Handbook of Software Engineering and Knowledge Engineering Vol. 0, No. 0 (1999) 000–000

AGENT-ORIENTED SOFTWARE ENGINEERING

Michael Wooldridge
Dept of Computer Science, University of Liverpool, Liverpool L69 7ZF, UK

Paolo Ciancarini
Dipartimento di Scienze dell'Informazione, University of Bologna, 47127 Bologna, Italy

Software and knowledge engineers continually strive to develop tools and techniques to manage the complexity that is inherent in the systems they have to build. In this article, we argue that *intelligent agents* and *agent-based systems* offer novel opportunities for developing effective tools and techniques. Following a discussion on the classic subject of what makes software complex, we introduce intelligent agents as software structures capable of making "rational decisions". Such rational decision-makers are well-suited to the construction of certain types of software, which mainstream software engineering has had little success with. We then go on to examine a number of prototype techniques proposed for engineering agent systems, including formal specification and verification methods for agent systems, and techniques for implementing agent specifications.

Keywords: autonomous agents, multi-agent systems.

1. Introduction

Over the past three decades, software engineers have derived a progressively better understanding of the characteristics of complexity in software. It is now widely recognised that *interaction* is probably the most important single characteristic of complex software (see, e.g., [21]). Software architectures that contain many network-aware, dynamically interacting components, each with their own thread of control, and engaging in complex coordination protocols to get or offer a plethora of services to other components, are typically orders of magnitude more complex to correctly and efficiently engineer than those that simply compute a function of some input through a single thread of control.

Unfortunately, it turns out that many (if not most) real-world applications have precisely these characteristics. As a consequence, a major research topic in Computer Science over at least the past two decades has been the development of tools and techniques to model, understand, and implement systems in which interaction is the norm. The advent of global computing platforms, like the Internet and the World Wide Web, has only increased the requirement of designing systems including complex interactions.

Many researchers now believe that in future, computation itself will be understood as chiefly as a process of interaction [23]. This has in turn led to the search for new computational abstractions, models, and tools with which to conceptualise and implement interacting systems.

Since the 1980s, software agents and multi-agent systems have grown into what is now one of the most active areas of research and development activity in computing generally. There are many reasons for the current intensity of interest, but certainly one of the most important is that the concept of an agent as an autonomous system, capable of interacting with other agents in order to satisfy its design objectives, is

a natural one for software designers. Just as we can understand many systems as being composed of essentially passive objects, which have state, and upon which we can perform operations, so we can understand many others as being made up of interacting, semi-autonomous agents which offer services.

This paper has the following structure: Section defines what we mean by the term "agent", and summarises why such agents might be appropriate for engineering certain complex software systems. We then describe some typical application domains for multi-agent systems. In section , we describe agent-oriented specification techniques, focussing in particular on the requirements that an agent-oriented specification framework will have. In section , we discuss how such specifications can be implemented, either by directly executing them, or else by automatically synthesising executable systems from specifications. Section discusses how implemented systems may be verified, to determine whether or not they satisfy their specifications. Finally, in section , we conclude with some comments on future issues for agent-oriented software engineering.

Note that sections through to include some material from [55], where a fuller examination of, in particular, the specification, implementation, and verification of agent-based systems may be found.

2. Agent-Based Systems

By an *agent-based system*, we mean one in which the key abstraction used is that of an *agent*. By an *agent*, we mean an abstraction that enjoys the following properties [58, pp116–118]:

- *autonomy*: agents encapsulate some state (which is not accessible to other agents), and make decisions about what to do based on this state, without the direct intervention of humans or others;
- *reactivity*: agents are *situated* in an environment, (which may be the physical world, a user via a graphical user interface, a collection of other agents, the Internet, or perhaps many of these combined), are able to *perceive* this environment (through the use of potentially imperfect sensors), and are able to respond in a timely fashion to changes that occur in it;
- *pro-activeness*: agents do not simply act in response to their environment, they are able to exhibit goal-directed behaviour by *taking the initiative*;
- *social ability*: agents interact with other agents (and possibly humans) via some kind of *agent-communication language*, and typically have the ability to engage in social activities (such as cooperative problem solving or negotiation) in order to achieve their goals.

A *multi-agent system* is a system composed of a number of such agents, which typically interact with one-another in order to satisfy their goals.

An obvious question to ask is why agents and multi-agent systems are seen as an important new direction in software engineering. There are several reasons [27, pp6–10]:

- *Natural metaphor.*

  Just as the many domains can be conceived of consisting of a number of interacting but essentially passive *objects*, so many others can be conceived as interacting, active, purposeful *agents*. For example, a scenario currently driving much research and development activity in the agents field is that of software agents that buy and sell goods via the Internet on behalf of some users. It is natural to view the software participants in such transactions as (semi-)autonomous agents.

- *Distribution of data or control.*

  For many software systems, it is not possible to identify a single locus of control: instead, overall control of the systems is distributed across a number computing nodes, which are frequently geographically distributed. In order to make such systems work effectively, these nodes must be capable of autonomously interacting with each other — they must agents.

- *Legacy systems.*

  A natural way of incorporating legacy systems into modern distributed information systems is to *agentify* them: to "wrap" them with an agent layer, that will enable them to interact with other agents.

- *Open systems.*

  Many systems are *open* in the sense that it is impossible to know at design time exactly what components or services the system will be comprised of, and how they will be able to interact with one-another. To operate effectively in such systems, the ability to engage in flexible autonomous decision-making is critical. An important example of this kind of systems are middleware platforms like OMG's CORBA [40] or Sun's Jini [38]. These platforms include some concept of agenthood, which helps in designing some specific types of component or service.

Now that we have defined what an agent is, we can look at some example applications of agent technology.

3. Some Applications of Agent Technology

Agents have been applied in several application domains, amongst the most important of which have been the following.

Agents and Distributed Systems In distributed systems, the idea of an agent is often seen as a natural metaphor, and, by some, as a development of the concurrent object programming paradigm [1]. Specifically, multi-agent systems have been applied in the following domains:

- *Air traffic control.*

  Air-traffic control systems are among the oldest application areas in multi-agent systems [49, 15]. A recent example is OASIS (*O*ptimal *A*ircraft *S*equencing using *I*ntelligent *S*cheduling), a system that is currently undergoing field trials at Sydney airport in Australia [33]. The specific aim of OASIS is to assist an air-traffic controller in managing the flow of aircraft at an airport: it offers estimates of aircraft arrival times, monitors aircraft progress against previously derived estimates, informs the air-traffic controller of any errors, and perhaps most importantly, finds the optimal sequence in which to land aircraft. OASIS contains two types of agents: *global* agents, which perform generic domain functions (for example, there is a "sequencer agent", which is responsible for arranging aircraft into a least-cost sequence), and *aircraft agents*, one for each aircraft in the system airspace.

- *Business process management.*

  Workflow and business process control systems are an area of increasing importance in computer science. Workflow systems aim to automate the processes of a business, ensuring that different business tasks are expedited by the

appropriate people at the right time, typically ensuring that a particular document flow is maintained and managed within an organisation. The ADEPT system is a current example of an agent-based business process management system [25, 26]. In ADEPT, a business organisation is modelled as a society of negotiating, service providing agents. ADEPT is currently being tested on a British Telecom (BT) business process which involves some nine departments and 200 different tasks.

- *Industrial systems management.*

  The largest and probably best-known European multi-agent system development project to date was ARCHON [24]. This project developed and deployed multi-agent technology in several industrial domains. The most significant of these domains was a power distribution system, which was installed and is currently operational in northern Spain. Agents in ARCHON have two main parts: a *domain* component, which realises the domain-specific functionality of the agent, and a *wrapper* component, which provides the agent functionality, enabling the system to plan its actions, and to represent and communicate with other agents. The ARCHON technology has subsequently been deployed in several other domains, including particle accelerator control.

- *Distributed sensing.*

  The classic application of multi-agent technology was in distributed sensing [32, 11]. The broad idea is to have a system constructed as a network of spatially distributed sensors. The sensors may, for example, be acoustic sensors on a battlefield, or radars distributed across some airspace. The global goal of the system is to monitor and track all vehicles that pass within range of the sensors. This task can be made simpler if the sensor nodes in the network *cooperate* with one-another, for example by exchanging predictions about when a vehicle will pass from the region of one sensor to the region of another. This apparently simple domain has yielded surprising richness as an environment for experimentation into multi-agent systems: Lesser's well known *Distributed Vehicle Monitoring Testbed* (DVMT) provided the proving ground for many of today's multi-agent system development techniques [32].

- *Space shuttle fault diagnosis.*

  It is difficult to imagine a domain with harder real-time constraints than that of in-flight diagnosis of faults on a spacecraft. Yet one of the earliest applications of the PRS architecture was precisely this [18]. In brief, the procedures that an astronaut would use to diagnose faults in the space shuttle's reaction control systems were directly coded as PRS plans, and the PRS architecture was used to interpret these plans, and provide real-time advice to astronauts in the event of failure or malfunction in this system.

- *Factory process control.*

  Organisations can be modelled as societies of interacting agents. Factories are no exception, and an agent-based approach to modelling and managing factories has been taken up by several researchers. This work began largely with Parunak [50], who, in YAMS (*Y*et *A*nother *M*anufacturing *S*ystem) used the Contract Net protocol [48] for manufacturing control. More recently, Mori *et al* have used a multi-agent approach to controlling steel coil processing plant [36], and Wooldridge *et al* have described how the process of determining an optimal production sequence for some factory can naturally be viewed as a problem of negotiation between the various production cells within the factory [57].

Agents in the Internet Much of the hyperbole that currently surrounds all things agent-like is related to the phenomenal growth of the Internet [12, 5]. In particular, there is a lot of interest in *mobile* agents, that can move themselves around the Internet operating on a user's behalf. This kind of functionality is achieved in the TELESCRIPT language developed by General Magic, Inc., for *remote programming* [52]; related functionality is provided in languages such as Java. There are a number of rationales for this type of agent:

- *Electronic commerce.*

  Currently, commercial activity is driven primarily by humans making decisions about what goods to buy at what price, and so on. However, it is not difficult to see that certain types of commerce might usefully be automated. A standard motivating example is that of a "travel agent". Suppose I want to travel from Manchester to San Francisco. There are many different airlines, price structures and routes that I could choose for such a journey. I may not mind about the route, as long as the aircraft involved is not "fly-by-wire"; I may insist on a dietary option not available with some airlines; or I may not want to fly with Ruritanian airlines after I had a bad experience once. Trying to find the best flight *manually* given these preferences is a tedious business, but a fairly straightforward one. It seems entirely plausible that this kind of service will in future be provided by agents, who take a specification of your desired flight and preferences, and, after checking through a range of on-line flight information databases, will return with a list of the best options.

- *Hand-held PDAs with limited bandwidth.*

  Hand-held "personal digital assistants" are seen by many as a next step in the laptop computer market. Such PDAs are often provided with limited-bandwidth links to telecommunications networks. If a PDA has a query that needs to be resolved, that will require network information resources, it may be more efficient to send out an agent across the network whose purpose is to resolve this query remotely. The searching process is done by the agent at a remote site, and only the final result of the query need be sent back to the PDA that originated the query.

- *Information gathering.*

  The widespread provision of distributed, semi-structured information resources such as the world-wide web obviously presents enormous potential; but it also presents a number of difficulties, (such as "information overload"); agents are seen as a natural tool to perform tasks such as searching distributed information resources, and filtering out unwanted news and email [34, 31].

At the time of writing, most interest in mobile agents is centred around the Java programming language, which, in the form of applets (portable downloadable programs embedded within WWW pages), already provides a very widely used mobile object framework. Also of relevance is the work of the Object Management Group (OMG), a consortium of computer manufacturers who are developing, amongst other things, a mobile agent framework based on their well known CORBA (*C*ommon *O*bject *R*equest *B*roker *A*rchitecture) distributed object standard [40].

Agents in Interfaces Another area of much current interest is the use of agent in *interfaces*. The idea here is that of the agent as an *assistant* to a user in some task. The rationale is that current interfaces are in no sense *pro-active*: things only happen when some user initiates a task. The idea of an agent acting in the way that a good assistant would, by *anticipating* our requirements, seems very

attractive. Nicholas Negroponte, director of the MIT Media Lab, sees the ultimate development of such agents as follows [37]:

> "The 'agent' answers the phone, recognizes the callers, disturbs you when appropriate, and may even tell a white lie on your behalf. The same agent is well trained in timing, versed in finding opportune moments, and respectful of idiosyncrasies." (p150)
>
> "If you have somebody who knows you well and shares much of your information, that person can act on your behalf very effectively. If your secretary falls ill, it would make no difference if the temping agency could send you Albert Einstein. This issue is not about IQ. It is shared knowledge and the practice of using it in your best interests." (p151)
>
> "Like an army commander sending a scout ahead ... you will dispatch agents to collect information on your behalf. Agents will dispatch agents. The process multiplies. But [this process] started at the interface where you delegated your desires." (p158)

Some prototypical interface agents of this type are described in [34].

In the remainder of this article, we consider what it means to *specify*, *implement*, and *verify* agent-based systems.

4. Specification

In this section, we consider the problem of *specifying* an agent system. What are the requirements for an agent specification framework? What sort of properties must it be capable of representing? The predominant approach to specifying agents has involved treating them as *intentional systems* that may be understood by attributing to them *mental states* such as beliefs, desires, and intentions [9, 58]; see [56] for a detailed justification of this idea. Using this idea, a number of approaches for formally specifying agents have been developed, which are capable of representing the following aspects of an agent-based system:

- the *beliefs* that agents have — the information they have about their environment, which may be incomplete or incorrect;
- the *goals* that agents will try to achieve;
- the *actions* that agents perform and the effects of these actions;
- the *ongoing interaction* that agents have — how agents interact with each other and their environment over time.

We use the term *agent theory* to refer to a theory which explains how these aspects of agency interact to generate the behaviour of an agent. The most successful approach to (formal) agent theory appears to be the use of a *temporal modal logic* (space restrictions prevent a detailed technical discussion on such logics — see, e.g., [58] for extensive references). Two of the best known such logical frameworks are the Cohen-Levesque theory of intention [8], and the Rao-Georgeff belief-desire-intention model [43, 56]. The Cohen-Levesque model takes as primitive just two attitudes: beliefs and goals. Other attitudes (in particular, the notion of *intention*) are built up from these. In contrast, Rao-Georgeff take intentions as primitives, in addition to beliefs and goals. The key technical problem faced by agent theorists is developing a formal model that gives a good account of the interrelationships between the various attitudes that together comprise an agents internal state [58]. Comparatively few serious attempts have been made to specify real agent systems using such logics — see, e.g., [17] for one such attempt.

5. Implementation

Once given a specification, we must implement a system that is correct with respect to this specification. The next issue we consider is this move from abstract specification to concrete computational system. There are at least two possibilities for achieving this transformation that we consider here:

1. somehow directly execute or animate the abstract specification; or
2. somehow translate or compile the specification into a concrete computational form using an automatic translation technique.

In the sub-sections that follow, we shall investigate each of these possibilities in turn.

5.1. Directly Executing Agent Specifications

Suppose we are given a system specification, ϕ, which is expressed in some logical language L. One way of obtaining a concrete system from ϕ is to treat it as an *executable specification*, and *interpret* the specification directly in order to generate the agent's behaviour. Interpreting an agent specification can be viewed as a kind of constructive proof of satisfiability, whereby we show that the specification ϕ is satisfiable by *building a model* (in the logical sense) for it. If models for the specification language L can be given a computational interpretation, then model building can be viewed as executing the specification. To make this discussion concrete, consider the Concurrent METATEM programming language [16]. In this language, agents are programmed by giving them a temporal logic specification of the behaviour it is intended they should exhibit; this specification is directly executed to generate each agent's behaviour. Models for the temporal logic in which Concurrent METATEM agents are specified are linear discrete sequences of states: executing a Concurrent METATEM agent specification is thus a process of constructing such a sequence of states. Since such state sequences can be viewed as the histories traced out by programs as they execute, the temporal logic upon which Concurrent METATEM is based has a computational interpretation; the actual execution algorithm is described in [2].

Note that executing Concurrent METATEM agent specifications is possible primarily because the models upon which the Concurrent METATEM temporal logic is based are comparatively simple, with an obvious and intuitive computational interpretation. However, agent specification languages in general (e.g., the BDI formalisms of Rao and Georgeff [43]) are based on considerably more complex logics. In particular, they are usually based on a semantic framework known as *possible worlds* [6]. The technical details are somewhat involved for the purposes of this article: the main point is that, *in general*, possible worlds semantics do not have a computational interpretation in the way that Concurrent METATEM semantics do. Hence it is not clear what "executing" a logic based on such semantics might mean. In response to this, a number of researchers have attempted to develop executable agent specification languages with a simplified semantic basis, that has a computational interpretation. An example is Rao's AgentSpeak(L) language, which although essentially a BDI system, has a simple computational semantics [42].

5.2. Compiling Agent Specifications

An alternative to direct execution is *compilation*. In this scheme, we take our abstract specification, and transform it into a concrete computational model via some automatic synthesis process. The main perceived advantages of compilation over direct execution are in run-time efficiency. Direct execution of an agent specification, as in Concurrent METATEM, above, typically involves manipulating a symbolic

representation of the specification at run time. This manipulation generally corresponds to reasoning of some form, which is computationally costly. Compilation approaches aim to reduce abstract symbolic specifications to a much simpler computational model, which requires no symbolic representation. The 'reasoning' work is thus done off-line, at compile-time; execution of the compiled system can then be done with little or no run-time symbolic reasoning.

Compilation approaches usually depend upon the close relationship between models for temporal/modal logic (which are typically labeled graphs of some kind), and automata-like finite state machines. For example, Pnueli and Rosner [41] synthesise reactive systems from branching temporal logic specifications. Similar techniques have also been used to develop concurrent system skeletons from temporal logic specifications. Perhaps the best-known example of this approach to agent development is the *situated automata* paradigm of Rosenschein and Kaelbling [46]. They use an epistemic logic (i.e., a logic of *knowledge* [13]) to specify the perception component of intelligent agent systems. They then used an technique based on constructive proof to directly synthesise automata from these specifications [45].

The general approach of automatic synthesis, although theoretically appealing, is limited in a number of important respects. First, as the agent specification language becomes more expressive, then even offline reasoning becomes too expensive to carry out. Second, the systems generated in this way are not capable of *learning*, (i.e., they are not capable of adapting their "program" at run-time). Finally, as with direct execution approaches, agent specification frameworks tend to have no concrete computational interpretation, making such a synthesis impossible.

6. Verification

Once we have developed a concrete system, we need to show that this system is correct with respect to our original specification. This process is known as *verification*, and it is particularly important if we have introduced any informality into the development process. We can divide approaches to the verification of systems into two broad classes: (1) *axiomatic*; and (2) *semantic* (model checking). In the subsections that follow, we shall look at the way in which these two approaches have evidenced themselves in agent-based systems.

6.1. Axiomatic Approaches

Axiomatic approaches to program verification were the first to enter the mainstream of computer science, with the work of Hoare in the late 1960s [20]. Axiomatic verification requires that we can take our concrete program, and from this program systematically derive a logical theory that represents the behaviour of the program. Call this the program theory. If the program theory is expressed in the same logical language as the original specification, then verification reduces to a proof problem: show that the specification is a theorem of (equivalently, is a logical consequence of) the program theory. The development of a program theory is made feasible by *axiomatizing* the programming language in which the system is implemented. For example, Hoare logic gives us more or less an axiom for every statement type in a simple PASCAL-like language. Once given the axiomatization, the program theory can be derived from the program text in a systematic way.

Perhaps the most relevant work from mainstream computer science is the specification and verification of reactive systems using temporal logic, in the way pioneered by Pnueli, Manna, and colleagues [35]. The idea is that the computations of reactive systems are infinite sequences, which correspond to models for linear temporal logic. Temporal logic can be used both to develop a system specification, and to axiomatize a programming language. This axiomatization can then be used to systematically derive the theory of a program from the program text. Both the specification and the program theory will then be encoded in temporal logic, and

verification hence becomes a proof problem in temporal logic.

Comparatively little work has been carried out within the agent-based systems community on axiomatizing multi-agent environments. We shall review just one approach. In [54], an axiomatic approach to the verification of multi-agent systems was proposed. Essentially, the idea was to use a temporal belief logic to axiomatize the properties of two multi-agent programming languages. Given such an axiomatization, a program theory representing the properties of the system could be systematically derived in the way indicated above. A temporal belief logic was used for two reasons. First, a temporal component was required because, as we observed above, we need to capture the ongoing behaviour of a multi-agent system. A belief component was used because the agents we wish to verify are each symbolic AI systems in their own right. That is, each agent is a symbolic reasoning system, which includes a representation of its environment and desired behaviour. A belief component in the logic allows us to capture the symbolic representations present within each agent. The two multi-agent programming languages that were axiomatized in the temporal belief logic were Shoham's AGENT0 [47], and Fisher's Concurrent METATEM (see above). Note that this approach relies on the operation of agents being sufficiently simple that their properties can be axiomatized in the logic. It works for Shoham's AGENT0 and Fisher's Concurrent METATEM largely because these languages have a simple semantics, closely related to rule-based systems, which in turn have a simple logical semantics. For more complex agents, an axiomatization is not so straightforward. Also, capturing the semantics of concurrent execution of agents is not easy (it is, of course, an area of ongoing research in computer science generally).

6.2. Semantic Approaches: Model Checking

Ultimately, axiomatic verification reduces to a proof problem. Axiomatic approaches to verification are thus inherently limited by the difficulty of this proof problem. Proofs are hard enough, even in classical logic; the addition of temporal and modal connectives to a logic makes the problem considerably harder. For this reason, more efficient approaches to verification have been sought. One particularly successful approach is that of *model checking* [7]. As the name suggests, whereas axiomatic approaches generally rely on syntactic proof, model checking approaches are based on the semantics of the specification language.

The model checking problem, in abstract, is quite simple: given a formula ϕ of language L, and a model M for L, determine whether or not ϕ is valid in M, i.e., whether or not $M \models_L \phi$. Model checking-based verification has been studied in connection with temporal logic. The technique once again relies upon the close relationship between models for temporal logic and finite-state machines. Suppose that ϕ is the specification for some system, and π is a program that claims to implement ϕ. Then, to determine whether or not π truly implements ϕ, we take π, and from it generate a model M_π that corresponds to π, in the sense that M_π encodes all the possible computations of π; determine whether or not $M_\pi \models \phi$, i.e., whether the specification formula ϕ is valid in M_π; the program π satisfies the specification ϕ just in case the answer is 'yes'. The main advantage of model checking over axiomatic verification is in complexity: model checking using the branching time temporal logic CTL ([7]) can be done in polynomial time, whereas the proof problem for most modal logics is quite complex.

In [44], Rao and Georgeff present an algorithm for model checking agent systems. More precisely, they give an algorithm for taking a logical model for their (propositional) BDI agent specification language, and a formula of the language, and determining whether the formula is valid in the model. The technique is closely based on model checking algorithms for normal modal logics [19]. They show that despite the inclusion of three extra modalities, (for beliefs, desires, and intentions),

into the CTL branching time framework, the algorithm is still quite efficient, running in polynomial time. So the second step of the two-stage model checking process described above can still be done efficiently. However, it is not clear how the first step might be realised for BDI logics. Where does the logical model characterizing an agent actually comes from — can it be derived from an arbitrary program π, as in mainstream computer science? To do this, we would need to take a program implemented in, say, PASCAL, and from it derive the belief, desire, and intention accessibility relations that are used to give a semantics to the BDI component of the logic. Because, as we noted earlier, there is no clear relationship between the BDI logic and the concrete computational models used to implement agents, it is not clear how such a model could be derived.

7. Conclusions

In this article, we have given a summary of why agents should be perceived to be a significant technology for software engineering, and also of the main techniques for the specification, implementation, and verification of agent systems. Software engineering for agent systems is at an early stage of development, and yet the widespread acceptance of the concept of an agent implies that agents have a significant future in software engineering. If the technology is to be a success, then its software engineering aspects will need to be taken seriously. Probably the most important outstanding issues for agent-based software engineering are: (i) an understanding of the situations in which agent solutions are appropriate; and (ii) principled but *informal* development techniques for agent systems. While some attention has been given to the latter (in the form of analysis and design methodologies for agent systems [30, 60, 39, 10, 3, 29, 53]), almost no attention has been given to the former (but see [59]).

8. How to Find Out More About Agents

There are now many introductions to intelligent agents and multiagent systems. Ferber [14] is an undergraduate textbook, although as its name suggests, this volume focussed on multiagent aspects rather than on the theory and practice of individual agents. A first-rate collection of articles introducing agent and multiagent systems is Weiß [51]. Two collections of research articles provide a comprehensive introduction to the field of autonomous rational agents and multiagent systems: Bond and Gasser's 1988 collection, *Readings in Distributed Artificial Intelligence*, introduces almost all the basic problems in the multiagent systems field, and although some of the papers it contains are now rather dated, it remains essential reading [4]; Huhns and Singh's more recent collection sets itself the ambitious goal of providing a survey of the whole of the agent field, and succeeds in this respect very well [22]. For a general introduction to the theory and practice of intelligent agents, see Wooldridge and Jennings [58], which focuses primarily on the theory of agents, but also contains an extensive review of agent architectures and programming languages. For a collection of articles on the applications of agent technology, see [28]. A comprehensive roadmap of agent technology was published as [27].

9. References

1. G. Agha, P. Wegner, and A. Yonezawa, editors. *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press: Cambridge, MA, 1993.
2. H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. METATEM: A framework for programming in temporal logic. In *REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness (LNCS Volume 430)*, pages 94–129. Springer-Verlag: Berlin, Germany, June 1989.

3. Bernhard Bauer, Jörg P. Müller, and James Odell. Agent UML: A formalism for specifying multiagent software systems. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering — Proceedings of the First International Workshop (AOSE-2000)*. Springer-Verlag: Berlin, Germany, 2000.
4. A. H. Bond and L. Gasser, editors. *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann Publishers: San Mateo, CA, 1988.
5. C. Brown, L. Gasser, D. E. O'Leary, and Alan Sangster. AI on the WWW: Supply and demand agents. *IEEE Expert*, 10(4):44–49, August 1995.
6. B. Chellas. *Modal Logic: An Introduction*. Cambridge University Press: Cambridge, England, 1980.
7. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press: Cambridge, MA, 2000.
8. P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.
9. D. C. Dennett. *The Intentional Stance*. The MIT Press: Cambridge, MA, 1987.
10. Ralph Depke, Reiko Heckel, and Jochen Malte Kuester. Requirement specification and design of agent-based systems with graph transformation, roles, and uml. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering — Proceedings of the First International Workshop (AOSE-2000)*. Springer-Verlag: Berlin, Germany, 2000.
11. E. H. Durfee. *Coordination of Distributed Problem Solvers*. Kluwer Academic Publishers: Boston, MA, 1988.
12. O. Etzioni and D. S. Weld. Intelligent agents on the internet: Fact, fiction, and forecast. *IEEE Expert*, 10(4):44–49, August 1995.
13. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. The MIT Press: Cambridge, MA, 1995.
14. J. Ferber. *Multi-Agent Systems*. Addison-Wesley: Reading, MA, 1999.
15. N. V. Findler and R. Lo. An examination of Distributed Planning in the world of air traffic control. *Journal of Parallel and Distributed Computing*, 3, 1986.
16. M. Fisher. A survey of Concurrent METATEM — the language and its applications. In D. M. Gabbay and H. J. Ohlbach, editors, *Temporal Logic — Proceedings of the First International Conference (LNAI Volume 827)*, pages 480–505. Springer-Verlag: Berlin, Germany, July 1994.
17. M. Fisher and M. Wooldridge. On the formal specification and verification of multi-agent systems. *International Journal of Cooperative Information Systems*, 6(1):37–65, 1997.
18. M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, Seattle, WA, 1987.
19. J. Y. Halpern and M. Y. Vardi. Model checking versus theorem proving: A manifesto. In V. Lifschitz, editor, *AI and Mathematical Theory of Computation — Papers in Honor of John McCarthy*, pages 151–176. The Academic Press: London, England, 1991.
20. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.
21. C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1978.
22. M. Huhns and M. P. Singh, editors. *Readings in Agents*. Morgan Kaufmann Publishers: San Mateo, CA, 1998.
23. N. R. Jennings. On agent-base software engineering. *Artificial Intelligence*, 117:277–

296, 2000.
24. N. R. Jennings, J. M. Corera, and I. Laresgoiti. Developing industrial multi-agent systems. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 423–430, San Francisco, CA, June 1995.
25. N. R. Jennings, T. J. Norman, P. Faratin, P. O'Brien, and B. Odgers. Autonomous agents for business process management. *Applied Artificial Intelligence Journal*, 14(2):145–190, 2000.
26. N. R. Jennings, T. J. Norman, P. Faratin, P. O'Brien, and B. Odgers. Implementing a business process management system using ADEPT: A real-world case study. *Applied Artificial Intelligence Journal*, 14(5):421–490, 2000.
27. N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.
28. N. R. Jennings and M. Wooldridge, editors. *Agent Technology: Foundations, Applications and Markets*. Springer-Verlag: Berlin, Germany, 1998.
29. Elizabeth A. Kendall. Agent software engineering with role modelling. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering — Proceedings of the First International Workshop (AOSE-2000)*. Springer-Verlag: Berlin, Germany, 2000.
30. D. Kinny and M. Georgeff. Modelling and design of multi-agent systems. In J. P. Müller, M. Wooldridge, and N. R. Jennings, editors, *Intelligent Agents III (LNAI Volume 1193)*, pages 1–20. Springer-Verlag: Berlin, Germany, 1997.
31. M. Klusch, editor. *Intelligent Information Agents*. Springer-Verlag: Berlin, Germany, 1999.
32. V. R. Lesser and L. D. Erman. Distributed interpretation: A model and experiment. *IEEE Transactions on Computers*, C-29(12):1144–1163, 1980.
33. M. Ljunberg and A. Lucas. The OASIS air traffic management system. In *Proceedings of the Second Pacific Rim International Conference on AI (PRICAI-92)*, Seoul, Korea, 1992.
34. P. Maes. Agents that reduce work and information overload. *Communications of the ACM*, 37(7):31–40, July 1994.
35. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems — Safety*. Springer-Verlag: Berlin, Germany, 1995.
36. K. Mori, H. Torikoshi, K. Nakai, and T. Masuda. Computer control system for iron and steel plants. *Hitachi Review*, 37(4):251–258, 1988.
37. N. Negroponte. *Being Digital*. Hodder and Stoughton, 1995.
38. S. Oaks and H. Wong. *Jini in a Nutshell*. O'Reilly & Associates, Inc., 2000.
39. James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Representing agent interaction protocols in UML. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering — Proceedings of the First International Workshop (AOSE-2000)*. Springer-Verlag: Berlin, Germany, 2000.
40. The Object Management Group (OMG). See `http://www.omg.org/`.
41. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the Sixteenth ACM Symposium on the Principles of Programming Languages (POPL)*, pages 179–190, January 1989.
42. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, (LNAI Volume 1038)*, pages 42–55. Springer-Verlag: Berlin, Germany, 1996.
43. A. S. Rao and M. Georgeff. BDI Agents: from theory to practice. In *Proceedings of*

the First International Conference on Multi-Agent Systems (ICMAS-95), pages 312–319, San Francisco, CA, June 1995.
44. A. S. Rao and M. P. Georgeff. A model-theoretic approach to the verification of situated reasoning systems. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 318–324, Chambéry, France, 1993.
45. S. Rosenschein and L. P. Kaelbling. The synthesis of digital machines with provable epistemic properties. In J. Y. Halpern, editor, *Proceedings of the 1986 Conference on Theoretical Aspects of Reasoning About Knowledge*, pages 83–98. Morgan Kaufmann Publishers: San Mateo, CA, 1986.
46. S. J. Rosenschein and L. P. Kaelbling. A situated view of representation and control. In P. E. Agre and S. J. Rosenschein, editors, *Computational Theories of Interaction and Agency*, pages 515–540. The MIT Press: Cambridge, MA, 1996.
47. Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.
48. R. G. Smith. *A Framework for Distributed Problem Solving*. UMI Research Press, 1980.
49. R. Steeb, S. Cammarata, F. A. Hayes-Roth, P. W. Thorndyke, and R. B. Wesson. Distributed intelligence for air fleet control. In A. H. Bond and L. Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 90–101. Morgan Kaufmann Publishers: San Mateo, CA, 1988.
50. H. Van Dyke Parunak. Manufacturing experience with the contract net. In M. Huhns, editor, *Distributed Artificial Intelligence*, pages 285–310. Pitman Publishing: London and Morgan Kaufmann: San Mateo, CA, 1987.
51. G. Weiß, editor. *Multi-Agent Systems*. The MIT Press: Cambridge, MA, 1999.
52. J. E. White. Telescript technology: The foundation for the electronic marketplace. White paper, General Magic, Inc., 2465 Latham Street, Mountain View, CA 94040, 1994.
53. Mark Wood and Scott A. DeLoach. An overview of the multiagent systems engineering methodology. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering — Proceedings of the First International Workshop (AOSE-2000)*. Springer-Verlag: Berlin, Germany, 2000.
54. M. Wooldridge. *The Logical Modelling of Computational Multi-Agent Systems*. PhD thesis, Department of Computation, UMIST, Manchester, UK, October 1992.
55. M. Wooldridge. Agent-based software engineering. *IEE Proceedings on Software Engineering*, 144(1):26–37, February 1997.
56. M. Wooldridge. *Reasoning about Rational Agents*. The MIT Press: Cambridge, MA, 2000.
57. M. Wooldridge, S. Bussmann, and M. Klosterberg. Production sequencing as negotiation. In *Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM-96)*, pages 709–726, London, UK, April 1996.
58. M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.
59. M. Wooldridge and N. R. Jennings. Pitfalls of agent-oriented development. In *Proceedings of the Second International Conference on Autonomous Agents (Agents 98)*, pages 385–391, Minneapolis/St Paul, MN, May 1998.
60. M. Wooldridge, N. R. Jennings, and D. Kinny. A methodology for agent-oriented analysis and design. In *Proceedings of the Third International Conference on Autonomous Agents (Agents 99)*, pages 69–76, Seattle, WA, May 1999.