Developing Intelligent Agent Systems
Wiley Series in Agent Technology

Series Editor: Michael Wooldridge, Liverpool University, UK

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

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

Titles in the series:

Padgham/Winikoff: Developing Intelligent Agent Systems 0470861207 (June 2004)
Pitt (ed.): Open Agent Societies 047148668X (August 2004)
Developing Intelligent Agent Systems
A practical guide

Lin Padgham & Michael Winikoff
RMIT University, Melbourne, Australia
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreword from the Series Editor</td>
<td>ix</td>
</tr>
<tr>
<td>Preface</td>
<td>xi</td>
</tr>
<tr>
<td>Acknowledgement</td>
<td>xiii</td>
</tr>
<tr>
<td>1  Agents and Multi-Agent Systems</td>
<td>1</td>
</tr>
<tr>
<td>1.1 What is an Intelligent Agent?</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Why are Agents Useful?</td>
<td>4</td>
</tr>
<tr>
<td>2  Concepts for Building Agents</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Situated Agents: Actions and Percepts</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Proactive and Reactive Agents: Goals and Events</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Challenging Agent Environments: Plans and Beliefs</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Social Agents</td>
<td>12</td>
</tr>
<tr>
<td>2.5 Agent Execution Cycle</td>
<td>13</td>
</tr>
<tr>
<td>2.5.1 Choice of Plan to Execute</td>
<td>14</td>
</tr>
<tr>
<td>2.5.2 Many Ways to Achieve a Goal</td>
<td>16</td>
</tr>
<tr>
<td>2.6 Summary</td>
<td>19</td>
</tr>
<tr>
<td>3  Overview of the Prometheus Methodology</td>
<td>21</td>
</tr>
<tr>
<td>3.1 Why a New Methodology?</td>
<td>22</td>
</tr>
<tr>
<td>3.2 Prometheus: A Brief Overview</td>
<td>23</td>
</tr>
<tr>
<td>3.2.1 System Specification</td>
<td>24</td>
</tr>
<tr>
<td>3.2.2 Architectural Design</td>
<td>25</td>
</tr>
<tr>
<td>3.2.3 Detailed Design</td>
<td>26</td>
</tr>
<tr>
<td>3.3 Guidelines for Using Prometheus</td>
<td>27</td>
</tr>
<tr>
<td>3.4 Agent-Oriented Methodologies</td>
<td>29</td>
</tr>
<tr>
<td>4  System Specification</td>
<td>33</td>
</tr>
<tr>
<td>4.1 Goal Specification</td>
<td>34</td>
</tr>
<tr>
<td>4.1.1 Identify Initial Goals</td>
<td>35</td>
</tr>
<tr>
<td>4.1.2 Goal Refinement</td>
<td>36</td>
</tr>
<tr>
<td>4.2 Functionalities</td>
<td>41</td>
</tr>
<tr>
<td>4.3 Scenario Development</td>
<td>43</td>
</tr>
<tr>
<td>4.3.1 Goal Step Details</td>
<td>45</td>
</tr>
<tr>
<td>4.3.2 Capturing Alternative Scenarios</td>
<td>46</td>
</tr>
</tbody>
</table>
4.4 Interface Description ........................................ 47
  4.4.1 Percepts and Actions ................................ 48
  4.4.2 Data .................................................. 50
4.5 Checking for Completeness and Consistency ................. 51

5 Architectural Design: Specifying the Agent Types ............. 53
  5.1 Deciding on the Agent Types ................................ 56
  5.2 Grouping Functionalities .................................. 56
  5.3 Review Agent Coupling – Acquaintance Diagrams .......... 63
  5.4 Develop Agent Descriptors ................................ 65

6 Architectural Design: Specifying the Interactions ............. 67
  6.1 Interaction Diagrams from Scenarios ...................... 68
  6.2 Interaction Protocols from Interaction Diagrams .......... 74
  6.3 Develop Protocol and Message Descriptors ............... 77

7 Finalizing the Architectural Design ............................. 81
  7.1 Overall System Structure .................................. 82
  7.2 Identifying Boundaries of the Agent System ............... 82
  7.3 Describing Percepts and Actions .......................... 84
  7.4 Defining Shared Data Objects ............................. 88
  7.5 System Overview Diagram ................................ 91
  7.6 Checking for Completeness and Consistency ............... 94
    7.6.1 Consistency between Agents and Functionalities .. 95
    7.6.2 Consistency between Interaction Diagrams, Scenarios
           and Protocols ........................................ 95
    7.6.3 Consistency of Communication Specifications ....... 96
    7.6.4 Consistency between Descriptors and the System
           Overview Diagram .................................... 97

8 Detailed Design: Agents, Capabilities and Processes ........... 99
  8.1 Capabilities ............................................. 100
  8.2 Agent Overview Diagrams .................................. 102
  8.3 Process Specifications .................................... 103
  8.4 Develop Capability and Process Descriptors .............. 106

9 Detailed Design: Capabilities, Plans and Events ................ 109
  9.1 Capability Overview Diagrams ............................. 110
  9.2 Sub-tasks and Alternative Plans ........................... 112
    9.2.1 Identifying Context Conditions ....................... 113
    9.2.2 Coverage and Overlap ................................ 115
  9.3 Events and Messages ...................................... 116
  9.4 Action and Percept Detailed Design ....................... 117
  9.5 Data ..................................................... 118
  9.6 Develop and Refine Descriptors ........................... 118
  9.7 Checking for Completeness and Consistency ............... 119
    9.7.1 Agent Completeness .................................. 120
Foreword from the Series Editor

As the concepts and technologies associated with intelligent software agents make their transition from the research lab to the desk of the IT practitioner, issues such as reliable analysis and design methodologies come increasingly to the fore. If agent technology is to mature into a successful and widely used approach to software development, then it is of critical importance that methodologies are developed, which are accessible to students and IT professionals alike, enabling them to deploy this new and promising technology to full effect. Although several such methodologies have been tentatively proposed, the PROMETHEUS methodology set out in this book is arguably the most mature.

PROMETHEUS is a general purpose methodology for the development of software agent systems, in that it is not tied to any specific model of agency or software platform. The authors do an excellent job of describing the models and methods associated with PROMETHEUS and they show how these can be used to analyse and design multiagent systems by means of a detailed running example. Associated with the methodology, the authors have developed a freely available software design tool (PDT), which represents the state-of-the-art in multiagent systems development tools. All in all, this book represents a valuable contribution, not just for those with an interest in the ongoing debate about development methods for multiagent systems but also for those who simply want an answer to the question: How do I actually do it?

Mike Wooldridge, Liverpool, June 2004
Intelligent software agents\(^1\) are a powerful technology that is attracting considerable (and growing!) interest.

While there are books that cover research areas on agents or survey the field (including the excellent book by Michael Wooldridge (Wooldridge 2002)), there is no book that is aimed at an industrial software developer that answers not only the questions ‘what are agents?’ and ‘why are they useful?’ but also the crucial question ‘how do I design and build intelligent software agents?’.

Our book aims to provide a practical introduction to building intelligent agent systems. It covers everything a practitioner needs to know to build multi-agent systems of intelligent agents. It includes an introduction to the notion of agents, a description of the concepts involved, and a software engineering methodology covering specification, analysis, design and implementation of agent systems.

The core of the book is the *Prometheus* methodology for designing multi-agent systems. The methodology was developed over the past six or seven years in collaboration with Agent Oriented Software\(^2\), a company that markets the agent development platform, JACK\(^TM\) as well as agent solutions. The methodology has been used internally at Agent Oriented Software and has also been taught at industry workshops and within university courses. It has proven effective in assisting students and practitioners to develop and document their design and is now at a sufficient level of maturity that support tools have been developed.

Our goal in developing Prometheus was to have a process with associated deliverables that could be taught to industry practitioners and undergraduate students who do not have a background in agents and which they could then use to develop intelligent agent systems. Our evidence that we have achieved this is, at this stage, still anecdotal; however, the indications are that Prometheus *is* usable by nonexperts and that they find it useful.

We do *not* believe that Prometheus is complete and perfect, nor that it is a perfect fit as-is for all applications and all users. However, we do believe that it is usable and in our experience it is much better than not having a methodology. Like most methodologies, Prometheus is intended to be interpreted as a set of guidelines and you should use your common sense and take what is useful, adapting the methodology as needed to suit your needs.

Although we do not believe that Prometheus is perfect, it *is* general purpose in the sense of not being specific to BDI (Belief-Desire-Intention) agents. Only the later part of

\(^1\)This is shortened to ‘agents’ in the remainder of this book.

\(^2\)\texttt{http://www.agent-software.com}
the detailed design phase (Chapter 9) makes assumptions about particular types of agent platforms. The assumptions made are fairly general and correspond to a class of agent platforms that have hierarchical plans with triggers, and a description for each plan that indicates the context in which it is applicable.

**Electronic Bookstore: Case study**

We will be using an example of an electronic bookstore to illustrate the design process throughout the book. To enable easy following of the example, we will enclose all of these examples in a framed box (like this one), which may extend over page breaks, in which case the bottom and top of the frame on the adjacent pages will be missing. In addition, the collected details of the example can be found in Appendix A.

Electronic resources, including the forms in Appendix B and the Prometheus Design Tool (PDT), can be found at


**Audience**

This book is aimed at industrial software developers and at undergraduate students. It assumes knowledge of basic software engineering but does not require knowledge of Artificial Intelligence or of mathematics. Familiarity with Java will help in reading the examples in Chapter 10.

**Tool Support**

We believe that tool support is, if not essential, incredibly useful in developing large designs and in helping to keep them consistent. Thus, we have developed a prototype tool, the Prometheus Design Tool (PDT). This tool supports the process described in this book, of system specification, architectural design and detailed design. The detailed design produced by PDT can be straightforwardly converted to the JACK Development Environment (JDE), and consequently to JACK code. A similar approach could be used to develop plug-ins that enable PDT to produce skeleton code for a range of agent programming platforms.
Acknowledgements

This book has evolved out of material originally developed with Agent Oriented Software (AOS) and refined over the course of a number of years of teaching an Agent Oriented Programming and Design course and supervising students in developing multi-agent systems. We thank our colleagues at AOS, especially Andrew Lucas and Ralph Rönnquist, as well as the many students who took our courses and provided valuable feedback.

We acknowledge the support of the Australian Research Council (ARC) under grant CO01069343, and its continuation, grant LP0453486.

The Prometheus Design Tool was initially developed by Anna Edberg and Christian Andersson. Further development has been done by Claire Hennekam and Jason Khallouf.

We especially thank Ian Mathieson for his careful reading and commenting – this book has benefited considerably from his detailed comments.

Lin Padgham & Michael Winikoff
June 2004
Melbourne, AUSTRALIA

---

Agents and Multi-Agent Systems

This book is about designing and implementing intelligent agent systems. We therefore begin by answering the obvious first question, namely, ‘What is an agent?’. We answer this question by discussing the properties that characterize an intelligent agent, and contrast agents with objects. The usual second question is ‘Why should I bother with agents?’. We answer this question by arguing that agents are a natural progression from objects that provide a better abstraction and improved encapsulation, and also, perhaps more convincingly, by looking at applications of agent technology.

The remaining chapters of this book are dedicated to answering the third question ‘How do I develop agents and agent systems?’.

1.1 WHAT IS AN INTELLIGENT AGENT?

As is to be expected from a fairly young area of research, there is not yet a universal consensus on the definition of an agent. However, the Wooldridge and Jennings definition (see below) is increasingly adopted, and it is probably fair to say that most researchers in the field, when asked to provide their definition, will mention various properties drawn from those we discuss below.

☞ Definition: The following definition is from (Wooldridge 2002), which in turn 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 design objectives’.

Wooldridge distinguishes between an agent and an intelligent agent, which is further required to be reactive, proactive and social (Wooldridge 2002, page 23).
Let us first note that we are talking about software agents. Whenever we (or any other researcher in the field) say ‘agent’, we really mean ‘software agent’. The typical dictionary definition of agent as ‘an entity having the authority to act on behalf of another’ (e.g. a real estate agent) is not what we mean\(^1\).

Two basic properties of software agents are that they are autonomous and that they are situated in an environment. The first property, being autonomous, means that agents are independent and make their own decisions. This is one of the properties that distinguishes agents from objects. When we consider a system consisting of a number of agents, then a consequence of the agents being autonomous is that the system tends to be decentralized (we shall return to this in the next section).

The second property (situatedness) does not constrain the notion of an agent very much since virtually all software can be considered to be situated in an environment. However, where agents differ is the type of environments. Agents tend to be used where the environment is challenging; more specifically, typical agent environments\(^2\) are dynamic, unpredictable and unreliable. These environments are dynamic in that they change rapidly. By ‘rapidly’, we mean that the agent cannot assume that the environment will remain static while it is trying to achieve a goal. These environments are unpredictable in that it is not possible to predict the future states of the environment; often this is because it is not possible for an agent to have perfect and complete information about their environment, and because the environment is being modified in ways beyond the agent’s knowledge and influence. Finally, these environments are unreliable in that the actions that an agent can perform may fail for reasons that are beyond an agent’s control. For example, a robot attempting to lift an item may fail for a wide range of reasons including the item being too heavy.

Agents are often situated in dynamic environments that change rapidly. In particular, this means that an agent must respond to significant changes in its environment. For example, an agent controlling a robot playing soccer can make plans on the basis of the current position of the ball and of other players, but it must be prepared to adapt or abandon its plans should the environment change in a significant way. In other words, agents need to be reactive, responding in a timely manner to changes in their environment.

Another key property of agents is that they pursue goals over time, that is, they are proactive. One property of goals is that they are persistent; this is useful in that it makes agents more robust: an agent will continue to attempt to achieve a goal despite failed attempts.

Although objects can be reactive, and can be seen as having an implicit goal, they are not proactive in the sense of having multiple goals, and of these goals being explicit and persistent. Thus, proactiveness is another property that distinguishes agents from objects.

A key issue in agent architectures is balancing reactiveness and proactiveness. On the one hand, an agent should be reactive, so its plans and actions should be influenced by environmental changes. On the other hand, an agent’s plans and actions should be influenced by its goals. The challenge is to balance these two (often conflicting) influences: if the agent is too reactive, then it will be constantly adjusting its plans and not achieve

\(^1\)Some software agents may act as agents in this sense as well. For example, a software assistant that buys products or services on behalf of its user.

\(^2\)A more detailed analysis of different properties of environments can be found in (Wooldridge 2002) based on the taxonomy of (Russell and Norvig 1995).
WHAT IS AN INTELLIGENT AGENT?

its goals. However, if the agent is not sufficiently reactive, then it will waste time trying to follow plans that are no longer relevant or applicable.

Since failure of actions (and, more generally, of plans) is a possibility in challenging environments, agents must be able to recover from such failures, that is, they must be robust. A natural approach to achieving robustness is to be flexible. By having a range of ways of achieving a given goal, the agent has alternatives that can be used should a plan fail. These two properties are also distinguishing features of agents as compared to objects.

Finally, agents almost always need to interact with other agents, that is, agents are social. This interaction is often at a higher level: instead of just saying that agents exchange messages, agent interaction can be framed in terms of performatives\(^3\) such as ‘inform’, ‘request’, ‘agree’, and so on. These have standard semantics that are defined in terms of their effects on an agent’s mental state. Agent interaction is often viewed in terms of human interaction types such as negotiation, coordination, cooperation and teamwork.

On the basis of these properties, we use the following definition:

\begin{definition}
An Intelligent Agent is a piece of software that is
\begin{itemize}
  \item Situated – exists in an environment
  \item Autonomous – independent, not controlled externally
  \item Reactive – responds (in a timely manner!) to changes in its environment
  \item Proactive – persistently pursues goals
  \item Flexible – has multiple ways of achieving goals
  \item Robust – recovers from failure
  \item Social – interacts with other agents.
\end{itemize}
\end{definition}

In addition to these properties, there are a number of other properties that we regard as less central. This does not mean that these are not important, just that they are not important for all agent applications.

In pursuing their goals, we want agents to be rational. Part of being rational is that an agent should not do ‘dumb’ things such as simultaneously committing to two courses of action that conflict. For example, planning to spend money on a holiday at the same time as planning to spend that same money on a car. A detailed analysis of what is meant by ‘rational’ can be found in the work of Bratman (Bratman 1987). This analysis forms the basis of the Belief-Desire-Intention model for software agents (Rao and Georgeff 1992).

---

\(^3\)This is based on speech act theory, which is beyond the scope of this book, and we refer the reader to Chapter 8 of (Wooldridge 2002) for more details.
One definition of agents (strong agency) takes these various properties, and also requires that agents are viewed as having mental attitudes such as beliefs, goals and intentions. This intentional stance (Dennett 1987) has a surprisingly pragmatic justification: as a system becomes more complex, its behaviour can be predicted more reliably by abstracting away from how it achieves its goals and instead reasoning about what are its goals and beliefs. For example, in attempting to ascertain whether a piece of furniture will support a person’s weight, we could model the stress and calculate its load-bearing ability, or we could consider its design and reason that the goal of a chair is to be sat upon and so any chair should be able to support a person’s weight. The former stance is the ‘physical stance’, the latter is the ‘design stance’. The ‘intentional stance’ is an extension of this that is applied to active entities.

Although having agents that learn from their experiences can be essential for some applications, it can be disastrous for others. Similarly, there are applications in which modelling human emotions can be useful, such as interface agents or computer games, but equally, there are many applications in which this is not relevant.

There is a whole body of work devoted to mobile agents (Harrison et al. 1995; Kotz and Gray 1999). However, there is surprisingly little overlap between the work on intelligent agents and the work on mobile agents. Mobility is more of a system-level issue, with much work devoted to questions such as ‘How can a running program be stopped, moved to another machine and restarted?’ and associated issues in security.

### 1.2 WHY ARE AGENTS USEFUL?

Having described what agents are, we now turn to the question of why agent technology is useful. It is important to realize that, like other software technologies such as objects, agents are not magic. They are simply an approach to structuring and developing software that offers certain benefits, and that is very well suited to certain types of applications (in fact, one viewpoint considers agents to be an evolutionary step forward from objects (Odell 2002)). In order to understand why agents are useful, we need to understand how the distinctive features of agents translate into properties of software systems that are designed and built using agents. The usefulness of these properties (such as being decentralized) depends on the application, and so it is important to also understand how these software-system properties relate to application types and application areas.

In addition to looking at application types, we try to provide examples of documented applications. Unfortunately, the field of agents is still quite young, so there are not many well-documented applications. However, some applications have been documented in the literature (e.g. Jennings and Wooldridge (1998b)).

Perhaps the single most important advantage of agents is that they reduce coupling. Agents are autonomous, which can be seen as encapsulating invocation (Odell 2002; Parunak 1997). Whereas an object makes available methods that are triggered externally, an agent does not provide any control point to external entities. When a message is sent to an agent, the agent (being autonomous) has control over how it deals with the message.

Coupling is reduced not only by the encapsulation provided by autonomy but also by the robustness, reactiveness and proactiveness of agents. An agent can be relied upon
WHY ARE AGENTS USEFUL?

to persist in achieving its goals, trying alternatives that are appropriate to the changing environment. This means that when an agent takes on a goal, the responsibility for achieving that goal rests with that agent. Continuous supervision and checking is not needed. As an analogy, view an object as a reliable employee that lacks initiative and a sense of responsibility; supervising such an employee requires a significant amount of communication. On the other hand, an agent can be viewed as an employee that has a sense of responsibility and shows initiative. Supervising such an employee requires considerably less communication, and hence less coupling.

Reduced coupling can lead to software systems that are more modular, more decentralized and more changeable. This has led to the application of agents as an architectural ‘glue’ in a range of software applications. In this usage, agents are often used to ‘wrap’ legacy software. For example, see the list of applications built with Open Agent Architecture (OAA) listed in (Cheyer and Martin 2001).

It has been argued that agents are ‘well suited for developing complex distributed systems’ (Jennings 2001) since they provide more natural abstraction and decomposition of complex ‘nearly-decomposable’ systems.

One increasingly important class of systems that exhibit decentralization, complexity and distribution is open systems: software systems in which different parts are designed and written by different authors, without there being communication between the different authors. An example is the World Wide Web, where the authors of a web browser and of a web server probably did not ever talk to each other. Not surprisingly, standards play a key role in enabling software that was independently developed to work together. The web is a simple example of an open system since it is essentially concerned with transporting static documents, as opposed to providing services that change the state of servers. Other, more complex, examples of such systems include the semantic web and web services (Hendler 2001; McIlraith et al. 2001), and grid computing (Moreau 2002; Moreau et al. 2002).

In addition to providing reduced coupling, agents are also clearly applicable in situations in which the environment is challenging (dynamic, unpredictable, unreliable), in which failure is a possibility and in which recovery from failure must be done autonomously. An extreme example of an agent system that was required to deal with such situations was Remote Agent (Muscettola et al. 1998), which, in May 1999, was in control of NASA’s Deep Space 1 for two days, over 96 500 000 kilometres from the Earth.

Being proactive and reactive makes agents more human-like in the way they deal with problems. This has led to a number of applications in which software agents are used as substitutes for humans in certain limited domains. One application is the use of software agents to substitute for human pilots in military simulations (Tidhar et al. 1998). Other, more peaceful, applications include entertainment. The recent computer game Black & White used agents, specifically based on the Belief-Desire-Intention (BDI) model that is widely used in the agents community:

‘... To make agents who were psychologically plausible, we took the Belief-Desire-Intention architecture of an agent, fast becoming orthodoxy in the agent programming community, and developed it in a variety of ways ...’
Another area where agents have been applied is in film-making. The recent film *Lord of the Rings: The Two Towers* used a software package called *Massive* to generate the armies of Orcs, Elves and Humans. Each individual character was modelled as an agent.

Other application areas where software agents can provide benefits include Intelligent Assistants (Maes 1994), Electronic Commerce (Luck *et al.* 2003), Manufacturing (Shen and Norrie 1999), and Business process modelling (Jennings *et al.* 2000a,b).
Concepts for Building Agents

In the previous chapter, we defined agents as having a number of properties such as being situated, proactive and reactive. In this chapter, we begin to look at how we can design and build software that has these properties. We begin by considering what concepts lead to agents having certain properties. For example, in order for an agent to be proactive, it needs to have goals. Thus, the concept of a goal is an important one for designing and building proactive agents.

A software-engineering methodology assumes the existence of a set of concepts that it builds upon. For example, object-oriented notations such as UML (Booch et al. 1999) assume certain concepts such as object, class, inheritance, and so on. With agent-oriented methodologies, we also need an appropriate set of underlying concepts, and, not surprisingly, it turns out that the set of concepts is different to the object-oriented set.

The concepts that we describe in this chapter are used by the Prometheus methodology, which is the methodology covered in detail within the later sections of this book. Prometheus has been developed specifically in response to a need for assistance and direction in designing and building agent systems.

Our experience has been that the concepts identified are both necessary and sufficient for building the sort of applications that are appropriately approached using plan-based agents, and that they are simple and can be understood by undergraduate students. These concepts are based on the definition of agents, and in the remainder of this chapter, we explain these concepts and why they are appropriate.

2.1 SITUATED AGENTS: ACTIONS AND PERCEPTS

We began our definition of an agent with the basic property that an agent is software that is situated in an environment. The two concepts that capture the interface between an agent and its environment are the percepts from the environment and the actions that the agent can perform to affect the environment (Russell and Norvig 1995) (see Figure 2.1).
A percept\(^1\) is an item of information received from the environment by some sensor. For example, a fire-fighting robot may receive information such as the location of a fire and an indication of its intensity. An agent may also obtain information about the environment through sensing actions.

An action is something that an agent does, such as move_north or squirt. Agents are situated, and an action is basically an agent’s ability to affect its environment. In their simplest form, actions are atomic and instantaneous and either fail or succeed. In the more general case, actions can be durational (encompassing behaviours over time) and can produce partial effects; for example, even a failed move_to action may well have changed the agent’s location. In addition to external actions that directly affect the agent’s environment, we also want to consider internal actions. These correspond to an ability that the agent has, which is not structured in terms of plans and goals. Typically, the ability is a piece of code that either already exists or would not benefit from being written using agent concepts, for example, image processing in a vision sub-system.

At a very abstract level, we can view an agent as receiving percepts from an environment, somehow selecting an action to perform, and performing that action. These repeated steps form the execution cycle of an abstract agent. The concepts discussed in the following sections refine the internal execution of the agent. For example, by adding goals, these are able to be used to execute a series of actions over a period of time.

### 2.2 PROACTIVE AND REACTIVE AGENTS: GOALS AND EVENTS

We want our intelligent agents to be both proactive and reactive. The agent’s proactiveness implies the use of goals. A reactive agent is one that will change its behaviour in

\(^{1}\)From the Latin ‘perceptum’, which is the root of the English word ‘perceive’.
response to changes in the environment. An important aspect in decision-making is balancing proactive and reactive aspects. On the one hand we want the agent to stick with its goals by default and on the other hand we want it to take changes in the environment into account. The key to reconciling these aspects, thus making agents suitably reactive, is identifying significant changes in the situation. These are events (see Figure 2.2).

A goal (variously called task, objective, aim or desire) is something the agent is working on or towards, for example, extinguish fire, or rescue civilian. Often, goals are defined as states of the world that the agent wants to bring about; however, this definition does not allow some types of goals to be expressed such as maintenance goals (e.g. maintain cruising altitude), avoidance goals or safety constraints (e.g. never move the table while the robot is drilling). Goals give the agent its autonomy and proactiveness. An important aspect of proactiveness is the persistence of goals: if a plan for achieving a goal fails, then the agent will consider alternative plans for achieving the goal in question, until it is believed impossible or is no longer relevant.

An event is a significant occurrence that the agent should respond to in some way. Events are often extracted from percepts, although they may be generated internally by the agent, for example, on the basis of a clock. An event can trigger new goals, cause changes in information about the environment and/or cause actions to be performed immediately. Actions generated directly by events correspond to reflexive actions, executed without deliberation. Events are important in creating reactive agents in that they identify important changes that the agent needs to react to.

Percepts can be seen as particular kinds of events that are generated within the environment. We note that the percept may well have to be interpreted from the raw data available, in order to provide the percept/event that has significance to the agent. Particularly, if the raw data is an image data, it is likely to require significant processing. The first layer would extract features in the image, but even then it is likely to need to