between agents and expert systems is that expert systems like MYCIN are inherently *disembodied*. By this, I mean that they do not interact directly with any environment: they get their information not via sensors, but through a user acting as middleman. In the same way, they do not *act* on any environment, but rather give feedback or advice to a third party. In addition, expert systems are not generally capable of cooperating with other agents.

In summary, the main differences between agents and expert systems are as follows:

- 'Classic' expert systems are disembodied – they are not coupled to any environment in which they act, but rather act through a user as a 'middleman'.
- Expert systems are not generally capable of reactive, proactive behaviour.
- Expert systems are not generally equipped with social ability, in the sense of cooperation, coordination, and negotiation.

Copyrighted material

Despite these differences, some expert systems (particularly those that perform real-time control tasks) look very much like agents.

## 2.4 Agents as Intentional Systems

One common approach adopted when discussing agent systems is the *intentional stance*. With this approach, we 'endow' agents with *mental states*: beliefs, desires, wishes, hopes, and so on. The rationale for this approach is as follows. When explaining human activity, it is often useful to make statements such as:

INTENTIONAL STANCE

MENTAL STATE

---

Janine took her umbrella because she *believed* it was going to rain.
Michael worked hard because he *wanted* to finish his book.

---

These statements make use of a *folk psychology*, by which human behaviour is predicted and explained through the attribution of *attitudes*, such as believing and wanting (as in the above examples), hoping, fearing, and so on (see, for example, [Stich, 1983, p. 1] for a discussion of folk psychology). This folk psychology is well established: most people reading the above statements would say they found their meaning entirely clear, and would not give them a second glance.

FOLK PSYCHOLOGY

The attitudes employed in such folk psychological descriptions are called the *intentional* notions.[2] The philosopher Daniel Dennett has coined the term *intentional system* to describe entities 'whose behaviour can be predicted by the method of attributing belief, desires and rational acumen' [Dennett, 1987, p. 49]. Dennett identifies different 'levels' of intentional system as follows.

INTENTIONAL SYSTEM

> A *first-order* intentional system has beliefs and desires (etc.) but no beliefs and desires *about* beliefs and desires. . . . A *second-order* intentional system is more sophisticated; it has beliefs and desires (and no doubt other intentional states) about beliefs and desires (and other intentional states) – both those of others and its own.
>
> [Dennett, 1987, p. 243]

One can, of course, carry on this hierarchy of intentionality. A moment's reflection suggests that humans do not use more than about three layers of the intentional stance hierarchy when reasoning in everyday life (unless we are engaged in an artificially constructed intellectual activity, such as solving a puzzle). One interesting aspect of the intentional stance is that it seems to be a key ingredient in the way we *coordinate* our activities with others on a day-by-day basis.

> I call an old friend on the other coast and we agree to meet in Chicago at the entrance of a bar in a certain hotel on a particular day two months hence at 7:45 p.m., and everyone

[2]Unfortunately, the word 'intention' is used in several different ways in logic and the philosophy of mind. First, there is the mentalistic usage, as in 'I intended to kill him'. Second, an intentional notion is one of the attitudes, as above. Finally, in logic, the word intension (with an 's') means the internal content of a concept, as opposed to its extension. In what follows, the intended meaning should always be clear from context.

Copyrighted material

> who knows us predicts that on that day at that time we will meet up. And we do meet up. . . . The calculus behind this forecasting is intuitive psychology: the knowledge that I *want* to meet my friend and vice versa, and that each of us *believes* the other will be at a certain place at a certain time and *knows* a sequence of rides, hikes, and flights that will take us there. No science of mind or brain is likely to do better.
>
> [Pinker, 1997, pp. 63–64]

The intentional stance, intuitively appealing though it is, is not universally accepted within the philosophy of mind research community, and does not seem to sit comfortably with ideas like the *behavioural* view of action. The behavioural view of action (most famously associated with researchers such as B. F. Skinner) tried to give an explanation of human behaviour in terms of learning stimulus-response behaviours, which are produced via 'conditioning' with positive and negative feedback.[3]

> The stimulus-response theory turned out to be wrong. Why did Sally run out of the building? Because she believed it was on fire and did not want to die. . . . What [predicts] Sally's behaviour, and predicts it well, is whether she *believes* herself to be in danger. Sally's beliefs are, of course, related to the stimuli impinging on her, but only in a tortuous, circuitous way, mediated by all the rest of her beliefs about where she is and how the world works.
>
> [Pinker, 1997, pp. 62–63]

Now, we seem to be proposing to use phrases such as belief, desire, and intention to talk about computer programs. An obvious question, therefore, is whether it is legitimate or useful to attribute beliefs, desires, and so on to artificial agents. Is this not just anthropomorphism? McCarthy, among others, has argued that there are occasions when the *intentional stance* is appropriate as follows.

> To ascribe *beliefs*, *free will*, *intentions*, *consciousness*, *abilities*, or *wants* to a machine is legitimate when such an ascription expresses the same information about the machine that it expresses about a person. It is useful when the ascription helps us understand the structure of the machine, its past or future behaviour, or how to repair or improve it. It is perhaps never logically required even for humans, but expressing reasonably briefly what is actually known about the state of the machine in a particular situation may require mental qualities or qualities isomorphic to them. Theories of belief, knowledge, and wanting can be constructed for machines in a simpler setting than for humans, and later applied to humans. Ascription of mental qualities is most straightforward for machines of known structure such as thermostats and computer operating systems, but is most useful when applied to entities whose structure is incompletely known.
>
> [McCarthy, 1978] (The underlining is from [Shoham, 1990].)

What objects can be described by the intentional stance? As it turns out, almost any automaton can. For example, consider a light switch as follows.

[3] Skinner was an interesting character, although I think it is fair to say that many are uncomfortable with his more extreme views on behaviourism. My favourite story about Skinner is that he designed a guided missile controller in which a group of pigeons in the nose cone of a missile would be shown a video feed image of the missile's progress, and would guide the missile by 'pecking their way to the target', so to speak.

Copyrighted material

A *run*, $r$, of an agent in an environment is thus a sequence of interleaved environment states and actions: RUNS

$$r : e_0 \xrightarrow{\alpha_0} e_1 \xrightarrow{\alpha_1} e_2 \xrightarrow{\alpha_2} e_3 \xrightarrow{\alpha_3} \cdots \xrightarrow{\alpha_{u-1}} e_u.$$

Let

- $\mathcal{R}$ be the set of all such possible finite sequences (over $E$ and $Ac$)
- $\mathcal{R}^{Ac}$ be the subset of these that end with an action
- $\mathcal{R}^E$ be the subset of these that end with an environment state.

We will use $r, r', \ldots$ to stand for members of $\mathcal{R}$.

In order to represent the effect that an agent's actions have on an environment, we introduce a *state transformer* function (cf. [Fagin et al., 1995, p. 154]):

$$\tau : \mathcal{R}^{Ac} \to \mathbf{2}^E.$$

Thus a state transformer function maps a run (assumed to end with the action of an agent) to a set of possible environment states – those that could result from performing the action.

There are two important points to note about this definition. First, environments are assumed to be *history dependent*. In other words, the next state of an environment is not solely determined by the action performed by the agent and the current state of the environment. The actions made *earlier* by the agent also play a part in determining the current state. Second, note that this definition allows for *non-determinism* in the environment. There is thus *uncertainty* about the result of performing an action in some state.

If $\tau(r) = \emptyset$ (where $r$ is assumed to end with an action), then there are no possible successor states to $r$. In this case, we say that the system has *ended* its run. We will also assume that all runs eventually terminate.

Formally, we say that an environment $Env$ is a triple $Env = \langle E, e_0, \tau \rangle$, where $E$ is a set of environment states, $e_0 \in E$ is an initial state, and $\tau$ is a state transformer function.

We now need to introduce a model of the agents that inhabit systems. We model agents as functions which map runs (assumed to end with an environment state) to actions (cf. [Russell and Subramanian, 1995, pp. 580, 581]):

$$Ag : \mathcal{R}^E \to Ac.$$

Thus an agent makes a decision about what action to perform based on the history of the system that it has witnessed to date.

Notice that while environments are implicitly non-deterministic, agents are assumed to be deterministic. Let $\mathcal{AG}$ be the set of all agents.

We say a *system* is a pair containing an agent and an environment. Any system will have associated with it a set of possible runs; we denote the set of runs of agent $Ag$ in environment $Env$ by $\mathcal{R}(Ag, Env)$. For simplicity, we will assume that $\mathcal{R}(Ag, Env)$ contains only *terminated* runs, i.e. runs $r$ such that $r$ has no possible successor states: $\tau(r) = \emptyset$. (We will thus not consider infinite runs for now.)

Copyrighted material

Formally, a sequence

$$(e_0, \alpha_0, e_1, \alpha_1, e_2, \ldots)$$

represents a run of an agent $Ag$ in environment $Env = \langle E, e_0, \tau \rangle$ if

1. $e_0$ is the initial state of $Env$;
2. $\alpha_0 = Ag(e_0)$; and
3. for all $u > 0$,

   $$e_u \in \tau((e_0, \alpha_0, \ldots, \alpha_{u-1})),$$

   and

   $$\alpha_u = Ag((e_0, \alpha_0, \ldots, e_u)).$$

Two agents $Ag_1$ and $Ag_2$ are said to be *behaviourally equivalent* with respect to environment $Env$ if and only if $\mathcal{R}(Ag_1, Env) = \mathcal{R}(Ag_2, Env)$, and simply behaviourally equivalent if and only if they are behaviourally equivalent with respect to all environments.

Notice that, so far, I have said nothing at all about how agents are actually implemented; we will return to this issue later.

## Purely reactive agents

Certain types of agents decide what to do without reference to their history. They base their decision-making entirely on the present, with no reference at all to the past. We will call such agents *purely reactive*, since they simply respond directly to their environment. (Sometimes they are called *tropistic* agents [Genesereth and Nilsson, 1987]: tropism is the tendency of plants or animals to react to certain stimulae.)

PURELY REACTIVE AGENT
TROPISTIC AGENTS

Formally, the behaviour of a purely reactive agent can be represented by a function

$$Ag : E \to Ac.$$

It should be easy to see that, for every purely reactive agent, there is an equivalent 'standard' agent, as discussed above; the reverse, however, is not generally the case.

Our thermostat agent is an example of a purely reactive agent. Assume, without loss of generality, that the thermostat's environment can be in one of two states – either too cold, or temperature OK. Then the thermostat is simply defined as follows:

$$Ag(e) = \begin{cases} \text{heater off} & \text{if } e = \text{temperature OK}, \\ \text{heater on} & \text{otherwise.} \end{cases}$$

Copyrighted material

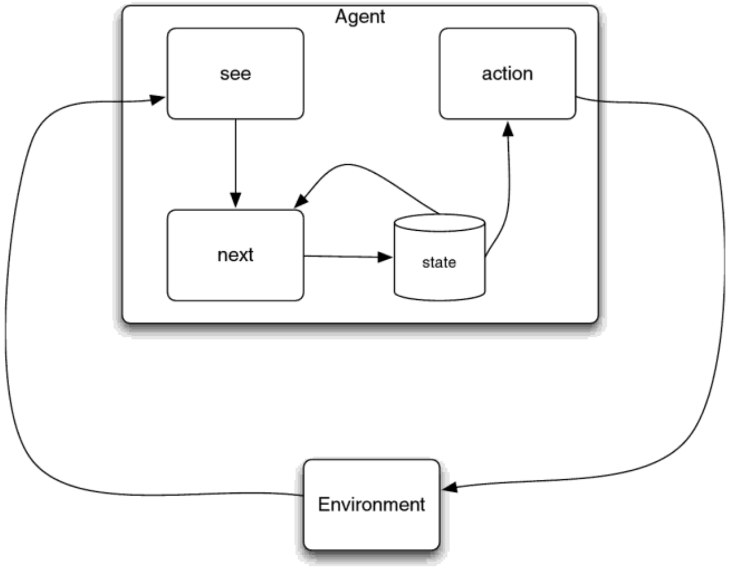

Figure 2.2: An agent that maintains state.

### Agents with state

Viewing agents at this abstract level makes for a pleasantly simple analysis. However, it does not help us to construct them. For this reason, we will now *refine* our abstract model of agents, by breaking it down into subsystems in exactly the way that one does in standard software engineering. As we refine our view of agents, we find ourselves making *design choices* that mostly relate to the subsystems that go to make up an agent – what data and control structures will be present. An *agent architecture* is essentially a map of the internals of an agent – its data structures, the operations that may be performed on these data structures, and the control flow between these data structures. Later in this book, we will discuss a number of different types of agent architecture, with very different views on the data structures and algorithms that will be present within an agent. For the purposes of this chapter, we will ignore the *content* of an agent's state, and simply consider the overall role of state in an agent's decision-making loop – see Figure 2.2.

Thus agents have some internal data structure, which is typically used to record information about the environment state and history. Let $I$ be the set of all internal states of the agent. An agent's decision-making process is then based, at least in part, on this information.

The perception function *see* represents the agent's ability to obtain information from its environment. The *see* function might be implemented in hardware in the case of an agent situated in the physical world: for example, it might be a video camera or an infrared sensor on a mobile robot. For a software agent, the sensors might be system commands that obtain information about the software environment, such as `ls`, `finger`, or suchlike. The *output*

Copyrighted material

PERCEPTS of the *see* function is a *percept* – 'a perceptual input'. Let $Per$ be a (non-empty) set of percepts. Then *see* is a function

$$see : E \rightarrow Per$$

The action-selection function *action* is defined as a mapping

$$action : I \rightarrow Ac$$

from internal states to actions. An additional function *next* is introduced, which maps an internal state and percept to an internal state:

$$next : I \times Per \rightarrow I.$$

The behaviour of a state-based agent can be summarized in the following way. The agent starts in some initial internal state $i_0$. It then observes its environment state $e$, and generates a percept $see(e)$. The internal state of the agent is then updated via the *next* function, becoming set to $next(i_0, see(e))$. The action selected by the agent is then $action(next(i_0, see(e)))$. This action is then performed, and the agent enters another cycle, perceiving the world via *see*, updating its state via *next*, and choosing an action to perform via *action*.

It is worth observing that state-based agents as defined here are in fact no more powerful than the standard agents we introduced earlier. In fact, they are *identical* in their expressive power – every state-based agent can be transformed into a standard agent that is behaviourally equivalent.

## 2.6 How to Tell an Agent What to Do

TASK SPECIFICATION We do not (usually) build agents for no reason. We build them in order to carry out *tasks* for us. In order to get the agent to do the task, we must somehow communicate the desired task to the agent. This implies that the task to be carried out must be *specified* by us in some way. An obvious question is how to specify these tasks: how to tell the agent what to do. One way to specify the task would be simply to write a program for the agent to execute. The obvious advantage of this approach is that we are left with no uncertainty about what the agent will do. It will do exactly what we told it to, and no more. But the very obvious disadvantage is that we have to think about exactly how the task will be carried out ourselves, and if unforeseen circumstances arise, the agent executing the task will be unable to respond accordingly. So, more usually, we want to *tell our agent what to do without telling it how to do it*. One way of doing this is to define tasks *indirectly*, via some kind of *performance measure*. There are several ways in which such a performance measure can be defined. The first is to associate *utilities* with states of the environment. UTILITY

### Utility functions

A utility is a numeric value representing how 'good' a state is: the higher the utility, the better. The task of the agent is then to bring about states that maximize utility – we do not specify to the agent how this is to be done. In this approach, a task specification would

Copyrighted material

simply be a function

$$u : E \to \mathbb{R}$$

which associates a real value with every environment state. Given such a performance measure, we can then define the overall utility of an agent in some particular environment in several different ways. One (pessimistic) way is to define the utility of the agent as the utility of the *worst* state that might be encountered by the agent; another might be to define the overall utility as the average utility of all states encountered. There is no right or wrong way: the measure depends upon the kind of task you want your agent to carry out.

The main disadvantage of this approach is that it assigns utilities to *local* states; it is difficult to specify a *long-term* view when assigning utilities to individual states. To get around this problem, we can specify a task as a function which assigns a utility not to individual states, but to runs themselves:

$$u : \mathcal{R} \to \mathbb{R}.$$

If we are concerned with agents that must operate independently over long periods of time, then this approach appears more appropriate to our purposes. One well-known example of the use of such a utility function is in the Tileworld [Pollack, 1990]. The Tileworld was proposed primarily as an experimental environment for evaluating agent architectures. It is a simulated two-dimensional grid environment on which there are agents, tiles, obstacles, and holes. An agent can move in four directions, up, down, left, or right, and if it is located next to a tile, it can push it. An obstacle is a group of immovable grid cells: agents are not allowed to travel freely through obstacles. Holes have to be filled up with tiles by the agent. An agent scores points by filling holes with tiles, the aim being to fill as many holes as possible. The Tileworld is an example of a *dynamic* environment: starting in some randomly generated world state, based on parameters set by the experimenter, it changes over time in discrete steps, with the random appearance and disappearance of holes. The experimenter can set a number of Tileworld parameters, including the frequency of appearance and disappearance of tiles, obstacles, and holes; and the choice between hard bounds (instantaneous) or soft bounds (slow decrease in value) for the disappearance of holes. In the Tileworld, holes appear randomly and exist for as long as their *life expectancy*, unless they disappear because of the agent's actions. The interval between the appearance of successive holes is called the *hole gestation time*. The performance of an agent in the Tileworld is measured by running the Tileworld testbed for a predetermined number of time steps, and measuring the number of holes that the agent succeeds in filling. The performance of an agent on some particular run is then defined as

$$u(r) = \frac{\text{number of holes filled in } r}{\text{number of holes that appeared in } r}.$$

This gives a normalized performance measure in the range 0 (the agent did not succeed in filling even one hole) to 1 (the agent succeeded in filling every hole that appeared). Experimental error is eliminated by running the agent in the environment a number of times, and computing the average of the performance.

Despite its simplicity, the Tileworld allows us to examine several important capabilities of agents. Perhaps the most important of these is the ability of an agent to *react* to changes

Copyrighted material

the set of all agents $\mathcal{AG}$ can be implemented on this machine. Again, any agent $Ag$ that required more than the available memory would not run.

Let us write $\mathcal{AG}_m$ to denote the subset of $\mathcal{AG}$ that can be implemented on $m$:

$$\mathcal{AG}_m = \{Ag \mid Ag \in \mathcal{AG} \text{ and } Ag \text{ can be implemented on } m\}.$$

Now, assume we have machine (i.e. computer) $m$, and we wish to place this machine in environment $Env$; the task we wish $m$ to carry out is defined by utility function $u : \mathcal{R} \to \mathbb{R}$. Then we can replace Equation (2.1) with the following, which more precisely defines the properties of the desired agent $Ag_{opt}$:

$$Ag_{opt} = \arg \max_{Ag \in \mathcal{AG}_m} \sum_{r \in \mathcal{R}(Ag,Env)} u(r)P(r \mid Ag, Env). \tag{2.2}$$

The subtle but important change in Equation (2.2) is that we are no longer looking for our agent from the set of all possible agents $\mathcal{AG}$, but from the set $\mathcal{AG}_m$ of agents that can actually be implemented on the machine that we have for the task.

Utility-based approaches to specifying tasks for agents have several disadvantages. The most important of these is that it is very often difficult to derive an appropriate utility function; the Tileworld is a useful environment in which to experiment with agents, but it represents a gross simplification of real-world scenarios. The second is that usually we find it more convenient to talk about tasks in terms of 'goals to be achieved' rather than utilities. This leads us to what I call *predicate* task specifications. PREDICATE TASK SPECIFICATION

## Predicate task specifications

Put simply, a predicate task specification is one where the utility function acts as a *predicate* over runs. Formally, we will say that a utility function $u : \mathcal{R} \to \mathbb{R}$ is a predicate if the range of $u$ is the set $\{0, 1\}$, that is, if $u$ guarantees to assign a run either 1 ('true') or 0 ('false'). A run $r \in \mathcal{R}$ will be considered to satisfy the specification $u$ if $u(r) = 1$, and fails to satisfy the specification otherwise.

We will use $\Psi$ to denote a predicate specification, and write $\Psi(r)$ to indicate that run $r \in \mathcal{R}$ which satisfies $\Psi$. In other words, $\Psi(r)$ is true if and only if $u(r) = 1$. For the moment, we will leave aside the questions of what form a predicate task specification might take.

## Task environments

A *task environment* is defined to be a pair $\langle Env, \Psi \rangle$, where $Env$ is an environment, and

$$\Psi : \mathcal{R} \to \{0, 1\}$$

is a predicate over runs. Let $\mathcal{TE}$ be the set of all task environments. A task environment thus specifies:

- the properties of the system the agent will inhabit (i.e. the environment $Env$), and also
- the criteria by which an agent will be judged to have either failed or succeeded in its task (i.e. the specification $\Psi$).

Copyrighted material

Given a task environment $\langle Env, \Psi \rangle$, we write $\mathcal{R}_\Psi(Ag, Env)$ to denote the set of all runs of the agent $Ag$ in the environment $Env$ that satisfy $\Psi$. Formally,

$$\mathcal{R}_\Psi(Ag, Env) = \{r \mid r \in \mathcal{R}(Ag, Env) \text{ and } \Psi(r)\}.$$

We then say that an agent $Ag$ succeeds in task environment $\langle Env, \Psi \rangle$ if

$$\mathcal{R}_\Psi(Ag, Env) = \mathcal{R}(Ag, Env).$$

In other words, $Ag$ succeeds in $\langle Env, \Psi \rangle$ if every run of $Ag$ in $Env$ satisfies specification $\Psi$, i.e. if

$$\forall r \in \mathcal{R}(Ag, Env) \text{ we have } \Psi(r).$$

Notice that this is in one sense a *pessimistic* definition of success, as an agent is only deemed to succeed if every possible run of the agent in the environment satisfies the specification. An alternative, *optimistic* definition of success is that the agent succeeds if *at least one* run of the agent satisfies $\Psi$:

$$\exists r \in \mathcal{R}(Ag, Env) \text{ such that } \Psi(r).$$

If required, we could easily modify the definition of success by extending the state transformer function $\tau$ to include a probability distribution over possible outcomes, and hence induce a probability distribution over runs. We can then define the success of an agent as the probability that the specification $\Psi$ is satisfied by the agent. As before, let $P(r \mid Ag, Env)$ denote the probability that run $r$ occurs if agent $Ag$ is placed in environment $Env$. Then the probability $P(\Psi \mid Ag, Env)$ that $\Psi$ is satisfied by $Ag$ in $Env$ would simply be

$$P(\Psi \mid Ag, Env) = \sum_{r \in \mathcal{R}_\Psi(Ag, Env)} P(r \mid Ag, Env).$$

## Achievement and maintenance tasks

The notion of a predicate task specification may seem a rather abstract way of describing tasks for an agent to carry out. In fact, it is a generalization of certain very common forms of tasks. Perhaps the two most common types of tasks that we encounter are *achievement tasks* and *maintenance tasks*.

**Achievement tasks** Those of the form 'achieve state of affairs $\varphi$'.

**Maintenance tasks** Those of the form 'maintain state of affairs $\psi$'.

ACHIEVEMENT TASKS

Intuitively, an achievement task is specified by a number of *goal states*; the agent is required to bring about one of these goal states (we do not care which one – all are considered equally good). Achievement tasks are probably the most commonly studied form of task in AI. Many well-known AI problems (e.g. the Blocks World) are achievement tasks. A task specified by a predicate $\Psi$ is an achievement task if we can identify some subset $\mathcal{G}$ of environment states $E$ such that $\Psi(r)$ is true just in case one or more of $\mathcal{G}$ occur in $r$; an agent is successful if it is guaranteed to bring about one of the states $\mathcal{G}$, that is, if every run of the agent in the environment results in one of the states $\mathcal{G}$.

Copyrighted material

Formally, the task environment $\langle Env, \Psi \rangle$ specifies an achievement task if and only if there is some set $\mathcal{G} \subseteq E$ such that for all $r \in \mathcal{R}(Ag, Env)$, the predicate $\Psi(r)$ is true if and only if there exists some $e \in \mathcal{G}$ such that $e \in r$. We refer to the set $\mathcal{G}$ of an achievement task environment as the *goal states* of the task; we use $\langle Env, \mathcal{G} \rangle$ to denote an achievement task environment with goal states $\mathcal{G}$ and environment *Env*.

A useful way to think about achievement tasks is as the agent *playing a game* against the environment. In the terminology of game theory [Binmore, 1992], this is exactly what is meant by a 'game against nature'. The environment and agent both begin in some state; the agent takes a turn by executing an action, and the environment responds with some state; the agent then takes another turn, and so on. The agent 'wins' if it can *force* the environment into one of the goal states $\mathcal{G}$.

Just as many tasks can be characterized as problems where an agent is required to bring about some state of affairs, so many others can be classified as problems where the agent is required to *avoid* some state of affairs. As an extreme example, consider a nuclear reactor agent, the purpose of which is to ensure that the reactor never enters a 'meltdown' state. Somewhat more mundanely, we can imagine a software agent, one of the tasks of which is to ensure that a particular file is never simultaneously open for both reading and writing. We refer to such task environments as *maintenance* task environments.

MAINTENANCE TASKS

A task environment with specification $\Psi$ is said to be a maintenance task environment if we can identify some subset $\mathcal{B}$ of environment states, such that $\Psi(r)$ is false if any member of $\mathcal{B}$ occurs in $r$, and true otherwise. Formally, $\langle Env, \Psi \rangle$ is a maintenance task environment if there is some $\mathcal{B} \subseteq E$ such that $\Psi(r)$ if and only if for all $e \in \mathcal{B}$, we have $e \notin r$ for all $r \in \mathcal{R}(Ag, Env)$. We refer to $\mathcal{B}$ as the *failure set*. As with achievement task environments, we write $\langle Env, \mathcal{B} \rangle$ to denote a maintenance task environment with environment *Env* and failure set $\mathcal{B}$.

It is again useful to think of maintenance tasks as games. This time, the agent wins if it manages to *avoid* all the states in $\mathcal{B}$. The environment, in the role of opponent, is attempting to force the agent into $\mathcal{B}$; the agent is successful if it has a winning strategy for avoiding $\mathcal{B}$.

More complex tasks might be specified by *combinations* of achievement and maintenance tasks. A simple combination might be 'achieve any one of states $\mathcal{G}$ while avoiding all states $\mathcal{B}$'. More complex combinations are of course also possible.

## Synthesizing agents

Knowing that there exists an agent which will succeed in a given task environment is helpful, but it would be more helpful if, knowing this, we also had such an agent to hand. How do we obtain such an agent? The obvious answer is to 'manually' implement the agent from the specification. However, there are at least two other possibilities (see [Wooldridge, 1997] for a discussion):

AGENT SYNTHESIS

1. we can try to develop an algorithm that will *automatically synthesize* such agents for us from task environment specifications, or

2. we can try to develop an algorithm that will *directly execute* agent specifications in order to produce the appropriate behaviour.

Copyrighted material

In this section, I briefly consider these possibilities, focusing primarily on agent synthesis.

Agent synthesis is, in effect, automatic programming: the goal is to have a program that will take as input a task environment, and from this task environment automatically generate an agent that succeeds in this environment. Formally, an agent synthesis algorithm *syn* can be understood as a function

$$syn : \mathcal{TE} \to (\mathcal{AG} \cup \{\perp\}).$$

Note that the function *syn* can output an agent, or else output $\perp$ – think of $\perp$ as being like `null` in Java. Now, we will say a synthesis algorithm is

**sound** if, whenever it returns an agent, this agent succeeds in the task environment that is passed as input, and

**complete** if it is guaranteed to return an agent whenever there exists an agent that will succeed in the task environment given as input.

Thus a sound and complete synthesis algorithm will only output $\perp$ given input $\langle Env, \Psi \rangle$ when no agent exists that will succeed in $\langle Env, \Psi \rangle$.

Formally, a synthesis algorithm *syn* is sound if it satisfies the following condition:

$$syn(\langle Env, \Psi \rangle) = Ag \text{ implies } \mathcal{R}(Ag, Env) = \mathcal{R}_\Psi(Ag, Env).$$

Similarly, *syn* is complete if it satisfies the following condition:

$$\exists Ag \in \mathcal{AG} \text{ s.t. } \mathcal{R}(Ag, Env) = \mathcal{R}_\Psi(Ag, Env) \text{ implies } syn(\langle Env, \Psi \rangle) \neq \perp.$$

Intuitively, soundness ensures that a synthesis algorithm always delivers agents that do their job correctly, but may not always deliver agents, even where such agents are in principle possible. Completeness ensures that an agent will always be delivered where such an agent is possible, but does not guarantee that these agents will do their job correctly. Ideally, we seek synthesis algorithms that are both sound *and* complete. Of the two conditions, soundness is probably the more important; there is not much point in complete synthesis algorithms that deliver 'buggy' agents.

## Notes and Further Reading

A view of artificial intelligence as the process of agent design is presented in [Russell and Norvig, 1995], and, in particular, Chapter 2 of [Russell and Norvig, 1995] presents much useful material. The definition of agents presented here is based on [Wooldridge and Jennings, 1995], which also contains an extensive review of agent architectures and programming languages. The question of 'what is an agent' is one that continues to generate some debate; a collection of answers may be found in [Müller et al., 1997]. The relationship between agents and objects has not been widely discussed in the literature, but see [Gasser and Briot, 1992]. Other interesting and readable introductions to the idea of intelligent agents include [Kaelbling, 1986] and [Etzioni, 1993]. A collection of papers exploring the notion of autonomy in software agents is [Hexmoor et al., 2003].

The abstract model of agents presented here is based on that given in [Genesereth and Nilsson, 1987, Chapter 13], and also makes use of some ideas from [Russell and Wefald, 1991] and [Russell

Copyrighted material

and Subramanian, 1995]. The properties of perception as discussed in this section lead to *knowledge theory*, a formal analysis of the information implicit within the state of computer processes, which has had a profound effect in theoretical computer science: this issue is discussed in Chapter 17.

The relationship between artificially intelligent agents and software complexity has been discussed by several researchers: [Simon, 1981] was probably the first. More recently, [Booch, 1994] gives a good discussion of software complexity and the role that object-oriented development has to play in overcoming it. [Russell and Norvig, 1995] introduced classification of environments that we presented in the sidebar, and distinguished between the 'easy' and 'hard' cases. [Kaelbling, 1986] touches on many of the issues discussed here, and [Jennings, 1999] also discusses the issues associated with complexity and agents.

The relationship between agent and environment, and, in particular, the problem of understanding how a given agent will perform in a given environment, has been studied empirically by several researchers. [Pollack and Ringuette, 1990] introduced the Tileworld, an environment for experimentally evaluating agents that allowed a user to experiment with various environmental parameters (such as the rate at which the environment changes – its *dynamism*). We discuss these issues in Chapter 4. An informal discussion on the relationship between agent and environment is [Müller, 1999].

The link between the goal-oriented view and the utility-oriented view of task specifications has not received too much explicit attention in the literature. This is perhaps a little surprising, given that, until the 1990s, the goal-oriented view was dominant in artificial intelligence, while this century, the utility-oriented view has dominated. One nice discussion on the links between the two views is [Haddawy and Hanks, 1998].

More recently, there has been renewed interest by the artificial intelligence planning community in *decision theoretic* approaches to planning [Blythe, 1999]. One popular approach involves representing agents and their environments as 'partially observable Markov decision processes' (POMDPs) [Kaelbling et al., 1998]. Put simply, the goal of solving a POMDP is to determine an optimal policy for acting in an environment in which there is uncertainty about the environment state (cf. our visibility function), and which is non-deterministic. Work on POMDP approaches to agent design is at an early stage, but shows promise for the future. The discussion on task specifications is adapted from [Wooldridge, 2000a] and [Wooldridge and Dunne, 2000].

**Class reading: [Franklin and Graesser, 1997].** This paper informally discusses various different notions of agency. The focus of the discussion might be on a comparison with the discussion in this chapter.

Copyrighted material

Chapter 3

# Deductive Reasoning Agents

The 'traditional' approach to building artificially intelligent systems, known as *symbolic AI*, suggests that intelligent behaviour can be generated in a system by giving that system a *symbolic* representation of its environment and its desired behaviour, and syntactically manipulating this representation. In this chapter, we focus on the apotheosis of this tradition, in which these symbolic representations are *logical formulae*, and the syntactic manipulation corresponds to *logical deduction*, or *theorem proving*. SYMBOLIC AI

I will begin by giving an example to informally introduce the ideas behind deductive reasoning agents. Suppose we have some robotic agent, the purpose of which is to navigate around an office building picking up trash. There are many possible ways of implementing the control system for such a robot – we shall see several in the chapters that follow – but one way is to give it a description, or *representation*, of the environment in which it is to operate. Figure 3.1 illustrates the idea (adapted from [Konolige, 1986, p. 15]). SYMBOLIC REPRESENTATION

In order to build such an agent, it seems we must solve two key problems.

**The transduction problem** The problem of translating the real world into an accurate, adequate symbolic description of the world, in time for that description to be useful.

**The representation/reasoning problem** The problem of representing information symbolically, and getting agents to manipulate/reason with it, in time for the results to be useful.

The former problem has led to work on vision, speech understanding, learning, etc. The latter has led to work on knowledge representation, automated reasoning, automated planning, etc. Despite the immense volume of work that the problems have generated, many people would argue that neither problem is anywhere near solved. Even seemingly trivial problems, such as common-sense reasoning, have turned out to be extremely difficult.

Despite these problems, the idea of agents as theorem provers is seductive. Suppose we have some theory of agency – some theory that explains how an intelligent agent should behave so as to optimize some performance measure (see Chapter 2). This theory might

Copyrighted material

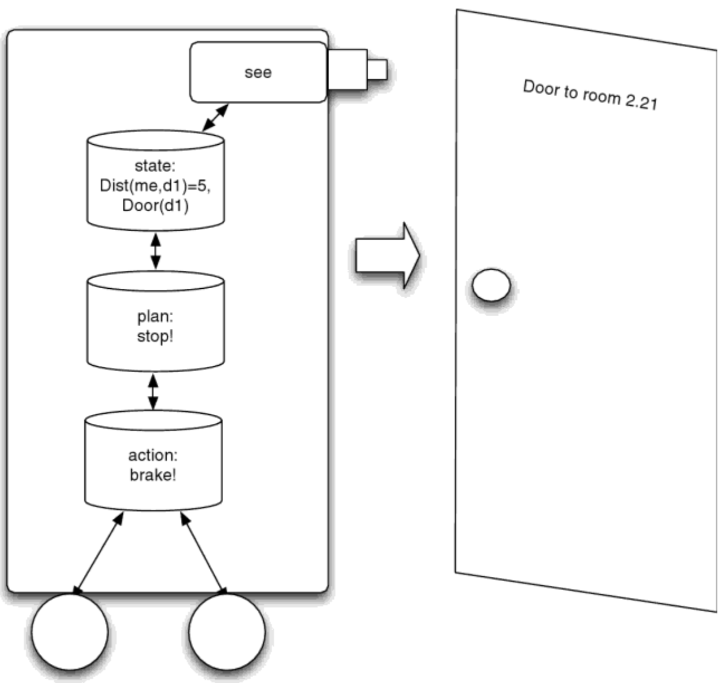

Figure 3.1: A robotic agent that contains a symbolic description of its environment.

explain, for example, how an agent generates goals so as to satisfy its design objective, how it interleaves goal-directed and reactive behaviour in order to achieve these goals, and so on. Then this theory $\varphi$ can be considered as a *specification* for how an agent should behave. The traditional approach to implementing a system that will satisfy this specification would involve *refining* the specification through a series of progressively more concrete stages, until finally an implementation was reached. In the view of agents as theorem provers, however, no such refinement takes place. Instead, $\varphi$ is viewed as an *executable specification*: it is *directly executed* in order to produce the agent's behaviour.

LOGICAL SPECIFICATION

EXECUTABLE SPECIFICATION

## 3.1 Agents as Theorem Provers

DELIBERATE AGENTS

To see how such an idea might work, we shall develop a simple model of logic-based agents, which we shall call *deliberate* agents [Genesereth and Nilsson, 1987, Chapter 13]. In such agents, the internal state is assumed to be a database of formulae of classical first-order predicate logic. For example, the agent's database might contain formulae such as

$$Open(valve221)$$
$$Temperature(reactor4726, 321)$$
$$Pressure(tank776, 28).$$

Copyrighted material

It is not difficult to see how formulae such as these can be used to represent the properties of some environment. The database is the *information* that the agent has about its environment. An agent's database plays a somewhat analogous role to that of *belief* in humans. Thus a person might have a belief that valve 221 is open – the agent might have the predicate $Open(valve221)$ in its database. Of course, just like humans, agents can be wrong. Thus I might believe that valve 221 is open when it is in fact closed; the fact that an agent has $Open(valve221)$ in its database does not mean that valve 221 (or indeed any valve) is open. The agent's sensors may be faulty, its reasoning may be faulty, the information may be out of date, or the interpretation of the formula $Open(valve221)$ intended by the agent's designer may be something entirely different.

Let $L$ be the set of formulae of classical first-order logic, and let $D = \mathbf{2}^L$ be the set of $L$ *databases*, i.e. the set of sets of $L$-formulae. The internal state of an agent is simply a set of formulae, i.e. an element of $D$. We write $DB, DB_1, \ldots$ for members of $D$. An agent's decision-making process is modelled through a set of *deduction rules*, $\rho$. These are simply rules of inference for the logic. We write $DB \vdash_\rho \varphi$ if the formula $\varphi$ can be proved from the database $DB$ using only the deduction rules $\rho$. An agent's perception function *see* remains unchanged:

BELIEF DATABASE

DEDUCTION RULES

$$see : S \to Per.$$

Similarly, our *next* function has the form

$$next : D \times Per \to D.$$

It thus maps a database and a percept to a new database. However, an agent's action selection function, which has the signature

$$action : D \to Ac,$$

is defined in terms of its deduction rules. The pseudo-code definition of this function is given in Figure 3.2.

The idea is that the agent programmer will encode the deduction rules $\rho$ and database $DB$ in such a way that if a formula $Do(\alpha)$ can be derived, where $\alpha$ is a term that denotes an action, then $\alpha$ is the best action to perform. Thus, in the first part of the function (lines 3–7), the agent takes each of its possible actions $\alpha$ in turn, and attempts to prove the formula $Do(\alpha)$ from its database (passed as a parameter to the function) using its deduction rules $\rho$. If the agent succeeds in proving $Do(\alpha)$, then $\alpha$ is returned as the action to be performed.

What happens if the agent fails to prove $Do(\alpha)$, for all actions $a \in Ac$? In this case, it attempts to find an action that is *consistent* with the rules and database, i.e. one that is not explicitly forbidden. In lines 8–12, therefore, the agent attempts to find an action $a \in Ac$ such that $\neg Do(\alpha)$ cannot be derived from its database using its deduction rules. If it can find such an action, then this is returned as the action to be performed. If, however, the agent fails to find an action that is at least consistent, then it returns a special action *null* (or *noop*), indicating that no action has been selected.

In this way, the agent's behaviour is determined by the agent's deduction rules (its 'program') and its current database (representing the information the agent has about its environment).

Copyrighted material

```
function $action(DB : D)$ returns an action $Ac$
begin
    for each $\alpha \in Ac$ do
        if $DB \vdash_\rho Do(\alpha)$ then
            return $\alpha$
        end-if
    end-for
    for each $\alpha \in Ac$ do
        if $DB \nvdash_\rho \neg Do(\alpha)$ then
            return $\alpha$
        end-if
    end-for
    return $null$
end function $action$
```

Figure 3.2: Action selection as theorem proving.

To illustrate these ideas, let us consider a small example (based on the vacuum cleaning world example of [Russell and Norvig, 1995, p. 51]). The idea is that we have a small robotic agent that will clean up a house. The robot is equipped with a sensor that will tell it whether it is over any dirt, and a vacuum cleaner that can be used to suck up dirt. In addition, the robot always has a definite orientation (one of *north*, *south*, *east*, or *west*). In addition to being able to suck up dirt, the agent can move forward one 'step' or turn right 90°. The agent moves around a room, which is divided grid-like into a number of equally sized squares (conveniently corresponding to the unit of movement of the agent). We will assume that our agent does nothing but clean – it never leaves the room, and further, we will assume in the interests of simplicity that the room is a $3 \times 3$ grid, and the agent always starts in grid square $(0, 0)$ facing north.

To summarize, our agent can receive a percept *dirt* (signifying that there is dirt beneath it), or *null* (indicating no special information). It can perform any one of three possible actions: *forward*, *suck*, or *turn*. The goal is to traverse the room continually searching for and removing dirt. See Figure 3.3 for an illustration of the vacuum world.

DOMAIN PREDICATES

First, note that we make use of three simple *domain predicates* in this exercise:

$$In(x, y) \quad \text{agent is at } (x, y), \tag{3.1}$$

$$Dirt(x, y) \quad \text{there is dirt at } (x, y), \tag{3.2}$$

$$Facing(d) \quad \text{the agent is facing direction } d. \tag{3.3}$$

Now we specify our *next* function. This function must look at the perceptual information obtained from the environment (either *dirt* or *null*), and generate a new database which includes this information. But, in addition, it must *remove* old or irrelevant information, and also, it must try to figure out the new location and orientation of the agent. We will therefore specify the *next* function in several parts. First, let us write *old*(*DB*) to denote the set of 'old' information in a database, which we want the update function *next* to remove:

$$old(DB) = \{P(t_1, \ldots, t_n) \mid P \in \{In, Dirt, Facing\} \text{ and } P(t_1, \ldots, t_n) \in DB\}.$$

Copyrighted material