1 Applications of Intelligent Agents

N. R. Jennings and M. Wooldridge
Queen Mary & Westfield College
University of London

1.1 Introduction

Intelligent agents are a new paradigm for developing software applications. More than this, agent-based computing has been hailed as 'the next significant breakthrough in software development' (Sargent, 1992), and 'the new revolution in software' (Ovum, 1994). Currently, agents are the focus of intense interest on the part of many sub-fields of computer science and artificial intelligence. Agents are being used in an increasingly wide variety of applications, ranging from comparatively small systems such as email filters to large, open, complex, mission critical systems such as air traffic control. At first sight, it may appear that such extremely different types of system can have little in common. And yet this is not the case: in both, the key abstraction used is that of an *agent*. Our aim in this article is to help the reader to understand why agent technology is seen as a fundamentally important new tool for building such a wide array of systems. More precisely, our aims are five-fold:

- to introduce the reader to the concept of an agent and agent-based systems,
- to help the reader to recognize the domain characteristics that indicate the appropriateness of an agent-based solution,
- to introduce the main application areas in which agent technology has been successfully deployed to date,
- to identify the main obstacles that lie in the way of the agent system developer, and finally
- to provide a guide to the remainder of this book.

We begin, in this section, by introducing some basic concepts (such as, perhaps most importantly, the notion of an agent). In Section 1.2, we give some general guidelines on the types of domain for which agent technology is appropriate. In Section 1.3, we survey the key application domains for intelligent agents. In Section 1.4, we discuss some issues in agent system development, and finally, in Section 1.5, we outline the structure of this book.

Before we can discuss the development of agent-based systems in detail, we have to describe what we mean by such terms as 'agent' and 'agent-based system'. Unfortunately, we immediately run into difficulties, as some key concepts in agent-based computing lack universally accepted definitions. In particular, there is no real agreement even on the core question of exactly what an agent is (see Franklin and Graesser (1996) for a discussion). However, we believe that most researchers

would find themselves in broad agreement with the following definitions (Wooldridge and Jennings, 1995).

First, an agent is a computer system situated in some environment, and that is capable of *autonomous action* in this environment in order to meet its design objectives. Autonomy is a difficult concept to pin down precisely, but we mean it simply in the sense that the system should be able to act without the direct intervention of humans (or other agents), and should have control over its own actions and internal state. It may be helpful to draw an analogy between the notion of autonomy with respect to agents and encapsulation with respect to object-oriented systems. An object encapsulates some state, and has some control over this state in that it can only be accessed or modified via the methods that the object provides. Agents encapsulate state in just the same way. However, we also think of agents as encapsulating *behavior*, in addition to state. An object does not encapsulate behavior: it has no control over the execution of methods – if an object *x* invokes a method *m* on an object *y*, then *y* has no control over whether *m* is executed or not – it just *is*. In this sense, object *y* is not autonomous, as it has no control over its own actions. In contrast, we think of an agent as having *exactly* this kind of control over what actions it performs. Because of this distinction, we do not think of agents as invoking methods (actions) on agents – rather, we tend to think of them *requesting* actions to be performed. The decision about whether to act upon the request lies with the recipient.

Of course, autonomous computer systems are not a new development. There are many examples of such systems in existence. Examples include:

- any process control system, which must monitor a real-world environment and perform actions to modify it as conditions change (typically in real time) – such systems range from the very simple (for example, thermostats) to the extremely complex (for example, nuclear reactor control systems),
- software daemons, which monitor a software environment and perform actions to modify the environment as conditions change – a simple example is the UNIX `xbiff` program, which monitors a user's incoming email and obtains their attention by displaying an icon when new, incoming email is detected.

It may seem strange that we choose to call such systems agents. But these are not *intelligent* agents. An intelligent agent is a computer system that is capable of *flexible* autonomous action in order to meet its design objectives. By *flexible*, we mean that the system must be:

- *responsive*: agents should perceive their environment (which may be the physical world, a user, a collection of agents, the Internet, etc.) and respond in a timely fashion to changes that occur in it,
- *proactive*: agents should not simply act in response to their environment, they should be able to exhibit opportunistic, goal-directed behavior and take the initiative where appropriate, and

- *social*: agents should be able to interact, when they deem appropriate, with other artificial agents and humans in order to complete their own problem solving and to help others with their activities.

Hereafter, when we use the term ‘agent’, it should be understood that we are using it as an abbreviation for ‘intelligent agent’. Other researchers emphasize different aspects of agency (including, for example, mobility or adaptability). Naturally, some agents may have additional characteristics, and for certain types of applications, some attributes will be more important than others. However, we believe that it is the presence of all four attributes in a single software entity that provides the power of the agent paradigm and which distinguishes agent systems from related software paradigms – such as object-oriented systems, distributed sys-ems, and expert systems (see Wooldridge (1997) for a more detailed discussion).

By an *agent-based* system, we mean one in which the key abstraction used is that of an agent. In principle, an agent-based system might be conceptualized in terms of agents, but implemented without any software structures corresponding to agents at all. We can again draw a parallel with object-oriented software, where it is entirely possible to design a system in terms of objects, but to implement it without the use of an object-oriented software environment. But this would at best be unusual, and at worst, counterproductive. A similar situation exists with agent technology; we therefore expect an agent-based system to be both designed and implemented in terms of agents. A number of software tools exist that allow a user to implement software systems as agents, and as societies of cooperating agents.

Note that an agent-based system may contain any non-zero number of agents. The *multi*-agent case – where a system is designed and implemented as several interacting agents, is both more general and significantly more complex than the *single*-agent case. However, there are a number of situations where the single-agent case is appropriate. A good example, as we shall see later in this chapter, is the class of systems known as *expert assistants*, wherein an agent acts as an expert assistant to a user attempting to use a computer to carry out some task.

1.2 Agent Application Domain Characteristics

Now that we have a better understanding of what the terms ‘agent’ and ‘agent-based system’ mean, the obvious question to ask is: *what do agents have to offer*? For any new technology to be considered as useful in the computer marketplace, it must offer one of two things:

- the ability to solve problems that have hitherto been beyond the scope of automation – either because no existing technology could be used to solve the problem, or because it was considered too expensive (difficult, time-consuming, risky) to develop solutions using existing technology; or
- the ability to solve problems that can already be solved in a significantly better (cheaper, more natural, easier, more efficient, or faster) way.

1.2.1 Solving New Types of Problem

Certain types of software system are inherently more difficult to correctly design and implement than others. The *simplest* general class of software systems are *functional*. Such systems work by taking some input, computing a function of it, and giving this result as output. Compilers are obvious examples of functional systems. In contrast, *reactive* systems, which maintain an ongoing interaction with some environment, are inherently much more difficult to design and correctly implement. Process control systems, computer operating systems, and computer network management systems are all well-known examples of reactive systems. In all of these examples, a computer system is required that can operate independently, typically over long periods of time. It has long been recognized that reactive systems are among the most complex types of system to design and implement (Pnueli, 1986), and a good deal of effort has been devoted to developing software tools, programming languages, and methodologies for managing this complexity – with some limited success. However, for certain types of reactive system, even specialized software engineering techniques and tools fail – new techniques are required. We can broadly subdivide these systems into three classes:

- open systems,
- complex systems, and
- ubiquitous computing systems.

1.2.1.1 Open Systems

An open system is one in which the structure of the system itself is capable of dynamically changing. The characteristics of such a system are that its components are not known in advance, can change over time, and may be highly heterogeneous (in that they are implemented by different people, at different times, using different software tools and techniques). Computing applications are increasingly demanded by users to operate in such domains. Perhaps the best-known example of a highly open software environment is the Internet – a loosely coupled computer network of ever expanding size and complexity. The design and construction of software tools to exploit the enormous potential of the Internet and its related technology is one of the most important challenges facing computer scientists in the 1990s, and for this reason, it is worth using it as a case study. The Internet can be viewed as a large, distributed information resource, with nodes on the network designed and implemented by different organizations and individuals with widely varying agendas. Any computer system that must operate on the Internet must be capable of dealing with these different organizations and agendas, without constant guidance from users (but within well-defined bounds). Such functionality is almost certain to require techniques based on negotiation or cooperation, which lie very firmly in the domain of multi-agent systems (Bond and Gasser, 1988).

1.2.1.2 Complex Systems

The most powerful tools for handling complexity in software development are *modularity* and *abstraction*. Agents represent a powerful tool for making systems modular. If a problem domain is particularly complex, large, or unpredictable, then it may be that the only way it can reasonably be addressed is to develop a number of (nearly) modular components that are specialized (in terms of their representation and problem solving paradigm) at solving a particular aspect of it. In such cases, when interdependent problems arise, the agents in the systems must cooperate with one another to ensure that interdependencies are properly managed. In such domains, an agent-based approach means that the overall problem can be partitioned into a number of smaller and simpler components, which are easier to develop and maintain, and which are specialized at solving the constituent subproblems. This decomposition allows each agent to employ the most appropriate paradigm for solving its particular problem, rather than being forced to adopt a common uniform approach that represents a compromise for the entire system, but which is not optimal for any of its subparts. The notion of an autonomous agent also provides a useful *abstraction* in just the same way that procedures, abstract data types, and, most recently, objects provide abstractions. They allow a software developer to conceptualize a complex software system as a society of cooperating autonomous problem solvers. For many applications, this high-level view is simply more appropriate than the alternatives.

1.2.1.3 Ubiquity

Despite the many innovations in human-computer interface design over the past two decades, and the wide availability of powerful window-based user interfaces, computer-naïve users still find most software difficult to use. One reason for this is that the user of a software product typically has to describe each and every step that needs to be performed to solve a problem, down to the smallest level of detail. If the power of current software applications is ever to be fully utilized by such users, then a fundamental rethink is needed about the nature of the interaction between computer and user. It must become an equal partnership – the machine should not just act as a dumb receptor of task descriptions, but should *cooperate* with the user to achieve their goal. As Negroponte wrote, ‘the future of computing will be 100% driven by delegating to, rather than manipulating computers’ (Negroponte, 1995). To deliver such functionality, software applications must be:

- *autonomous*: given a vague and imprecise specification, it must determine how the problem is best solved and then solve it, without constant guidance from the user,
- *proactive*: it should not wait to be told what to do next, rather it should make suggestions to the user,

- *responsive*: it should take account of changing user needs and changes in the task environment, and
- *adaptive*: it should come to know user's preferences and tailor interactions to reflect these.

In other words, it needs to behave as an intelligent agent. These considerations give rise to the idea of an agent acting as an 'expert assistant' with respect to some application, knowledgeable about both the application itself and the user, and capable of acting *with* the user in order to achieve the user's goals. We discuss some prototypical expert assistants in Section 1.3.

1.2.2 Improving the Efficiency of Software Development

Agent technology gives us the tools with which to build applications that we were previously unable to build. But it can also provide a better means of conceptualizing and/or implementing a given application. Here, three important domain characteristics are often cited as a rationale for adopting agent technology (cf. Bond and Gasser, 1988):

- data, control, expertise, or resources are inherently distributed,
- the system is naturally regarded as a society of autonomous cooperating components, or
- the system contains legacy components, which must be made to interact with other, possibly new software components.

1.2.2.1 Distribution of Data, Control, Expertise, or Resources

When the domain involves a number of distinct problem solving entities (or data sources) that are physically or logically distributed (in terms of their data, control, expertise, or resources), and which need to interact with one another in order to solve problems, then agents can often provide an effective solution. For example, in a distributed health care setting, general practitioners, hospital specialists, nurses, and home care organizations have to work together to provide the appropriate care to a sick patient (Huang et al., 1995). In this domain, there is:

- *distribution of data*: the general practitioner has data about the patient, which is very different from that of the hospital nurse – even though it concerns the same person,
- *distribution of control*: each individual is responsible for performing a different set of tasks,
- *distribution of expertise*: the specialist's knowledge is very different from that of either the general practitioner or the nurse, and
- *distribution of resources*: a specialist is responsible for the beds required by their patients, a general practitioner for paying hospital bills, etc.

In cases like this, agents provide a natural way of modeling the problem: real-world entities and their interactions can be directly mapped into autonomous problem solving agents with their own resources and expertise, and which are able to interact with others in order to get tasks done. Also, in the case of distributed data sources, (as in sensor networks (Lesser and Corkill, 1983) and seismic monitoring (Mason, 1995)), the use of agents means that significant amounts of processing can be carried out at the data source, with only high-level information exchanged. This alleviates the need to send large amounts of raw data to a distant central processor, thus making more efficient use of communications bandwidth.

1.2.2.2 Natural Metaphor

The notion of an autonomous agent is often the most appropriate metaphor for presenting a given software functionality. For example:

- a program that filters email can be presented to its user via the metaphor of a *personal digital assistant* (Maes, 1994), and
- meeting scheduling software can naturally be presented as an empowered, autonomous, social agent that can interact with other similar agents on the user's behalf.

In such applications, the fact that these functions are implemented through a series of local agents also means that they can be *personalized* to reflect the preferences of their user. Finally, in computer games (Wavish and Graham, 1995) and virtual reality systems (Bates, 1994), characters can naturally be represented as agents.

1.2.2.3 Legacy Systems

Large organizations have many software applications (especially information systems) that perform critical organizational functions. To keep pace with changing business needs, these systems must be periodically updated. However, modifying such *legacy systems* is in general very difficult: the system's structure and internal operation will become corrupted with the passage of time, designs and documentation are lost, and individuals with an understanding of the software move on. Completely rewriting such software tends to be prohibitively expensive, and is often simply impossible. Therefore, in the long term, the only way to keep such legacy systems useful is to incorporate them into a wider cooperating community, in which they can be exploited by other pieces of software. This can be done, for example, by building an 'agent wrapper' around the software to enable it to interoperate with other systems (Genesereth and Ketchpel, 1994; Jennings et al., 1993). The process of transforming a software application such as a database into an agent is sometimes referred to as *agentification* (Shoham, 1993).

1.2.3 The Limitations of Agent Solutions

Although agent technology has an important role to play in the development of leading-edge computing applications, it should not be oversold. Most applications that currently use agents could be built using non-agent techniques. Thus the mere fact that a particular problem domain has distributed data sources or involves legacy systems does not *necessarily* imply that an agent-based solution is the most appropriate one – or even that it is feasible. As with all system designs, the ultimate choice is dictated by many factors. This section has identified the types of situation in which an agent-based solution *should be considered*, as opposed to those in which it *should be deployed*. Moreover, it should be noted that the very nature of the agent paradigm leads to a number of problems, common to all agent- based applications:

- *No overall system controller*. An agent-based solution may not be appropriate for domains in which global constraints have to be maintained, in domains where a real-time response must be guaranteed, or in domains in which dead-locks or livelocks must be avoided.
- *No global perspective*. An agent's actions are, by definition, determined by that agent's local state. However, since in almost any realistic agent system, complete global knowledge is not a possibility, this may mean that agents make globally sub-optimal decisions. The issue of reconciling decision making based on local knowledge with the desire to achieve globally optimal performance is a basic issue in multi-agent systems research (Bond and Gasser, 1988).
- *Trust and delegation*. For individuals to be comfortable with the idea of delegating tasks to agents, they must first *trust* them. Both individuals and organizations will thus need to become more accustomed and confident with the notion of autonomous software components, if they are to become widely used. Users have to gain confidence in the agents that work on their behalf, and this process can take time. During this period, the agent must strike a balance between continually seeking guidance (and needlessly distracting the user) and never seeking guidance (and exceeding its authority). Put crudely, an agent must know its limitations.

1.3 Agent Application Domains

> *Agents are the next major computing paradigm and will be pervasive in every market by the year 2000.* (Janca, 1995)

There are several orthogonal dimensions along which agent applications could be classified. They can be classified by the type of the agent, by the technology used to implement the agent, or by the application domain itself. We choose to use the domain type, since this view fits best with the objectives and structure of the rest of this chapter (see Nwana (1996) for an alternative typology). Our aim in producing this classification scheme is simply to give a feel for the breadth and variety of

agent applications. More comprehensive descriptions of specific agent systems can be found in (AA97; Chaib-draa, 1995; Nwana, 1996; PAAM 1996; van Parunak, 1996).

1.3.1 Industrial Applications

Industrial applications of agent technology were among the first to be developed: as early as 1987, Parunak reports experience with applying the contract net task allocation protocol in a manufacturing environment (see below). Today, agents are being applied in a wide range of industrial applications.

1.3.1.1 Process Control

Process control is a natural application for intelligent agents and multi-agent systems, since process controllers are themselves autonomous reactive systems. It is not surprising, therefore, that a number of agent-based process control appli-cations should have been developed. The best known of these is ARCHON, a soft-ware platform for building multi-agent systems, and an associated methodology for building applications with this platform (Jennings, 1995). ARCHON has been applied in several process control applications, including electricity transportation management (the application is in use in northern Spain), and particle accelerator control. ARCHON also has the distinction of being one of the earliest field-tested multi-agent systems in the world. Agents in ARCHON are fairly heavyweight computational systems, with four main components: a *high-level communication module* (HLCM), which manages inter-agent communication; a *planning and coordination module* (PCM), which is essentially responsible for deciding what the agent will do; *an agent information management module* (AIM), which is responsible for maintaining the agent's model of the world, and finally, an *underlying intelligent system* (IS), which represents the agent's domain expertise. The HLCM, PCM, and AIM together constitute a kind of 'agent wrapper', which can be used to encapsulate an existing intelligent system (or indeed any pre-existing software application) and turn it into an agent.

1.3.1.2 Manufacturing

Parunak (1987) describes the YAMS (Yet Another Manufacturing System), which applies the well-known Contract Net protocol (Smith, 1980) to manufacturing control. The basic problem can be described as follows. A manufacturing enterprise is modeled as a hierarchy of *workcells*. There will, for example, be workcells for milling, lathing, grinding, painting, and so on. These workcells will be further grouped into flexible manufacturing systems (FMS), each of which will provide a functionality such as assembly, paint spraying, buffering of products, and so on. A collection of such FMSs is grouped into a factory. A single company or organization may have many different factories, though these factories may

duplicate functionality and capabilities. The goal of YAMS is to efficiently manage the production process at these plants. This process is defined by some constantly changing parameters, such as the products to be manufactured, available resources, time constraints, and so on. In order to achieve this enormously complex task, YAMS adopts a multi-agent approach, where each factory and factory component is represented as an agent. Each agent has a collection of plans, representing its capabilities. The contract net protocol allows tasks (i.e., production orders) to be delegated to individual factories, and from individual factories down to FMSs, and then to individual workcells. The contract net is based on the idea of negotiation, and hence YAMS views the problem of deciding how best to process a company's product manufacturing requirements as a negotiation problem. A some-what similar approach was developed in (Wooldridge et al., 1996), where the problem of determining an optimal production sequence for a factory was analyzed using the tools of game and negotiation theory.

1.3.1.3 Air Traffic Control

Kinny et al. (1996) describe a sophisticated agent-realized air traffic control system known as OASIS. In this system, which is undergoing field trials at Sydney airport in Australia, agents are used to represent both aircraft and the various air traffic control systems in operation. The agent metaphor thus provides a useful and natural way of modeling real-world autonomous components. As an aircraft enters Sydney airspace, an agent is allocated for it, and the agent is instantiated with the information and goals corresponding to the real-world aircraft. For example, an aircraft might have a goal to land on a certain runway at a certain time. Air traffic control agents are responsible for managing the system. OASIS is implemented using the AAII's own dMARS system. This system allows an agent to be implemented using the belief-desire-intention model of agency – one of the most popular approaches to reasoning about agents in theoretical multi-agent systems.

1.3.2 Commercial Applications

1.3.2.1 Information Management

As the richness and diversity of information available to us in our everyday lives has grown, so the need to manage this information has grown. The lack of effective information management tools has given rise to what is colloquially known as the information overload problem. Put simply, the sheer volume of information available to us via the Internet and World Wide Web (WWW) represents a very real problem. The potential of this resource is immediately apparent to anyone with more than the most superficial experience of using the WWW. But the reality is often disappointing. There are many reasons for this. Both human factors (such as users getting bored or distracted) and organizational factors (such as poorly organized pages with no semantic markup) conspire against users attempting to use

the resource in a systematic way. We can characterize the information overload problem in two ways:

- *Information filtering*. Every day, we are presented with enormous amounts of information (via email and newsnet news, for example), only a tiny proportion of which is relevant or important. We need to be able to sort the wheat from the chaff, and focus on information we *need*.
- *Information gathering*. The volume of information available prevents us from actually *finding* information to answer specific queries. We need to be able to obtain information that meets our requirements, even if this information can only be collected from a number of different sites.

One important contributing factor to information overload is almost certainly that an end user is required to constantly *direct* the management process. But there is in principle no reason why such searches should not be carried out by agents, acting autonomously to search the Web on behalf of some user. The idea is so compelling that many projects are directed at doing exactly this. Three such projects are described below:

- Maxims: (Maes, 1994) describes an electronic mail filtering agent called Maxims. The program 'learns to prioritize, delete, forward, sort, and archive mail messages on behalf of a user' (p. 35). It works by 'looking over the shoulder' of a user as he or she works with their email reading program, and uses every action the user performs as a lesson. Maxims constantly makes internal predictions about what a user will do with a message. If these predictions turn out to be inaccurate, then Maxims keeps them to itself. But when it finds it is having a useful degree of success in its predictions, it starts to make suggestions to the user about what to do.
- Newt**:** (Maes, 1994) also describes an example of an Internet news filtering program called Newt. This program, implemented in C++ on a UNIX plat-form, takes as input a stream of usenet news articles, and as output gives a subset of these articles that it is recommended the user reads. The Newt agent is 'programmed' by means of training examples: the user gives Newt examples of articles that would and would not be read, and this feedback is used to modify an internal 'model' (we use the term very loosely) of the user's interests. The user can, if desired, explicitly program Newt by giving it precise rules (e.g., "give me all articles containing the word 'agent'"). Newt retrieves words of interest from an article by performing a full-text analysis using the vector space model for documents.
- The Zuno Digital Library: A digital library is an organized, managed collection of data, together with services to assist the user in making use of this data. The Zuno Digital Library (ZDL) system is a multi-agent system that enables a user to obtain a single, coherent view of incoherent, disorganized data sources such as the World Wide Web, a user's own data, collections of articles on publishing house sites, and so on (Zuno, 1997). Agents in ZDL play one of three roles:

- *consumer* – representing end users of the system, who can be thought of as consuming information;
- *producer* – representing 'content providers', who own the information that customers consume; and
- *facilitator* – mapping between consumers and producers.

Consumer agents in the system are responsible for representing the user's interests. They maintain models of users, and use these models to assist them, by proactively providing information they require, and shielding them from information that is not of interest. ZDL thus acts both as an information *filter* and an information *gatherer*.

1.3.2.2 Electronic Commerce

Currently, commerce is almost entirely driven by human interactions; humans decide when to buy goods, how much they are willing to pay, and so on. But in principle, there is no reason why *some* commerce cannot be *automated*. By this, we mean that some commercial decision making can be placed in the hands of agents. Lest the reader suppose that this is fanciful, and that no sensible commercial organization would make their decision making (and hence money spending) the responsibility of a computer program, it should be remembered that this is precisely what happens today in the electronic trading of stocks and shares. Widespread electronic commerce is, however, likely to lie some distance in the future. In the near term, electronic trading applications are likely to be much more mundane and small scale. As an example, Chavez and Maes (1996) describe a simple 'electronic marketplace' called Kasbah. This system realizes the market-place by creating 'buying' and 'selling' agents for each good to be purchased or sold respectively. Commercial transactions take place by the interactions of these agents.

1.3.2.3 Business Process Management

Company managers make informed decisions based on a combination of judgement and information from many departments. Ideally, all relevant information should be brought together before judgement is exercised. However obtaining pertinent, consistent and up-to-date information across a large company is a complex and time consuming process. For this reason, organizations have sought to develop a number of IT systems to assist with various aspects of the manage-ment of their business processes. Project ADEPT (Jennings et al., 1996) tackles this problem by viewing a business process as a community of negotiating, service-providing agents. Each agent represents a distinct role or department in the enterprise and is capable of providing one or more services. For example, a design department may provide the service of designing a telecom network, a legal department may offer the service of checking that the design is legal, and the marketing department may provide the service of costing the design. Agents who

require a service from another agent enter into a negotiation for that service to obtain a mutually acceptable price, time, and degree of quality. Successful negotiations result in binding agreements between agents. This agent-based approach offers a number of advantages over more typical workflow solutions to this problem. The proactive nature of the agents means services can be scheduled in a just-in-time fashion (rather than pre-specified from the beginning), and the responsive nature of the agents means that service exceptions can be detected and handled in a flexible manner. The current version of the system has been tested on a British Telecom (BT) business process involving some 200 activities and nine departments and there are plans to move toward full scale field trials.

1.3.3 Medical Applications

Medical informatics is a major growth area in computer science: new applications are being found for computers every day in the health industry. It is not surprising, therefore, that agents should be applied in this domain. Two of the earliest applications are in the areas of health care and patient monitoring.

1.3.3.1 Patient Monitoring

The Guardian system described in (Hayes-Roth et al., 1989) is intended to help manage patient care in the Surgical Intensive Care Unit (SICU). The system was motivated by two concerns: first, that the patient care model in a SICU is essentially that of a team, where a collection of experts with distinct areas of expertise cooperate to organize patient health care; and second, that one of the most important factors in good SICU patient health care is the adequate sharing of information between members of the critical care team. In particular, specialists tend to have very little opportunity to monitor the minute-by-minute status of a patient; this task tends to fall to nurses, who, in contrast, often do not have the expertise to interpret the information they obtain in the way that an appropriate expert would. The Guardian system distributes the SICU patient monitoring function among a number of agents, of three different types:

- *perception/action agents* – responsible for the interface between Guardian and the world, mapping raw sensor input into a usable symbolic form, and translating action requests from Guardian into raw effector control commands,
- *reasoning agents* – responsible for organizing the system's decision making process, and
- *control agents* – of which there will only ever be one, with overall, top-level control of the system.

These agents are organized into hierarchies, and the system as a whole is closely based on the blackboard model of control, wherein different agents ('knowledge sources') cooperate via sharing knowledge in a common data structure known as a blackboard.

1.3.3.2 Health Care

Huang et al. (1996) describe a prototypical agent-based distributed medical care system. This system is designed to integrate the patient management process, which typically involves many individuals. For example, a general practitioner may suspect that a patient has breast cancer, but this suspicion cannot be con-firmed or rejected without the assistance of a hospital specialist. If the specialist confirms the hypothesis, then a care programme must be devised for treating the patient, involving the resources of other individuals. The prototype system allows a natural representation of this process, with agents mapped onto the individuals and, potentially, organizations involved in the patient care process. Agents in the prototype contain a knowledge-based (intelligent) system, containing the domain expertise of the agent, a human-computer interface, allowing the user to add, remove, or view system goals, and a communications manager, which realizes the message-passing functionality of agents. The intelligent system component is based on the KADS model of expertise, and the whole agent architecture is implemented in PROLOG. Message passing is realized via extensions to standard email.

1.3.4 Entertainment

The leisure industry is often not taken seriously by the computer science com-munity. Leisure applications are frequently seen as somehow peripheral to the 'serious' applications of computers. And yet leisure applications such as computer games can be extremely lucrative – consider the number of copies of id Software's 'Quake' sold since its release in 1996. Agents have an obvious role in computer games, interactive theater, and related virtual reality applications: such systems tend to be full of semi-autonomous animated characters, which can naturally be implemented as agents.

1.3.4.1 Games

Wavish et al. (1996) describe several applications of agent technology to com-puter games. For example, they have developed a version of the popular Tetris computer game, where a user must try to make a wall out of irregularly shaped falling blocks. The agent in the game takes the part of the user, who must control where the blocks fall. Trying to program this agent using traditional symbolic AI techniques would require going through a knowledge elicitation stage, repre-senting knowledge about the game and the role of the user in terms of symbolic data structures such as rules, and so on. This approach would be entirely un-realistic for a game like Tetris, which has hard real-time constraints. Wavish and colleagues thus use an alternative *reactive* agent model called RTA (Real Time Able). In this approach, agents are

programmed in terms of *behaviors*. These behaviors are simple structures, which loosely resemble rules but do not require complex symbolic reasoning.

1.3.4.2 Interactive Theater and Cinema

By interactive theater and cinema, we mean a system that allows a user to play out a role analogous to the roles played by real, human actors in plays or films, interacting with artificial, computer characters that have the behavioral characteristics of real people. Agents that play the part of humans in theater-style applications are often known as *believable* agents, a term coined by Joe Bates. His vision is of

> *[Agents that] provide the illusion of life, thus permitting … [an] audience's suspension of disbelief.* (Bates, 1994)

A number of projects have been set up to investigate the development of such agents. For example, (Hayes-Roth et al., 1995) describe a *directed improvization* system, in which human players act out roles in a dynamic narrative, creating a new, user directed 'work'.

1.3.5 Comments

At a certain level of abstraction, many of the above applications share common features. Individual agents are designed and built to enact particular roles. These agents are autonomous, goal directed entities, which are responsive to their environment. They must typically interact with other agents in order to carry out their role. Such interactions are a natural consequence of the inevitable interdependencies which exist between the agents, their environment, and their design objectives. As the agents are autonomous, the interactions are usually fairly sophisticated – involving cooperation, coordination and negotiation. Following on from this view, two important observations can be made about developing agent-based applications:

- The detailed problem solving actions of the agent can only be determined at run time. Individual behavior is regulated by a complex interplay between the agent's internal state and its external influences (its environment and the other agents). Thus the precise 'trajectory' that an agent will follow can only be discovered by running the agent in its environment.
- Because the behavior of individual agents is not uniquely determined at design time, the behavior of the system as a whole can also only emerge at run time.

These points have some obvious (and perhaps alarming) implications for the use of an agent-based approach in safety-critical application domains such as air traffic control, where it is essential that the system satisfies its specification. We comment on this issue in Section 1.4.

The above applications also illustrate that there are several different dimensions along which we can analyze agent-based systems:

- *Sophistication of the Agents*. Agents can be seen as performing three broad types of behavior. At the simplest level, there is the 'gopher' agent, which carries out comparatively simple tasks, based on well-defined, pre-specified rules and assumptions. The next level of sophistication involves 'service performing' agents, which carry out a high-level, but still well-defined task at the request of a user (e.g., arrange a meeting or find an appropriate flight). Finally, there are the predictive/proactive agents, capable of flexible autonomous behavior in the way we discussed above. Simple examples are agents that volunteer information or services to a user, without being asked, whenever it is deemed to be appropriate.
- *Role of the Agents*. Agents play many types of role. For example, in several of the industrial and commercial applications discussed above, the role of the agent system is to provide a decision support functionality. The agents act autonomously and proactively to gather information and to make recommendations, but a human operator makes the ultimate decisions. In contrast, in the entertainment domain (for example), the agent completely automates a problem solving role – it is thus delegated a particular activity and is entirely responsible for carrying it out.
- *Granularity of View*. In some of the applications discussed above, the significant unit of analysis and design is the *individual* agent, whereas in other applications it is the *society* of agents that is key. The decision about whether to adopt a single-agent or multi-agent approach is generally determined by the domain, and is similar in nature to decisions about whether monolithic, centralized solutions or distributed, decentralized solutions are appropriate. Single-agent systems are in a sense much simpler than multi-agent systems, since they do not require the designer to deal with issues such as cooperation, negotiation, and so on.

1.4 The Agent Development Bottleneck

At the time of writing, expertise in designing and building agent applications is scarce. There is little in the way of production-quality software support for building agent applications, and still less general understanding of the issues that need to be addressed when building such systems. Worse, most of today's agent systems are built from scratch, using bespoke tools and techniques, which cannot easily be applied to other types of system. This is particularly worrying because a lot of infrastructure is required before the main components of an agent system can be built. At the moment, most developers rebuild this infrastructure from scratch in every new case; this process is clearly not sustainable.

However, there is an increasing awareness that this situation must change if agents are to make it into the mainstream of software development (Kinny, 1996;

Wooldridge, 1997). Support is required throughout the agent system development process. In what follows, we use a traditional software engineering model as a basis for structuring our discussion of this issue. We begin our discussion with requirements specification, since the stage that precedes it – requirements analysis – does not differ in any significant way from that for other types of software system. The requirements analysis stage typically concludes with a *user requirements document* – a statement of the properties that the analyst or developer believes the user (client) wants of the new system. This document is usually written in non-technical language, so that it can be understood by both client and developer.

1.4.1 Requirements Specification

Once an understanding is reached of the client's needs with respect to the newly commissioned system, a precise statement is required of what the developers intend to build. This document tends to be written as formally as possible – it aims to be a precise statement of the properties of the system that the developers will build. A number of approaches have been developed to agent system requirements specification, and in particular to formal (mathematical) approaches (Wooldridge, 1997). The details of these formalisms tend to be rather complex, but broadly speaking, they characterize agents as *rational decision makers*. By this, we mean systems whose internal state can be expressed in terms of 'mentalistic' constructs such as belief, desire, and intention. An agent's decision making can then be characterized in terms of these constructs. It may be difficult to see how one might specify a computer system in this way, but the idea is really quite straightforward. One builds a specification out of a set of 'rules' (strictly speaking, logical formulas), somewhat like this:

if	*agent 1 believes that agent 2 believes runway 1 is clear*
then	*agent 1 should believe that agent 2 has erroneous information,* and *agent 1 should intend that agent 2 is informed of its error.*

One might wonder why terms such as 'believe' and 'intend' are used. Beliefs are generally used to refer to the information that agents have about their environment. This information can be incorrect – in just the same way that information we have about the environment (our beliefs) could be wrong. The term intention is used to refer to a goal that the agent will pursue until it either succeeds or fails completely – it admits the possibility of failure, a realistic possibility in many complex systems, one which traditional formalisms tend to ignore. The chapter by Georgeff and Rao is the clearest expression of this view in this volume. The reader is also referred to (Wooldridge, 1997) for a detailed discussion on these approaches.

1.4.2 System Design

The design of robust, efficient systems is one of the most important issues addressed by software engineers, and a great deal of effort has been devoted to making system design a methodical, rigorous process. At the time of writing, comparatively few workers in the agent research community have considered how design methodologies might be applied to agent systems. This situation must clearly change if agent systems are to become widely used, and in particular, the following key questions need to be addressed:

- *Adopting an agent-based approach.* In much of the work that goes under the agent banner, the rationale behind the choice of an agent-based approach (over other, arguably simpler alternatives) is unclear. The analysis of appropriate domain properties (Sections 1.2 and 1.3) represents a step toward this end.
- *Macro-system Structure.* Once given an abstract, high-level system specification, we need to transform this specification into agents, and relate these agents to one another in a system structure. This process is known as *refinement.* In just the same way that object-oriented design methodologies exist, which give guidelines relating to the identification of objects and object interdependencies, so we might expect soon to see *agent-oriented design methodologies*, which offer clear guidelines about how to decompose a problem into agents and what effects a given decomposition is likely to have on the system performance. The design problem is exacerbated by the fact that legacy systems often need to be incorporated in agent systems. Thus design needs to be done both from a top-down perspective (how should the problem be decomposed ideally?) and from a bottom-up perspective (what software groupings already exist and cannot be changed?). Both perspectives must be accommodated.

1.4.3 System Implementation

An agent system design will both describe the various different roles that exist within the system and characterize the relationships that exist between these roles. However, the design will not generally prescribe an implementation of these agents. So, having identified the various agent roles in a system, the next step is to determine how each of these roles can be best realized. An agent architecture needs to be devised or adopted for each role, which will deliver the required functional and non-functional characteristics of the role.

Many agent architectures have been developed by the intelligent agents community, with many different properties (Wooldridge and Jennings, 1995). At one extreme, there are ‘strong AI’ systems, which allow users to build agents as knowledge-based systems, or even as logic theorem provers. In order to build agents using such systems, one goes through the standard knowledge-based system process of knowledge elicitation and representation, coding an agent’s behavior in terms of rules, frames, or semantic nets. At the other extreme, there are many agent frameworks that simply offer enhanced versions of (for example) the Java

programming language – they include no AI techniques at all. Neither of these extremes is strictly right or wrong: they both have merits and drawbacks. In general, of course, the simplest solution that will effectively solve a problem is often the best. There is little point in using complex reasoning systems where simple Java-like agents will do. Obviously, more detailed guidelines to assist with this decision making process are desirable.

1.4.4 System Testing, Debugging, and Verification

As long ago as 1987, Les Gasser wrote:

> *Concurrency, problem-domain uncertainty, and non-determinism in execution together conspire to make it very difficult to comprehend the activity in a distributed intelligent system […] we urgently need graphic displays of system activity linked to intelligent model-based tools which help a developer reason about expected and observed behavior.* (Gasser, 1987)

Gasser's comments remain as true of agent system development in the late 1990s as they did of the mid-1980s. Sadly, testing and debugging are still much neglected areas of the agent development process. Developers need assistance with *visualizing* what is happening, and need *debugging* facilities to step through the execution and amend behavior where appropriate. Visualization is particularly important – determining what is happening in asynchronous, concurrent systems is an exceedingly difficult task. This is true both within an agent and even more so between agents. Traditional debugging features which allow execution to be stopped, internal states to be examined, states to be changed, message arrivals to be simulated, and so on are also essential. In most systems these features and tools are developed from scratch – clearly an undesirable state of affairs.

1.5 The Structure of This Book

The chapters in this book cover a wide spectrum of issues related to the applications of intelligent agents and multi-agent systems. There are introductory chapters that explain the basic concepts and summarize the state of the art, there are vision chapters which look to the future of agent applications (from both a technical and a marketing perspective), and, finally, there are experience chapters, which deal with specific agent systems and applications.

In more detail:

- Technological issues that are addressed include: architectures for individual agents and multi-agent systems; frameworks for implementing agent applications; methodologies for designing agents and multi-agent systems; techniques for attaining efficient and coherent communication, cooperation and

coordination; and techniques for personalizing agents to reflect the needs of individual users.

- Predictions are made about the following issues: the types of markets where agents are likely to have the highest impact; the economic value of various agent markets; the time scales by which agent technology will enter various aspects of mainstream software solutions; and the (technical and non-technical) obstacles that need to be overcome if agents are to achieve their potential.
- Application domains for which agent systems are described include: telecommunications systems, personal digital assistants, information management, information economies, business applications, air traffic control, computer simulation, transportation management, and financial management.

The range of author affiliations (British Telecom, US West, IBM, GPT, Sharp, Daimler-Benz, Siemens, GTE Labs, France Telecom, US Treasury, and Swiss Bank Corporation) covers a significant proportion of the large commercial organizations that are currently investing in agent technology, and illustrates forcefully the pervasive nature of this exciting new technology.

1.5.1 Introductory Chapters

These two chapters introduce the basic terms and concepts of agent systems. They provide the underpinning for the remainder of this book and offer a good point of entry to those who are new to the field.

Nwana and Ndumu provide a broadly based and up-to-date introduction to the basic concepts and terminology of agent systems. They define a typology that specifies the range of agents which currently exist, offer an assessment of the key problems and promises of the different agent types, and describe a number of exemplar systems.

Laufmann takes a complementary approach to that of Nwana and Ndumu. He identifies and specifies an ideal intelligent agent, and shows how the current agent landscape can be divided by considering extant systems as highlighting particular attributes of this ideal model. He espouses three different, but related, views of agents – agents as automated personal assistants, agents as cooperating problem solvers, and agents as communicating software entities. He then concentrates on the third view and outlines a strategy for software development that should lead to substantially improved agent applications in the near term.

1.5.2 Vision Chapters

These four chapters assess where agents are today and where they are likely to be in five to ten years time. When taken together, the chapters convey the excitement and potential of this field and highlight the key open problems.

Janca presents an analysis of the current state of agent-enabled applications and describes how this is likely to change in the coming years. He identifies two broad classes of agents – user interface agents which act on behalf of the user (capturing their goals and translating these into actions) and process agents which are responsible for enacting these goals. Based on an analysis of the current market for agent technology (see also the contribution by Guilfoyle) he notes that the current generation of applications exist in self-contained niches, that the agents use proprietary formats, and that there are no standards for interconnection (see contributions by Huhns and Singh and Kearney). To circumvent these short-comings, he presents an agent design model which acts as a unifying framework for agent applications and which can have standard re-useable parts. In this view, building different applications becomes more of a 'plug-and-play' than a 'start from scratch' endeavor. Finally, he presents a series of commercial challenges that need to be overcome before agents can reach their full potential.

Guilfoyle presents an analysis of the market for agent technology (also see Janca's contribution). She identifies the main areas in which agent technology is likely to have an impact and the main vendors that are currently active in this field. She assesses the current trends of agent applications and offers various predictions about how the size and direction of the market will expand over time.

Foss concentrates on the concept of information brokerage (one of the key agent application domains identified by Janca and Guilfoyle). He outlines the role of intelligent agents as an intermediary service between consumers and infor-mation sources, and presents a vision of a future commercial information trading environment which is rich enough for true electronic commerce to take place. This view includes dealing adequately with the currently neglected issues of taxation, rights, royalties, and revenues. A case is then made for the pivotal role of agent technology in this vision (see Plu's contribution for an indication of how such a vision can begin to be realized using current generation technology).

Kearney concentrates solely on one class of agents – personal digital assistants (PDAs). Again, this is an important current and future market for agent technology as identified by Janca and Guilfoyle. For this application class, he posits a role for agent software which involves mediating between the individual user and a range of online services (see also the contributions of Foss and Janca). He suggests a path from the current state of the art through successive generations of PDAs to a state in which agents are ubiquitous providers of personalized services. For this vision to be attained, he identifies agent interoperability as the key problem to be solved (see also the contribution by Huhns and Singh).

1.5.3 Systems and Their Applications

The chapters in this section represent the current state of the art in applied agent systems. They outline the types of agent frameworks and architectures that are

currently being used to develop and implement agent systems and discuss, in detail, a wide range of agent applications built using this technology.

Georgeff and Rao present results related to the design and implementation of rational software agents. In this work, they address one of the fundamental problems of agent research – the gap between theory and practice. This gap is currently large and both sides of the field plough along in isolation. However, in their work, the authors use a formal framework to specify and implement their agents. The particular class of agents they address are Belief-Desire-Intention (BDI) agents (also see the contribution by Burmeister et al.). Thus, they sketch the theoretical foundation of their agents and how they are realized in a practical agent implementation. They then discuss how this implementation was used to build large-scale, real-world agent applications in the domains of air traffic management, business process management, and air combat modeling.

Burmeister et al. discuss how agent based techniques can be applied in the domains of traffic management and manufacturing. They present their core technology for realizing their agent applications: COSY, a modular agent architecture based on the concepts of behaviors, resources, and intentions (see also the contribution by Georgeff and Rao); and DASEDIS, a development environment for building such agents (cf. the development environment of Haugeneder and Steiner). The application of this architecture to two traffic applications (traffic management and freight logistics) and two manufacturing applications (flexible manufacturing and plant control) is then discussed.

Haugeneder and Steiner present a conceptual design and specification of an agent architecture and a multi-agent language (called MAIL) for implementing single agents and cooperative interaction among them. This design is then refined into a multi-agent environment for constructing cooperative applications based on this model (cf. the DASEDIS framework of Burmeister et al.). Finally, the architecture and the environment are demonstrated on two sample applications in the areas of group scheduling and road traffic management (see also the contribution of Burmeister et al.).

Weihmayer and Velthuijsen provide an overview of the role of agent systems in the domain of telecommunications. They identify the types of telecommuni-cations applications for which agent solutions have been constructed. These include process support (see also contributions by Huhns and Singh and Janca), network control, service management (see also Plu's contribution), network management, transmission switching, and network design. They then discuss the justification for the intense interest on the part of telecommunications organizations in agent technology, present a number of detailed system case studies, and, finally, offer a balanced analysis of the future of agents in telecommunications.

Huhns and Singh describe how a set of autonomous agents can cooperate to provide coherent management of transaction workflows. In particular, they concentrate on the problem of how agents with diverse and heterogeneous

information models can interoperate with one another (cf. the approach advocated by Kearney). Their approach to such interoperation is predicated on the fact that the agents maintain models of their information and resources. These models are then mapped into a common ontology to ensure coherent semantics among the cooperating group. Their approach is exemplified by considering a multi-agent system for telecommunication service provisioning.

Plu presents an agent-based view of how organizations can design and build computer infrastructures that provide sophisticated information and electronic commerce services to many customers. Using the ISO standard architecture for open distributed processing, he shows, with the aid of scenarios, the central role that agent technology can play at many of the model's distinct levels. He also discusses issues related to mobile agents – identifying when they are appropriate and the types of advantages they confer over their static counterparts.

Sycara et al. present the design and implementation of a series of agents for the tasks of filtering, evaluating, and integrating information on the Internet. Their agents team up, on demand, depending on the user, the task, and the situation, to retrieve and filter information. The agents and their organizational structure also adapt to the prevailing task and user. Their system is demonstrated in the domain of financial portfolio management (see also Wenger and Probst's contribution).

Goldberg and Senator detail the FAIS system, which links and evaluates reports of large cash transactions to identify potential money laundering activities. The system, which has been operational since 1993, is designed and implemented as a cooperative endeavor involving humans and software agents (see Haugeneder and Steiner's contribution for further thoughts on such human-computer cooperation). The main focus of the presently implemented system is knowledge discovery and information analysis in large data spaces. However, as the system's range of applicability has increased, so the need for greater agent-to-agent cooperation has emerged. To this end, the authors describe the next generation of FAIS and discuss how adopting an agent-based perspective eases the system migration task.

Wenger and Probst analyse the potential role and impact of the agent paradigm in the general area of providing financial services. They highlight the characteristics of typical agent-based applications and provide illustrative scenarios for the domains of mortgage sales, corporate financial management, and portfolio management (see also the contribution by Sycara et al.). They then discuss the various ways in which intelligent agents can add value to financial operations, present a detailed scenario based on an investment sales transaction service, and assess the future role of agent technology in this market segment.

References

AA97 (1997) *Proceedings of the First International Conference on Autonomous Agents'97*, Marina del Rey, CA. ACM Press.

Bates, J. (1994) The role of emotion in believable agents. *Communications of the ACM*, 37(7), 122–125.

Bond, A. H., Gasser, L. (Eds.) (1988*) Readings in Distributed Artificial Intelligence.* Morgan Kaufmann.

Chaib-draa, B., (1995) Industrial applications of distributed AI. *Communications of the ACM*, 38(11), 47–53.

Chavez, A., Maes, P. (1996) Kasbah: An agent marketplace for buying and selling goods. *Proceedings of First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Systems*, London, UK.

Crabtree, B., Jennings, N. R. (Eds.) (1996*) Proceedings of First International Conference on the Practical Application of Intelligent Agents and Multi-agent Systems*, London, UK.

Franklin, S., Graesser, A. (1996) Is it an agent, or just a program*? Proceedings Third International Workshop on Agent Theories, Architectures and Languages*, Budapest, Hungary, 193–206.

Gasser, L., Braganza, C., Herman, N. (1987) MACE: A flexible testbed for distributed AI research. In: M. N. Huhns (Ed.) *Distributed AI*. Morgan Kaufmann.

Genesereth, M. R., Ketchpel, S. P. (1994) Software agents. *Communications of the ACM*, 37(7), 48–53.

Hayes-Roth, B., Hewett, M., Waashington, R., Hewett, R., Seiver, A. (1995) Distributing intelligence within an individual. In: L. Gasser, M. N. Huhns (Eds.) *Distributed AI*, Volume II, 385–412. Morgan Kaufmann.

Huang, J., Jennings, N. R., Fox, J. (1995) An agent-based approach to health care management. *Int. Journal of Applied Artificial Intelligence*, 9(4), 401–420.

Janca, P. C. (1995) Pragmatic application of information agents. BIS Strategic Report.

Jennings, N. R., Varga, L. Z., Aarnts, R. P., Fuchs, J., Skarek, P. (1993) Transforming stand-alone expert systems into a community of cooperating agents. *Int. Journal of Engineering Applications of Artificial Intelligence*, 6(4), 317–331.

Jennings, N. R., Corera, J. M., Laresgoiti, I. (1995) Developing industrial multi-agent systems. In: *Proceedings of the First International Conference on Multi-agent Systems*, (ICMAS-95), 423-430.

Jennings, N. R., Faratin, P., Johnson, M. J., Norman, T. J., O'Brien, P., Wiegand, M. E. (1996) Agent-based business process management. *Int. Journal of Cooperative Information Systems*, 5(2, 3), 105–130.

Lesser, V. R., Corkill, D. D. (1983) The distributed vehicle monitoring testbed: a tool for investigating distributed problem solving network. *AI Magazine*, Fall, 15–33.

Maes, P. (1994) Agents that reduce work and information overload. *Communications of the ACM*, 37(7), 31–40.

Mason, C. L. (1995) Cooperative interpretation of seismic data for nuclear test ban treaty verification: a DAI approach. *Int. Journal of Applied Artificial Intelligence*, 9(4), 371–400.

Negroponte, N. (1995*) Being Digital*. Hodder and Stoughton.

Ovum Report (1994) Intelligent agents: the new revolution in software.

PAAM'97 (1997) *Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-agent Systems*, London.

Parunak, H. V. D. (1987) Manufacturing experience with the contract net. In: M. N. Huhns (Ed.) *Distributed AI*. Morgan Kaufmann.

Parunak, H. V. D. (1996) Applications of distributed artificial intelligence in industry. In: G. M. P. O'Hare, N. R. Jennings (Eds.) *Foundations of Distributed Artificial Intelligence*, 139–164. Wiley.

Pnueli, A. (1986) Specification and development of reactive systems. In: *Information Processing 86*, Elsevier/North-Holland.

Sargent, P. (1992) Back to school for a brand new ABC. In: *The Guardian*, 12 March, p. 28.

Shoham, Y. (1993) Agent-oriented programming. *Artificial Intelligence*, 60(1), 51-92.

Smith, R. (1980) The contract net protocol. *IEEE Trans. on Computers*, C-29(12), 1104–1113.

Wavish, P., Graham, M. (1996) A situated action approach to implementing characters in computer games. *Int. Journal of Applied Artificial Intelligence*, 10(1), 53–74.

Wooldridge, M., Jennings, N. R. (1995) Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2), 115–152.

Wooldridge, M. (1997) Agent based software engineering. *IEE Proceedings on Software Engineering*, 144(1), 26–37.

Wooldridge, M., Bussmann, S., Klosterberg, M. (1996) Production sequencing as negotiation. In: *Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-agent Systems*, London.

Zuno Ltd (1997) See http://www.dlib.com/.