FEATURE

SOFTWARE ENGINEERING WITH AGENTS: Pitfalls and Pratfalls

While agent-based systems are becoming increasingly well understood, multiagent systems development is not. This article[1] identifies key pitfalls that await the agent system developer and, where possible, makes recommendations to avoid or rectify these pitfalls.

MICHAEL J. WOOLDRIDGE AND NICHOLAS R. JENNINGS
Queen Mary and Westfield College

Industrial-strength software systems are inherently difficult to engineer correctly and efficiently. Since the software crisis was recognized in the 1960s and 1970s, significant research and development effort has been directed at making it easier and cheaper to engineer increasingly complex software systems. Despite significant progress, software engineering's fundamental problems, articulated most famously by Fred Brooks in his 1976 book *The Mythical Man-Month*, remain as true today as then.

Elsewhere we have argued that the notion of an *agent* as a self-contained problem-solving system capable of autonomous, reactive, proactive, social behavior represents yet another tool for the software engineer.[1,2] We believe this tool can lead to improvements in engineering certain types of complex distributed systems. However, comparatively little effort has been devoted to understanding the pragmatics of engineering such systems. If agent technology is to achieve its potential, then these pragmatic aspects must be studied and understood. Ignoring them will result in a backlash against agents similar to that experienced against expert systems, logic programming, and all the other good ideas that have promised to fundamentally change computing.

Our aims in this article are to restate the main arguments favoring the view that intelligent agents and multiagent systems can potentially play a significant role in complex, distributed-systems engineering; and to emphasize the main pitfalls awaiting the agent system developer, in hopes that these pitfalls can subsequently be avoided.

We've modeled the article on Bruce Webster's *Pitfalls of Object-Oriented Development*.[3] Following an overview of the main arguments in favor

[1] This article is an updated and revised version of "Pitfalls of Agent-Oriented Development," *Proc. Second Int'l Conf. Autonomous Agents* (Agents 98), ACM Press, New York, 1998.

1089-7801/99/$10.00 ©1999 IEEE

of agents as a software engineering paradigm, we identify six categories of problems. We focus on pitfalls common to agent-based development projects and ignore generic software development issues.[4] Finally, we present a case study of a significant agent-based development project, in which we encountered—and overcame—pitfalls.

AGENTS AND SOFTWARE ENGINEERING

Arguably, software engineering's most significant improvements have resulted from the introduction of powerful abstractions with which to manage software's inherent complexity. The key advances in program design and development over the past three decades—procedural abstraction, abstract data types, and most recently, object-oriented programming—all represent increasingly powerful examples of such abstractions.

Probably the single most compelling argument in favor of agents for software engineering is that they represent yet another such abstraction. Just as many systems may naturally be understood and modeled as a collection of interacting but passive objects, so many other systems may be naturally understood and modeled as a collection of interacting autonomous agents.[1,5] We expect an agent-oriented view of software to complement—not replace—the object-oriented view.

Developers will typically implement agents using object-oriented techniques, and there will usually be fewer agents in the system than objects. However, agents and objects do compete in the sense that agent technology is more appropriate than object technology for applications that can be naturally modeled as societies of interacting autonomous entities.[5]

Other rationales support agents as a useful paradigm for software engineering. In particular, the technology of autonomous agents and multiagent systems appears appropriate for building systems in which

- data, control, expertise, or resources are distributed;
- agents provide a natural metaphor for delivering system functionality; or
- a number of legacy systems must be made to interwork.

Although agent technology is an appropriate tool for developing certain applications, it nevertheless has its problems. We describe these in the following sections.

POLITICAL PITFALLS

Software engineers are likely to encounter two major pitfalls in a corporate environment.

You oversell agents

Agents are a powerful, natural metaphor for conceptualizing, designing, and implementing many complex, distributed applications. Some tasks, however, are simply beyond the scope of automation. Indeed, many of the systems that have been built using agent technology could likely have been built just as easily with nonagent techniques.

Atomic problem-solving components within agent-based systems must still perform the necessary domain tasks, and their implementation can only use the currently available (limited) techniques. Developers can obtain extra leverage by applying multiple problem-solving methods and by carefully managing the interactions between the components, but ultimately, a developer will need to write these components.

Another aspect of overselling is to erroneously assume that agents have somehow solved the problems that have dogged artificial intelligence since its inception. Agent systems typically use AI techniques—in this sense, they are an application of AI technology—but their "intelligent" capabilities are limited by AI's state of the art.

You get dogmatic about agents

Although they have been used in a wide range of applications,[5] agents are not a universal solution. For many applications, conventional software development paradigms (such as object-oriented programming) are more appropriate. Indeed, given the relative immaturity of agent technology and the small number of deployed agent applications, an agent-based solution should evidence clear advantages before developers contemplate such an approach.

Another form of dogma associated with agents relates to their definition. Most agent developers have their own opinion on exactly what constitutes an agent—and no two developers appear to share exactly the same opinion.[2] Having made a valid case for an agent-based approach, developers then tend to shoehorn their solution to fit with their problem, even when their solution is inappropriate to the problem at hand.

MANAGEMENT PITFALLS

Managers, in addition to software developers, may also fall prey to the following pitfalls.

You don't know why you want agents

Managers reading forecasts such as "agents will generate U.S. $2.6 billion in revenue by the year 2000"[6] will undoubtedly want part of this revenue. Yet frequently, managers proposing an agent project have no clear idea about the benefits of agent systems. Consequently, they often initiate agent projects without clear goals. Without goals, there are also no criteria for assessing success and project performance. Catastrophic project failures can thus occur, seemingly out of the blue.

A related issue concerns a general lack of understanding concerning the "why and how" of using agent technology. Having perhaps first developed an agent technology or specific agents, managers often then search for a suitable application. Invariably, when an application is matched with a technology, the agent's full potential is not achieved—the agents have either the wrong functionality or emphasis.

You want generic solutions

Another common pitfall is devising an architecture or testbed that supposedly enables a whole range of potential systems to be built when what is actually required is a bespoke design to tackle a single application. As those with object-oriented development experience know, reuse is difficult to attain unless development addresses a close-knit range of problems with similar characteristics.[4] Additionally, general solutions are more difficult and costly to develop, and often require extensive tailoring to work in different applications. Generally, architectures for agents and multiagent systems are suitable only with certain types of applications.

CONCEPTUAL PITFALLS

Developers, having a suitable application that would benefit from agent technology, may experience misconceptions about what they can achieve with agents.

You believe in silver bullets

The holy grail of software engineering is a "silver bullet": a technique that will provide an order-of-magnitude improvement in software development.[7] Many technologies have been promoted as the silver bullet—automatic programming, expert systems, graphical programming, and formal methods, for example.

There are clearly good arguments—largely untested—for agent technology's leading to improved development of complex distributed software systems. Certainly no scientific evidence supports the claim that agents offer any advance in software development—the evidence to date is purely anecdotal. With time, demonstrable benefits for software development through agent technology will be proven, but if agents do lead to a genuine improvement in software development practice, the advance is unlikely to represent an order-of-magnitude improvement.

You forget agents are *software*

The development of any agent system, however trivial, is essentially a process of experimentation. Unfortunately, the experimental process encourages developers to forget that they are actually developing software. Mundane software engineering processes—requirements analysis, specification, design, verification, and testing—are forgotten and the project flounders. Although development techniques for agent systems are in their infancy, almost any principled software development technique is better than none. Thus, in the absence of agent-oriented development techniques, developers can adapt object-oriented techniques to great effect.

You forget agents are *multithreaded* software

Long recognized as exceedingly complex to design and implement, multithreaded systems have been extensively researched. Researchers have tried to understand this complexity and to develop formalisms and tools for managing it.[8] Despite this effort, the problems inherent in developing multithreaded systems cannot be considered solved.

By their very nature, multiagent systems tend to be multithreaded, both within an agent and certainly within the society of agents. (A society of agents is one that works together to achieve a common task.) Multiagent systems tend to lack any central control, which makes conflict between system components a possibility. The multiagent system developer must therefore recognize and plan for problems such as synchronization, mutual exclusion for shared resources, deadlock, and livelock.[8]

ANALYSIS AND DESIGN PITFALLS

Once past the initial process of deciding to use agent technology with a suitable application, developers can still be tripped up in the design phase.

You ignore related technology

As Oren Etzioni noted, "intelligent agents are 99% computer science and 1% AI".[9] Given this, develop-

ers should exploit conventional software technologies and techniques wherever possible to engineer the conventional 99 percent. Such exploitation speeds up development, avoids reinventing the wheel, and enables sufficient time to be devoted to the value-added agent component. Many agent projects could benefit from exploiting available technology such as distributed computing platforms (for example, the Object Management Group's CORBA) to handle low-level interoperation of heterogeneous distributed components; database systems to handle large information-processing requirements; and expert systems to handle reasoning and problem-solving tasks.

An issue related to this pitfall is the failure of many projects to exploit both official and de facto standards. In a field as new as agent systems, few established standards exist. Thus agents developed by different organizations are unable to interoperate. However, despite the lack of internationally accepted standards, de facto standards can be employed in many cases. Java, clearly becoming a de facto standard for agent systems development, lets developers leverage many freely available software tools and components (such as the Remote Method Invocation application program interface).

Another de facto standard is KQML,[10] an agent communication language that has been used in many agent development projects. Also, the Foundation for Intelligent Physical Agents is developing a second-generation agent communication language and an agent infrastructure.[11]

Your design doesn't exploit concurrency

Minimal, or in extreme cases nonexistent, concurrent problem solving is one of the most obvious features of a poor multiagent design. Typically, in poorly designed systems one agent does some processing, produces some results, and then enters into an idle state. The results are passed to another (previously inactive) agent, which processes them, produces more results, returns to inactivity, and so on. Such a multiagent design is poor because there is only ever a single thread of control. An agent-based solution may be inappropriate for systems requiring only a single thread of control—a traditional object-oriented solution will likely be adequate.

The analysis and design phases, therefore, should result in developers' producing a system that exploits concurrent problem-solving activity. Concurrency lets the system simultaneously handle multiple objectives and perspectives, responding and reacting to the environment at many different levels. Concurrency also enables multiple, complementary problem-solving methods to interwork.

You ignore legacy

When using a new technology to build systems, developers might assume that it is possible—or necessary—to start from a blank slate and design every component from scratch. A software system's most important components, however, will often be legacy; that is, functionally essential but technologically obsolete software components that cannot readily be rebuilt.

Legacy components can be incorporated into an agent system by wrapping them with an agent layer.[12] By providing them with a software layer that realizes an agent-level API, legacy components can communicate and cooperate with agents. Such a solution extends the functionality of legacy software.

AGENT-LEVEL PITFALLS

At the micro, or agent, level of software development, designers can encounter three problems.

You want your own agent architecture

When first attempting an agent project, developers may think that no existing agent architecture meets the problem requirements and that it is therefore necessary to design one. Designing an agent architecture from scratch in order to build agents is often a mistake for two key reasons. First, developing a new, reliable architecture with sufficient power takes significant research and development. Such effort could otherwise be devoted to gaining experience with, and ultimately proving, the technology. Second, unless you were to carry out the design process in tandem with a major research effort, it is unlikely that the resulting architecture would be sufficiently novel to generate either interest or revenue.

Developers should therefore study the various agent architectures described in the literature[2] and either license one or implement an off-the-shelf design.

Your agents use too much AI

It is tempting to focus exclusively on the application's agent-specific, "intelligence" aspects in building an agent application. These aspects, after all, often serve as the project's justification. Too much focus on AI can result in an agent framework overburdened with experimental techniques—such as

natural language interfaces, planners, theorem provers, and reason maintenance systems—and thus be unusable.

A more successful, if short-term, strategy is to build agents with a minimum of AI techniques. As such systems succeed in a "useful first" strategy, they can be progressively evolved into richer systems.[9]

Your agents use no AI

At one extreme, developers are preoccupied with building agent systems employing only the most sophisticated and complex AI techniques available and consequently failing to provide a sufficiently robust basis for the system. At the other extreme, developers build so-called agents that do nothing to justify the use of the term. It's increasingly common, for instance, to find straightforward distributed systems referred to as multiagent systems.

A very different but equally common example is the practice of referring to Web pages having any behind-the-scenes processing as "agents". Such practices are unhelpful for several reasons. First, they will cause the term "agent" to lose its meaning. Second, such practices raise the expectations of software recipients, who will be disappointed when they ultimately receive a very conventional piece of software. Finally, these practices lead to cynicism on the part of software developers (who come to believe that "agent" is simply another meaningless management buzzword).

SOCIETY-LEVEL PITFALLS

Finally, at the macro, or society, level, five potential pitfalls await the developer.

You see agents everywhere

After first learning about multiagent systems, there is a tendency to view everything as an agent, which is perceived to be somehow conceptually clean. After all, an object-oriented language is considered "pure" if everything in the language is an object—isn't the situation identical for multiagent systems? If you adopt this viewpoint, you end up with agents for everything, even addition and subtraction. In the enormously influential ACTOR paradigm of concurrent computation,[13] this is essentially what happens.

The overheads of managing agents and interagent communication will rapidly outweigh any benefits of an agent-based solution. Generally, agents should be coarse grained, in that each should embody significant, coherent computational functionality. As a rule of thumb, a multiagent system is often regarded as large if it contains more than 10 agents.

You have too few agents

While some designers imagine a separate agent for every possible task, others appear not to recognize the value of a multiagent approach. They create a system that completely fails to exploit the agent paradigm's power, and develop a solution with a very small number of agents doing all the work. Such solutions tend to fail the standard software engineering test of coherence, which requires that a software module should have a single, coherent function. The result is as if a developer were to write an object-oriented program by bundling all the functionality into a single class.

You obsess on infrastructure

A major obstacle to the wider use of agent technology is that there are no widely used software platforms for developing multiagent systems. Such platforms would provide all the basic infrastructure (for message handling, tracing and monitoring, runtime management, and so on) required to create a multiagent system. As a result, almost every multiagent system project that we have encountered has devoted significant resources to implementing this infrastructure from scratch. During this implementation stage, valuable time (and hence money) is often spent implementing libraries and software tools that, in the end, do little more than exchange KQML-like messages[10] across a network.

Your agents interact too freely

Numerous systems interacting with one another using simple rules can generate behavior that appears considerably more complex than the sum of the components would indicate.[14] Therein lies one of the great strengths—and weaknesses—of the multiagent systems approach. The strength is that developers can exploit this emergent functionality to provide simple, robust cooperative behavior. The weakness is that emergent functionality is akin to chaos.

One technique for managing multiagent dynamics is to restrict the way that agents interact. Thus, very simple cooperation protocols are preferable to richer ones, with "one-shot" protocols (such as requesting and replying) being both adequate and desirable for many applications.

Your system lacks structure

A common misconception is that agent-based systems require no real structuring. Although perhaps

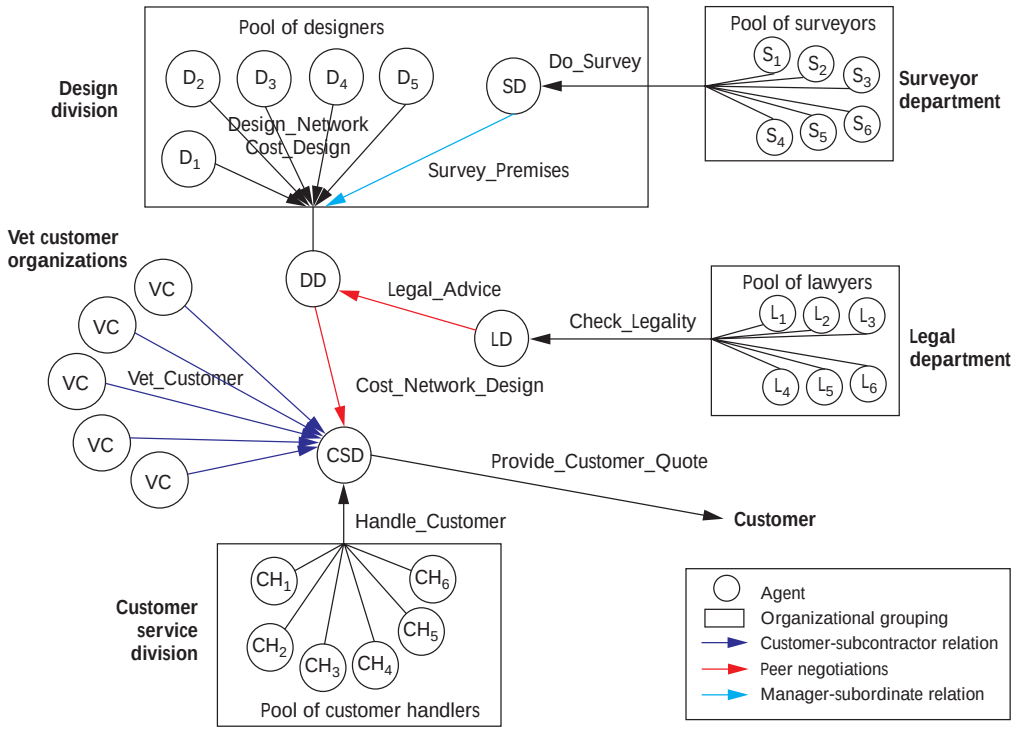

Figure 1. The Advanced Design Environment for Process Tasks (ADEPT) manages a British Telecom business process that provides quotes for installing a network to deliver telecommunications services.

true in certain cases, it should not be viewed as the only way of developing agent societies. Many agent systems require considerably more system-level engineering, particularly large-scale systems or those in which the society is supposed to act with some commonality of purpose. In such cases, a means of structuring the society is needed to reduce the system's complexity, increase the system's efficiency, and more accurately model the problem being tackled. The precise nature of this structuring clearly depends on the problem at hand. Common options include close-knit teams of agents cooperatively achieving a common goal; abstraction hierarchies modeling the problem from different perspectives; and intermediaries acting as a single point of contact for a number of agents.[15]

REAL-WORLD CASE STUDY

In this case study, we show how potential pitfalls were managed or avoided in a large-scale industrial project with which we were involved. The system was developed in the Advanced Design Environment for Process Tasks (ADEPT) project.[16] ADEPT is a multiagent application shell that can be instantiated for various business application domains. When instantiated, the system manages the business process. The specific domain considered here is managing a British Telecom business process that provides quotes for installing a network to deliver telecommunications service. The process, as Figure 1 shows, involves BT's customer service, design, surveyor, and legal departments, and the various organizations that provide the outsourced service of vetting customers (VCs).

To initiate the process, a customer contacts Customer Service with a set of requirements. In parallel to capturing the requirements, Customer Service arranges to have the customer vetted. If the customer fails the vetting procedure, the quote process terminates. Assuming the customer is satisfactory, their requirements are mapped against the service portfolio. When requirements can be met by an off-the-shelf item, an immediate quote is offered.

In the case of bespoke services, however, the process is more complex. Customer Service further analyzes the customer's requirements, and Legal checks the legality of the proposed service. If the desired service is illegal, the quote process terminates. If the requested service is legal, the design phase can start.

To prepare a network design, BT must usually dispatch a surveyor to the customer's premises so

that a detailed plan of the existing equipment can be produced. When the network design and costing are complete, Design sends the quote to Customer Service, which, in turn, forwards it to the customer. This action completes the business process.

From this high-level system description, we identified a number of autonomous problem-solving entities. Each department became an agent, and each individual within a department became an agent. To achieve their individual objectives, agents interacted with one another. All interactions took the form of negotiations about which services the agents would provide to one another and under what terms and conditions. Successful negotiations resulted in mutually agreeable contracts.

Project Pragmatics

Centralized workflow systems are simply too unresponsive and unable to cope with unpredictable events. British Telecom therefore decided to devolve responsibility for managing the business process to software entities that could respond more rapidly to changing circumstances. Since the various devolved functions inevitably have interdependencies, these software entities must interact to resolve conflicts. Such an approach leaves autonomous agents as the most natural means of modeling the solution.

The ADEPT project was run according to a strong set of software management procedures, with a particular focus on the analysis, specification, and design phases. Developers borrowed a number of object-oriented techniques (in particular, use cases and interaction diagrams). However, the project would have benefitted from stronger testing and evaluation procedures, especially when it came to demonstrating the value-added nature of the agent-oriented approach.

From the beginning, ADEPT was conceived as a full-fledged distributed application. This meant the project had to expend considerable effort tackling the problems of debugging and visualizing multithreaded, distributed software. Indeed, a significant proportion of the research and development effort was on tools to track events in the system. Rather than reimplementing communications from first principles, we built ADEPT on top of a commercial CORBA platform. This platform provided the basis of handling distribution and heterogeneity in the ADEPT system. ADEPT agents also required the ability to undertake context-dependent reasoning, and so we incorporated a widely used expert system shell into the agent architecture.

On the negative side, ADEPT failed to exploit any available standards for agent communication languages. This shortcoming restricted the interoperation of the ADEPT system. ADEPT did, however, exploit an off-the-shelf communications framework, and it also used an architecture that had been developed in two previous projects, Architecture for Cooperating Heterogeneous Online Systems (ARCHON) and Generic Rules and Agent Model Testbed Environment (GRATE).[12,17] This shortened the analysis and design phases.

The business process domain has a large number of legacy components, especially databases and scheduling software. In this case, we wrapped these as resources or tasks within particular agents.

Agent Interactions

ADEPT agents embodied comparatively small amounts of AI technology compared to most agent systems. For example, planning was handled by storing partial plans in a plan library (in the style of the Procedural Reasoning System).[18] AI techniques were used specifically for agent negotiation and the way that agents responded to their environment. In the former case, each agent had a rich set of rules governing which negotiation strategy to adopt in which circumstances, how it should respond to incoming negotiation proposals, and when it should change its negotiation strategy. In the latter case, agents were required to respond to unanticipated events in a dynamic and uncertain environment.

ADEPT agents were relatively coarse grained in nature. They represented organizations, departments, or individuals. Each such agent controlled a number of resources and was capable of a range of problem-solving behaviors. This led to a system design in which there were typically fewer than 10 agents at each level of abstraction and in which the primitive agents were still capable of fulfilling some high-level goals.

The complexity of agent interactions in ADEPT was ameliorated through its notion of agency: a collection of agents working together to achieve an objective. For example, Design was an agency composed of individual designers. From the outside, agents interacted with a representative of the agency and not with the individual members. This helped reduce the number of interactions in the system and so limited the scope for unexpected emergent behavior.

The other technique for controlling interactions was to allow service contracts to cover multiple invocations. Thus, the Customer Service agent negotiated with the VC agents for a number of vettings (say, 50) in a given contract. This meant that

a new negotiation was not required for each and every quote to be generated. There was still a reasonable degree of uncertainty about how negotiations would progress, primarily because the agents were free to adopt whichever negotiation strategy best fitted their current situation. We explored this facet of the system through systematic empirical evaluation of a number of negotiation use cases.

CONCLUSION

In the future, we intend to investigate development methodologies for agent-based systems. Such methodologies will offer developers a systematic framework to address the pragmatic concerns of software engineers charged with the development of agent-based systems. We also intend to develop models that will enable us to predict and verify the behavior of complex multiagent systems. ■

ACKNOWLEDGMENTS

We thank Simon Lewis for informing us about Bruce Webster's book, *Pitfalls of Object-Oriented Development* (M&T Books, 1995), which provided a model framework for this article. Thanks also to Van Parunak for detailed, helpful comments on an earlier article draft.

REFERENCES

1. M. Wooldridge and N.R. Jennings, "Intelligent Agents: Theory and Practice," *The Knowledge Eng. Rev.*, Vol. 10, No. 2, 1995, pp. 115–152.
2. M. Wooldridge and N.R. Jennings, "Pitfalls of Agent-Oriented Development," *Proc. Second Int'l Conf. Autonomous Agents*, ACM Press, New York, May 1998, pp. 385–391.
3. B.F. Webster, *Pitfalls of Object-Oriented Development*, M&T Books, New York, 1995.
4. M. Wooldridge, "Agent-based Software Engineering, *IEE Proc. Software Eng.*, Vol. 144, No. 1, IEE Press, London, 1997, pp. 26–37.
5. N.R. Jennings and M. Wooldridge, "Applications of Intelligent Agents," in *Agent Technology: Foundations, Applications, and Markets*, N.R. Jennings and M. Wooldridge, eds., Springer-Verlag, Berlin, 1998, pp. 3–28.
6. C. Guilfoyle, *Intelligent Agents: The Next Revolution in Software*, Ovum Ltd., London, 1994.
7. F.P. Brooks, "No Silver Bullet," *Proc. IFIP Tenth World Computer Conf.*, H.J. Kugler, ed., Elsevier Science, Amsterdam, Netherlands, pp. 1,069–1,076.
8. M. BenAri, *Principles of Concurrent and Distributed Programming*, Prentice Hall, Englewood Cliffs, N.J., 1990.
9. O. Etzioni, "Moving Up the Information Food Chain: Deploying Softbots on the Worldwide Web," *Proc. 13th Nat'l Conf. Artificial Intelligence* (AAAI 96), AAAI Press, San Mateo, Calif., 1996.
10. J. Mayfield, Y. Labrou, and T. Finin, "Evaluating KQML as an Agent Communication Language," in *Intelligent Agents II* (LNAI Volume 1037), M. Wooldridge, J.P. Müller, and M. Tambe, eds., Springer-Verlag, Berlin, 1996, pp. 347–360.
11. *FIPA 97, Version 2, Part 2: Agent Communication Language*, Foundation for Intelligent Physical Agents, Geneva, 1998.
12. N.R. Jennings et al., "Using ARCHON to Develop Real-World DAI Applications for Electricity Transportation Management and Particle Acceleration Control," *IEEE Expert*, Vol. 11, No. 6, Dec. 1996, pp. 60–88.
13. G. Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, Mass., 1986.
14. L. Steels, "Cooperation between Distributed Agents through Self Organization," *Decentralized AI—Proc. First European Workshop on Modeling Autonomous Agents in a Multi-Agent World*, Y. Demazeau and J.P. Müller, eds., Elsevier Science, Amsterdam, Netherlands, 1990, pp. 175–196.
15. A.H. Bond and L. Gasser, eds., *Readings in Distributed Artificial Intelligence*, Morgan Kaufmann, San Mateo, Calif., 1988.
16. N.R. Jennings et al., "Agent-Based Business Process Management," *Int'l J. Cooperative Information Systems*, Vol. 5, No. 23, 1996, pp. 105–130.
17. N.R. Jennings, "Specification and Implementation of a Belief Desire Joint-Intention Architecture for Collaborative Problem Solving," *J. Intelligent and Cooperative Information Systems*, Vol. 2, No. 3, 1993, pp. 289–318.
18. M.P. Georgeff and A.L. Lansky, "Reactive Reasoning and Planning," *Proc. Sixth Nat'l Conf. Artificial Intelligence* (AAAI 87), AAAI Press, San Mateo, Calif., 1987, pp. 677–682.

Michael J. Wooldridge is a reader in the Department of Electronic Engineering at Queen Mary and Westfield College, University of London. He is an associate editor of the *Journal of Autonomous Agents and Multi-Agent Systems* and is on the editorial board of the *Journal of Applied AI*. He coordinates the European Network of Excellence for agent-based computing (http://www.AgentLink.org/). Wooldridge earned a PhD in computation from the University of Manchester.

Nicholas R. Jennings is a professor of intelligent systems and head of the Intelligent Systems Group in the Department of Electronic Engineering at Queen Mary and Westfield College. He is editor-in-chief of the *Journal of Autonomous Agents and Multi-Agent Systems*. Jennings received the International Joint Conference on Artificial Intelligence (IJCAI) Computers and Thought Award in 1999. He earned a PhD in artificial intelligence from the University of London.

Readers may contact the authors at Dept. of Electronic Engineering, Queen Mary and Westfield College, Univ. of London, London E1 4NS, UK; {M.J.Wooldridge, N.R.Jennings}@elec.qmw.ac.uk.