Responsible Agent Behavior

A Distributed Computing Perspective

EBRAHIM (ABE) MAMDANI AND JEREMY PITT
Imperial College of Science, Technology, and Medicine

Software agents that are autonomous, communicative, and possibly intelligent processes raise new questions for developers of distributed systems. Specifically, what is responsible agent behavior, and who, as the owner, is legally responsible for it? The answers involve an understanding of human-agent interaction, agent-oriented middleware, and social behavior.

Society is fundamentally and unequivocally set up to hold individuals accountable for their actions. When agents act on a user's behalf, however, the legal and social ramifications can be obscure. While researchers in artificial intelligence (AI) focus primarily on the intelligence of agents, we are concerned with the legal aspects of autonomous, asynchronously communicating, and perhaps intelligent software, especially in open systems.

An agent is an identifiable computational entity that automates some aspect of task performance or decision making to benefit a human entity. The human entity delegating its responsibility to the computational one is what we call the owner of the agent. By ownership, we mean that there is a specific person or organization that is responsible for the agent's actions.

Two or more agents acting together form a multiagent system. To establish a practical agent society, the agents must communicate with each other openly and directly, and be prepared to encounter new agents. Therefore, agents are loosely coupled from the control (as well as ownership) perspective, and an owner may not know with whom its agent is communicating. A communicative act is an action that can have legal or social consequences (for example, the creation of a contract or unintended disclosure of confidential information), for which the owner must take responsibility.

Furthermore, the immediate effect of a communicative act cannot be predicted, unless the state and behavior of each agent is specified and fixed in advance. However, communication in an agent society is asynchronous, so many software agents will demonstrate nondeterministic behavior.

Programming experience shows that complex behavior of software requires many internal states. We have to assume that some software agents will have a sufficiently large number of internal states to be capable of seemingly intelligent behavior (for example, Deep Blue 2). Hence, an agent's future external behavior cannot be guaranteed on the basis of its past behavior, even if that behavior has been monitored over time. Complete compliance tests of intelligent agents, therefore, may not be achievable because of the (possibly) large number of internal states. Thus, the best we can say is that an agent has not exhibited noncompliant behavior yet.

There are two conclusions from this argument: First, that communication between agents implies a contract between owners, and, second, that complexity of agents implies possibly unpredictable behavior. Therefore, an appropriate legal framework is required to underwrite the conse-

 1089-7801/00/$10.00 ©2000 IEEE

quences of (communicative) actions and to provide safeguards against unlawful activities. The legal implications of agent technology require new ways of thinking about working with an agent, new requirements for agent-oriented middleware, and additional types of social behavior to be considered when designing a multiagent system.

HUMAN-AGENT CO-WORKING

The main characteristic of working with an agent is not that the software agent displays intelligence but that it encapsulates and demarcates its owner's legal responsibility. The agent's capability to act intelligently only increases that responsibility. Software agents can act as surrogates for their respective human entities, carrying out routine but intelligent tasks without constant monitoring. Given that appliances and services will likely be increasingly and directly accessible on a computer network, software agents will be important intermediaries between humans and the physical world.

Software agents encapsulate and demarcate their owners' legal responsibility.

The delegating human authority for each agent must be explicitly identified. Agents will be bound to one or more physical computational entities (hardware) by middleware, but can only be expected to interact with other software agents asynchronously, and are loosely connected to each other from the standpoints of conventional control and shared knowledge. For this reason, the interaction between agent and owner must be tightly coupled in the sense of an identifiable human taking responsibility for an agent's behavior. This applies to all agents, not just to personal agents. Tight coupling does not imply that an agent's every action needs the owner's sanction, but that the owner assumes responsibility for all possible behavior of an agent while it acts autonomously as its owner's surrogate. This has a range of consequences, including

- The owner will be responsible for damage caused by agents either deliberately or accidentally (for example, because of software bugs).
- The owner will often not be the agent's developer, so the actual owners may seek guarantees from the developer, intermediate suppliers, and perhaps governmental licensing authorities.
- Because an agent's damage may come to light only after a considerable time, agent developers may be required to incorporate accident black boxes or audit trails to log the agent's behavior.

AGENT-ORIENTED MIDDLEWARE

Computationally, a software agent is a process that encapsulates some notion of state, communicates with other processes by message passing, and can perceive (affect) its environment. The agent process will also require a set of ontologies, grounded in the application domain; an AI component, for reasoning about the domain; and a more or less anthropomorphic shell, for user interaction. Collectively, agents may be physically distributed, but individually, they are logically distinct and unlikely to operate in isolation.

To function at all, each agent must be bound to one or more hardware platforms. Middleware is any entity that is interposed between a client and a server, a peer and another peer, or an application and a network. This entity could also provide additional functionality that might be associated with a human middleman, broker, or arbiter, for example.

Middleware Services

Middleware functionality is implicitly or explicitly evident in distributed object-oriented computing, the Web, network computing, and computer-telephony integration (CTI),[1] providing the following services:

- Dynamic binding services (such as CORBA) between agent and hardware entity, such that the binding also provides a handle for any entity "owning" the agent.
- Location services (such as the Web) that enable agents to find and communicate with each other either directly or indirectly, to map task requests to service instances, and to facilitate agent interaction in either client-server or peer-peer configurations.
- Application services, including third-party service provision and deployment, dynamic self-configuration, and push-pull operations between clients and servers (as in network computing).
- Management services such as registration and lifecycle management (defining and maintaining type descriptions and service instances, and ensuring service availability), banking, and billing.

Agent-oriented middleware should provide these services to agent-based applications and systems. However, while there is some overlap between the services, until now middleware for agents has typically provided only a subset of the first two types.[2-4]

Additional Services

In addition to the service requirements described above, a generic middleware for agent-based systems should have services for supporting responsible behavior. Management services are an example of this, but we also perceive the need for federation, ownership, and mobility services.

Federation services. Negotiation, delegation, and cooperation techniques are used to forge temporary alliances among otherwise autonomous operators to form a federation of multiple service providers. Federation is important in light of deregulation, open markets, and personalized service delivery; and brokerage is required by autonomous and heterogeneous organizations to communicate, collaborate, and coordinate. However, a federation must be governed by a contract, and a federation service must also include service-level agreements, contract management, and dispute resolution.

Ownership services. Agent technology offers a profound shift in the concept of ownership as it relates to computation. Rather than owning hardware and purchasing a license to run somebody else's software, in the new model someone owns software and buys a license to run it on somebody else's hardware. Ownership services encompass the functions needed to manage such ownership in an open, distributed computing environment.

Ownership services and related support functions can be compared to automobile ownership. For example,

- A driver (user) owns a vehicle (agent and/or software) but pays road tax to drive it (legally) on the roads (third-party-owned infrastructure).
- A driver registers ownership with a vehicle licensing authority (trusted third party); passes a licensing exam, which qualifies him or her to drive a certain class of vehicles; and buys insurance.
- A driver buys fuel for the vehicle (CPU cycles), which requires a yearly roadworthiness certificate (agent upgrade by self-configuration), and whose steering wheel may be on either side (communication protocol, either HTTP or IIOP).

Mobility services. Agents need mobility to more efficiently use bandwidth, limited resources (for example, battery power), or intermittent connections. Mobility requires binding the software to different processors over time. Efficient middleware would make the process transparent so that agent mobility will not be noticeable.

Middleware will thus provide agents with additional functionality that we would, intuitively, associate with certain actors in human societies. This would obviously be realized in computational terminology of types and services, and so provides agents with the opportunity to be dynamic (that is, mobile, self-configuring, and opportunistic). With such a definition, an open issue is whether middleware is distinct from other agents or is an agent itself with human-delegated responsibility.

This line of thought leads to a key design tradeoff. We trade fat middleware and thin agents (maximizing the abstraction of agents' common functionality, "delegating" the functionality to a separate middleware component) for thin middleware and fat agents (maximizing each agent's autonomy and minimizing the common functionality delegated to the middleware).

As agents become more sophisticated, middleware will manage some services, and agents will manage others. As the intelligence of an overall system increases, the distribution of intelligence will inevitably tend toward the agents and away from the middleware. We are therefore moving away from intelligent networks toward networked intelligence, in which any middleware functionality will itself be an agent. Managing this transition and developing an appropriate software model is a major open issue facing developers of agent-based systems.

SOCIAL BEHAVIOR

Developers clearly face many challenges in social, chaotic, temporal, and emergent behavior that must be resolved in order to realize agents' full potential. Other issues, already well known and outside the scope of this article, include shared ontologies, delegation of trust, uncertainty, learning, and interagent communication.

Agent 'Police'

Agent compliance with a standard will never be easy to test, and an intelligent entity possesses the freedom to be "economical with the truth"—perhaps omitting key facts—in satisfying its goals. Therefore, the monitoring and policing of external and observable agent behavior remains an open issue.

One form of monitoring agent behavior is self-policing: Agents are implemented such that actions leading to potentially harmful social behavior are short-circuited out of the reasoning process. A companion approach involves sentinel agents that patrol antisocial behavior, which raises the problem of defining antisocial behavior and methods of sanctioning it. The importance of addressing these issues will grow as larger open-agent communities develop.

Balancing emergent behavior against application constraints is a key issue in agent development.

Error Recovery

Errors, malfunctions, and generally chaotic behavior can occur anywhere in a distributed system, setting off a ripple effect that can cause serious, and widening, damage. Distributed models of computing must consider these issues seriously because of the inherent autonomy, concurrency, nondeterminism, asynchrony, and possible nontermination of distributed, emergent algorithms.

Agent designers should be more concerned about errors that might arise within the communicating agent community (assuming that all lower levels work well). Messages can be delayed, lost, bottlenecked within a busy agent, even misunderstood by one "stupid" agent in the midst of other good ones. The possibility of livelock in nondeterministic negotiation is also of concern.

Little is known of the pathology of agent communities. Therefore, something akin to system reboot, an action taken by human owners, will be the final recourse to recovery. However, we must take care to distinguish accidental failure from malicious behavior, and to recognize planned malicious behavior masquerading as a software bug.

Awareness of Time

Current definitions of communicative acts see acts performed by agents as isolated and without consequences, but consider the following scenario:

Suppose Agent A issues a bid request that fails to get any response from Agent B. Further, suppose that Agent A knows that Agent B has the requisite skills to respond to the bid (having done so in the past, say), and also that Agent B is still active (having just replied to another, unrelated query from Agent A). Then this deliberate nonresponse is also a communication of a kind, from which Agent A may be able to draw inferences.

What's more, the full effect of a communicative act may only emerge over time, after some sequence of acts between communicating agents. Because a series of communications defines the context of an agent interaction, an agent must track past communications. Furthermore, an agent might be involved in more than one thread, and these threads might also interact with one another. These problems require more research.

Emergent Behavior

Successful societies evolve. Given a communication language and proximity enabled by middleware, an awareness of "good" and "bad" behavior and of time (particularly the future), agents can be programmed to change their behavior. While an individual agent can learn to improve its performance, a society of agents can evolve to find a pareto optimal configuration for a certain task and environment. This is a form of emergent behavior in which individuals collaborate to achieve collectively what none could do alone.

Multiagent system design is software engineering based on process-oriented, rather than object-oriented, design. The power of the multiagent paradigm stems from the ability to continually reorganize, and if the multiagent system is extensible, to accommodate and exploit new agents with new functionality.

The risk of top-down design is that emergent behavior will be designed out, but to the extent that it is not, how is the resultant system verified? How is it visualized? How is the agent life cycle accommodated? In a safety-critical application, a group of learning agents reorganizing themselves for pareto optimal behavior might, to some, be less than reassuring. Balancing the power of emergent behavior against the application constraints is a key issue for future development of agent populations.

CONCLUSIONS

The limitations of software intelligence will influence how agent-based systems are designed. The asynchronous nature of agent relationships suggests that agenthood is assigned so that each agent retains sufficient autonomy over the tasks for which it is responsible. Thus, insofar as a developer is free to partition a system into individual

agents, partitioning should be done with autonomy (and consequently minimum interagent communication) as the criterion. Meanwhile, the developer must take into account agents developed and owned by other organizations, and should consider legal, formal, and related ownership matters as paramount design issues.

Lehman and Belady[5] classify software into three types:

- *S*-type software is precisely specified; thus, its correctness can be tested against the specifications. An example is a program designed to invert matrices.
- *P*-type software lacks a precise specification but relies upon a good model for its effective operation. An example is a chess-playing program.
- *E*-type software is embedded software whose sole purpose is to respond to external sensors and effect changes upon some external environment.

Software agents can be seen as *P*-type in the sense that they exhibit autonomous, proactive behavior and reason about the external environment. However, they can also be seen as *E*-type embedded systems in the sense that they are meant to be responsive (reactive) to the external environment. It is also possible that software agents are a novel type of interactive software.

If two communicating agents A and B respond to distinctly disjoint environment domains, then each is completely reliant on the other in dealing with the other's domain. However, if in the external world their domains overlap, the agents can note the communicative acts the other produced and observe how it behaved, and are thus able to verify the communications. Even if two agents do not directly share a common domain of discourse about the external world, they may do so indirectly via other agents with which they are communicating.

Finally, the practical issues of software agents must be grounded in the engineering aspects of computer science: its systems, processes, architectures, tools, and standards. If we can tackle the engineering of agent communities with full knowledge of the legal and social challenges, then that will help with the practical deployment of agents. ■

ACKNOWLEDGMENTS

This work owes much to discussion with many colleagues, and we are pleased to acknowledge their various contributions. We are especially grateful to Munindar Singh for his valuable and constructive comments, and to the anonymous referees. We also acknowledge the financial support of the European Union project IST-10298 ALFEBIITE (http://www.iis.ee.ic.ac.uk/alfebiite).

REFERENCES

1. D. Messerschmitt, "The Future of Computer-Telecommunications Integration," *IEEE Comm.*, vol. 34, no. 4, Apr. 1996, pp. 66-69.
2. K. Sycara, K. Decker, and M. Williamson, "Matchmaking and Brokering," *Proc. Second Int'l Conf. Multiagent Systems, (ICMAS 96)*, AAAI Press, Menlo Park, Calif., 1996.
3. T. Finin, Y. Labrou, and J. Mayfield, "KQML as an Agent Communication Language," in *Software Agents*, J. Bradshaw, ed., MIT Press, Cambridge, Mass., 1997.
4. Foundation for Intelligent Physical Agents, "FIPA 97 Specification Part 1: Agent Management System," Geneva, 1997; also available online at http://www.cselt.it/fipa/spec/fipa97/fipa97.htm.
5. M. Lehman and L. Belady, *Program Evolution: Processes of Software Change*, Academic Press, New York, 1985.

Ebrahim (Abe) Mamdani is a professor of electrical and electronic engineering at the Imperial College of Science, Technology, and Medicine, London, England. His research interests are in intelligent and interactive systems. He currently holds the Nortel Networks/Royal Academy of Engineering chair in telecommunications strategy and services at Imperial College and has acted as technical advisor to the Foundation for Intelligent Physical Agents (FIPA). Mamdani holds a BE and a PhD from London University. He is a Fellow of the IEEE and a Fellow of the Royal Academy of Engineering.

Jeremy Pitt is a lecturer in the Intelligent and Interactive Systems Group of the Department of Electrical and Electronic Engineering at the Imperial College of Science, Technology, and Medicine, London, England. He is principal investigator of the UK CASBAh project (funded by the UK Engineering and Physical Sciences Research Council and Nortel Networks), concerned with agent-oriented middleware for advanced telecommunications service delivery, and project manager of the European Union–funded project ALFEBIITE, investigating legal, ethical, and normative issues in multiagent societies. Pitt holds a BSc and a PhD from London University.

Contact the authors at Dept. of Electrical and Electronic Eng., Imperial College of Science, Technology, and Medicine, Exhibition Road, London, SW7 2BZ; {e.mamdani, j.pitt}@ic.ac.uk; http://www-ics.ee.ic.ac.uk/.