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,

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.