Agents for the Masses?

Jeffrey M. Bradshaw, Mark Greaves, and Heather Holmback, The Boeing Company
Tom Karygiannis and Wayne Jansen, National Institute of Standards and Technology
Barry G. Silverman, IntelliTek and the University of Pennsylvania
Niranjan Suri, Institute for Human and Machine Cognition, University of West Florida
Alex Wong, Sun Microsystems

IS IT POSSIBLE TO MAKE DEVELOPMENT OF SOPHISTICATED AGENTS SIMPLE ENOUGH TO BE PRACTICAL? THIS ARTICLE DISCUSSES THE AUTHORS' THEORETICAL AND TOOL-CREATION EFFORTS TO ANSWER THAT QUESTION IN THE AFFIRMATIVE.

AGENT TECHNOLOGY IS IN A state of paradox. The field has never enjoyed more energy and concomitant research progress, and yet the rate of uptake of new research results in fielded systems has been glacially slow. The few agents in the real world of everyday applications generate more heat than enlightenment; most are easily confused, few collaborate except in trivial prearranged fashion, and all enjoy little freedom of movement. Significantly, the current trapped state of our agents has less to do with lack of mobility mechanisms than with their unpreparedness to work fully in the open world of cyberspace and to interoperate outside a tightly circumscribed sphere of agent platforms and domains. The kinds of agents that we want—citizens of the wired world, equipped with stamped passports and Berlitz traveler's guides explaining foreign phrases and places that allow them to hail, meet, and greet agents of any sort in the open landscape of the Internet and, if not able to team up on a project, at least able to ask intelligibly for directions—these kinds of agents, alas, exist today only in our imaginations (and, of course, in the vision sections of our research proposals).

Actually building the sophisticated agent-based systems of the future will require research advances on at least three fronts:

- We must continue work on agent theory so that many currently unanswered questions about the scope and limitations of alternative approaches to agent design can be addressed.
- We must make agent frameworks and infrastructure powerful, interoperable, and secure enough to support robust large-scale coordinated problem-solving activity.[1]
- Perhaps more importantly, we must develop new sorts of tools to help nonspecialists unlock the power of agent technology.

The good news is that things are progressing well on the first two fronts. Various initiatives are beginning to provide an early preview of the faster, more reliable, and more secure versions of the next-generation Internet that our large-scale visions require. Middleware and Internet technologies and standards are now maturing to the point that agent framework developers can rely on off-the-shelf products as a ready substrate to their own work, rather than creating ad hoc alternatives from scratch. Advances in the difficult theoretical issues of dynamic agent communication, coordination, and control are beginning to let us better understand how to deploy large numbers of agents with confidence. Finally, recent work in theory and infrastructure has yielded exciting new kinds of blueprints for future systems that lie beyond the evolutionary development of current technologies. From grids[2] to Jini (see *http://java.sun.com/products/jini*), these approaches aim to provide a universal source of dynamically pluggable, pervasive, and dependable computing power, while guaranteeing levels of security and quality of service that will make new classes of agent applications possible.

However, a large and ugly chasm still separates the world of formal theory and infrastructure from the world of practical nuts-and-bolts agent-system development—this is where the third research front comes in. If agent technology is ever to become as widely used as ordinary object technology is today, we must create new sorts of tools to help nonguru developers bridge the gaps between the-

1094-7167/99/$10.00 © 1999 IEEE

Policies and exception handling: a fence and an ambulance

Twas a dangerous cliff, as they freely confessed,
Though to walk near its crest was so pleasant;
But over its terrible edge there had slipped
A duke and full many a peasant.
So the people said something would have to be done,
But their project did not at all tally;
Some said, "Put a fence around the edge of the cliff,"
Some, "An ambulance down in the valley.
(Joseph Malins)

With the potential for increased power and freedom that agent systems afford also come increased dangers and vulnerabilities. One of the most important recent developments in agent technology has been the growing momentum to find general management mechanisms for large-scale heterogeneous agent-based systems operating in open environments. It's a dangerous cliff. Do we need a fence or an ambulance? Or both?

With respect to the ambulance, Mark Klein and Chris Dellarocas at MIT are developing a shared exception-handling service for agents.[1] They characterize this service as a kind of *coordination doctor*: "it knows about the different ways multiagent systems can get 'sick,' actively looks system-wide for symptoms of such 'illnesses,' and prescribes specific interventions instantiated for this particular context from a body of general treatment procedures." The hope is that this approach will both simplify agent development and make exception-handling behavior more effective and tunable.

We believe that the policy-based fences for agent communication and management advocated in this article can complement and enhance an exception handling approach. We are collaborating with MIT on an experimental testbed that will integrate our KAoS agents and conversation policies with their exception-handling mechanisms. The policies governing some set of agents aim to describe expected behavior in sufficient detail that deviations can be easily detected. At the same time, related policy support services help make compliance as easy as possible. Standing between the policy support and exception handling services, shared enforcement mechanisms operate as a sort of "cop at the top of the cliff" to warn of potential problems before they occur. When, despite all precautions, an accident happens, the services of the exception handler are called as a last resort in to help repair the damage. In this manner, the policy-based fences and the exception-handling ambulances work together to ensure a safer environment for agent systems.

Beyond these initial considerations, a policy-based approach affords other advantages, such as: reuse, efficiency, context-sensitivity, and verification.

Reuse. In the domain of agent conversation, the requirement for reusable policies has manifested itself with different names and somewhat different concepts (such as FIPA's *interaction protocols*,[2] Jackal's *conversation specifications*,[3] and COOL's *conversation plans*[4]), but the current acceleration of convergence is heartening.[5,6] Policies encode successful patterns among agents and their platforms, packaging them in a form when they can be easily reused as occasion requires. We do something similar in human discourse when we adopt rules of parliamentary procedure as a way to structure a formal debate. Though the relationship among the debating parties may be adversarial, there is a mutual recognition that adopting common ground rules is in everyone's best interest. While the rules restrict our freedom and perhaps do not perfectly apply to every situation, we have the benefit of knowing that they have been tested and modified over many years to facilitate greater efficiency and fairness. By adopting such sets of rules when they apply, we reap the lessons learned from previous analysis and experience while saving ourselves the time it would have taken to reinvent them from scratch.

Efficiency. In addition to lightening the agent designers' workload, explicit policies can often increase the runtime efficiency of the agents themselves. For example, conversation policies reduce the agents' in-

ory, plumbing, and practice. Currently, full appreciation of leading-edge developments in agent theory and frameworks requires sophisticated knowledge of speech-act theory, formal semantics, linguistic pragmatics, logic, security design, Internet and middleware technologies, distributed computing, planning, and other disciplines that are not likely to be present in a typical developer's skill set. Without good tools, rapid advances in theory and infrastructure might paradoxically attenuate rather than accelerate the adoption of agent technology as members of the developer community spin their wheels or ultimately give up in disgust. Hence we must ask, is it possible to make development of sophisticated agents simple enough to be practical?

Fortunately, the agents community has not completely neglected the question of tools.[3] The DARPA CoABS program[4] and complementary initiatives in Europe and Asia are vigorously supporting research to accelerate the development of scalable interoperable agent theory and tools, and are promoting the eventual adoption of standards through bodies such as FIPA and the OMG. As part of these efforts, we are working to extend theory and create tools in two areas: agent communication and agent management. Common to both areas is the theme of using explicit representations of agent *policies* to help make interactions among agents and with their environment more simple and reliable (see the "Policies" sidebar). This article discusses our current research directions and preliminary results in each of these areas.

Theory and tools for agent communication

One of the few constants across competing accounts of agenthood is the emphasis on an agent's capability to coordinate with other agents through an explicit interagent communication language. This capability is vital to the promise of large-scale interoperability: the idea that sets of independently developed agents will be able to usefully work together. Conversely, we can trace many of the difficulties in assembling reliable, flexible, interoperable agent systems to an agent's communication subsystem. The problems span the gamut from differing interpretations of the core messaging protocols to inconsistent ontological frameworks and disagreements about the meanings of larger message sequences. One way to address these problems, and thereby to promote agent interoperability, is to create standards-based tools that make it easy for agent developers to "do the right thing" and create agents that operate using a common set of communication assumptions. The tools would facilitate interoperability and ensure robustness by generating appropriate conversation policies. As we'll show, we have a concept for one type of tool that would help developers using the same *agent communication language* to guarantee that their message sequences are consistent and conform to the semantics of the ACL.

From agent messages to agent conversations. Our working hypothesis over the past few years has been that agent communication is better modeled when conversations rather than isolated messages are taken as the primary unit of analysis.[5] As Terry Winograd and Fernando Flores observe,

> The issue here is one of finding the appropriate domain of recurrence. Linguistic behavior can be described in several distinct domains. The relevant regularities are not in individual speech acts (embodied in sentences) or in some kind of explicit agreement about meanings. They appear in the domain of conversation, in which successive speech acts are related to one another.[6]

ferential burden in comparison to unrestricted agent dialogue by limiting the space of alternative conversational productions that they need to consider. Because a significant measure of conversational planning for routine interactions can be encoded in conversation policies offline and in advance, the agents can devote more of their computational power at runtime to other things. Thus, the goal for conversation policy representation and implementation is to find a sweet spot somewhere between the extremes of the fixed protocols of typical distributed software on the one hand and something approaching completely unstructured freeform human dialogue on the other. Similarly, with respect to agent management policies, we want to express the policy formulator's intent at an optimum level somewhere between today's complex and limited Java security and resource management mechanisms and an overly simple global "switch" with high, medium, and low settings.

Context-sensitivity. Throughout this article, we have emphasized the importance of pragmatic considerations as they apply to agent behavior. The ability to selectively sense and contextually react to its environment is a hallmark of agent behavior. Explicit policy representation improves the ability of agents and agent platforms to be responsive to changing conditions, and if necessary reason about the implications of the policies which govern their behavior. As mentioned elsewhere in the article, the Java platform itself has moved in this direction by moving security preferences from their implicit representation in code to an explicit external representation in policy. In principle, this allows certain agent rights and privileges to be granted and revoked at runtime through reinitialization of the policy object.

Verification. By representing policies in an explicitly declarative form instead of burying them in the implementation code, we can better support important types of policy analysis. First—and this is absolutely critical for security policies—we can externally validate that the policies are sufficient for the agent's tasks, and we can bring both automated theorem-provers and human expertise to this task. Second, there are methods to ensure that agent behavior which follows the policy will also satisfy many of the important properties of reactive systems: liveness, recurrence, safety invariants, and so forth. Finally, with explicit policies governing different types of agent behavior, we can begin to understand and predict how policies would compose with one another, and how we might automatically generate code to implement a given policy.

In the context of the DARPA CoABS program, we are partnering with researchers at MIT, University of Massachusetts, and Cycorp in a longer-term effort to understand the interplay among agent control and conversation policies and mechanisms. In the coming months, we expect to have initial experimental results.

References

1. M. Klein and C. Dellarocas, "Exception Handling in Agent Systems," to be published in *Proc. Autonomous Agents '99*, ACM Press, New York, 1999.
2. FIPA 97 Specification, 1997; www.fipa.org.
3. R. S. Cost et al., "Jackal: A Java-Based Tool for Agent Development," *Proc. 1998 Nat'l Conf. Artificial Intelligence (AAAI-98) Workshop on Software Tools for Developing Agents*, AAAI Press, Menlo Park, Calif., 1998, pp. 73–82.
4. M. Barbuceanu, "Coordinating Agents by Role-Based Social Constraints and Conversation Plans," *Proc. AAAI '97*, AAAI Press, 1997, pp. 16–21.
5. M. Greaves, H. Holmback, and J.M. Bradshaw, "Agent Conversation Policies," *Handbook of Agent Technology*, J.M. Bradshaw, ed., AAAI Press/ MIT Press, Cambridge, Mass., 1999.
6. M. Greaves, H. Holmback, and J.M. Bradshaw, "What Is a Conversation Policy?" to appear in *Proc. Autonomous Agents '99 Workshop on Specifying and Implementing Conversation Policies*, ACM Press, 1999.

Informally, we can think of agent conversations as sequences of messages involving two or more agents that are intended to bring about a particular set of (perhaps jointly held) goals. So, for example, a conversation between a purchasing agent and a supplier agent might be intended to further the shared goal of continuing an existing business relationship, and also further individual profit-maximization goals. This focus on conversational goals, rather than on particular syntactic message-exchange patterns, marks an important difference from the way developers think about typical computing communication protocols. Individual agent conversations might admit side conversations and a great deal of flexibility in message content and sequencing while remaining consistent with the conversational goals and the rules of semantics and pragmatics such as those we discuss in the next subsection. Developers thus must explicitly consider issues related to planning rules and goal-directed behavior when designing agent communication.

Conversation policies. Although human conversation normally proceeds quite effectively without spelling out specific rules and hierarchies in advance, as agent developers we have found it useful to define prescriptive *conversation policies*. In practice, this means that before entering into a conversation, the agents involved first mutually agree on a given conversation policy—or set of policies—that will structure their interaction. Typically, these policies will have been previously defined by an agent developer and placed in a commonly accessible library (although they could of course have been coded into the behavioral logic of the agents themselves). Once the governing conversation policies have been agreed to, the agents involved abide by the constraints defined in the policy for the duration of that conversation.

We define conversation policies to be sets of declaratively specifiable constraints on agent conversation that can abstract from some of the details of the particular ACL and agent implementation.[7] Specific instances of agent conversations relate to their governing policies as a token relates to its type. The constraints that make up a policy can range from the very general (for example, that any contract conversation must include a negotiation phase) to the very specific (in KQML, an `ASK-IF` message type must be followed by a `TELL` message type or an error primitive). An interesting class of policies is concerned with semantic message constraints, because they involve restrictions on the meanings that individual messages can be used to convey at each stage of the conversation. For example, in an auction, one policy might restrict a price message to mean that the agent is committed to paying the price if the bid is accepted. We can also identify distinct policies that regulate pragmatic issues in communication, such as interruption and turn-taking behavior.

In the past, we have used small augmented

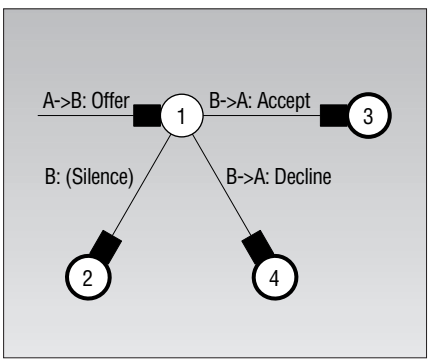

Figure 1. The KAoS Offer conversation policy. Note that "silence" is not a communicative act in the way we have been using the term.

finite-state machines to represent the allowable speech-act sequences in our conversation policies (see Figure 1). FSMs are easy to conceptualize and implement, and might be adequate for the routine interaction of many kinds of simple agents. However, they have limited ability to express many kinds of constraints relevant to conversations (for example, higher-level goal structures or information about overall timing and security). More expressive formalisms for conversation policies can be constructed out of statements in a suitable dynamic logic; less expressive (but perhaps more readily understandable) formalisms could be built out of regular expression grammars. Our Conversation Design Tool, which we describe later, is a tool for designing and verifying certain classes of conversation policies.

Assuming a suitably expressive constraint representation, we could easily structure conversation policies to allow for some degree of emergence. Rather than specifying the exact sequence and type of messages involved, such a policy would contain only high-level constraints. The resulting looser control of a conversation governed by this kind of policy would allow great flexibility for the agents involved, while of course requiring a greater sophistication and shouldering of computational burden on their part. More specifically, such a flexible policy might describe a relative sequence or pattern of *landmarks* (for example, an offer has been made; an offer has been accepted) in a conversation of a given type, each landmark defining a set of specifiable properties that must hold the agents involved at that point in the conversation, and the overall policy simply requiring that the transitions between conversational landmarks be made by an appropriate sequence of one or more communicative acts.

For example, a segment of the KAoS Offer conversation policy shown in Figure 1 involves an offer made by some agent A, immediately followed by an accept or decline by some other agent B.[5,8] While it is reasonable to think of offering and accepting as typically being a two-step process, this might not always be the case: between A's offer and B's acceptance, B might ask A for a clarification about payment, or how long it will take A to do the task, or if the offer will still be around if B delays acceptance. In these cases, there could be any number of subsequent exchanges between A and B until the acceptance (or nonacceptance) of the original offer. While the KAoS framework cannot completely specify actual emergent dialogues ahead of time as to the number and exact sequence of messages involved, it can still describe them as a relative sequence or pattern of messages *(*for example, it cannot accept an offer until one is made) and will have some restrictions on their structure (general provisions for turn-taking) and content (a requirement for explicit expiry conditions on the offer).

In the KAoS agent framework's current version,[5] the responsibility for conversation management is shared between two parts of each agent: the *conversation handler*, which the framework supplies along with other components of the *generic agent*, and the agent-specific *extension*, which the agent developer supplies. The conversation handler determines what conversational moves are allowed given the particular conversation policies currently in force and the conversation history to that point, and the extension deliberates among the possible options supplied by the conversation handler. Once the agent's extension has formulated a message to send, its conversation handler makes sure that the message complies with one of the allowable options. The conversation handler also handles any unforeseen errors and exceptions that might happen during message transport.

The role of pragmatics in agent conversation. Although current work on the semantics of basic communicative acts and team behavior provides a good starting point for conversation policy designers,[8,9] agent researchers have generally neglected the important role of pragmatics in agent communication languages. We believe that the considerations of pragmatics are important enough to warrant detouring from our main line of discussion to elaborate on them (see the "Pragmatics" sidebar).

Description of the conversation design tool. We are designing and developing the conversation design tool described in the "Pragmatics" sidebar as a specialized type of *heterogeneous reasoning system*.[10–13] In brief, an HRS is a composite formal reasoning support system that includes multiple-component logical subsystems, each with its own syntax, semantics, and proof theory. The HRS also includes inference rules that operate between the different subsystems. The goal of an HRS is twofold:

- to provide a logically rigorous environment in which a user's reasoning using multiple different representations can proceed simultaneously, and
- to support the logically sound transfer of intermediate results among the component reasoning systems.

An HRS is not itself an automatic theorem prover, although it can incorporate components that implement automated reasoning over some defined logical fragment. Rather, an HRS is a formal reasoning environment in which a user can employ the resources of several different representation and inference systems in order to reason about a particular domain.

Stanford University is developing the OpenProof system as an extensible framework that allows a logician to build a custom HRS for an identified domain. It is implemented as a collection of Java Beans, allowing additional user-defined deductive subsystems to be smoothly integrated. OpenProof currently includes implementations of several types of simple logical subsystems. These include both graphically based logical subsystems (for reasoning involving, for instance, Venn diagrams and blocks worlds) and sententially based logical subsystems (for reasoning using the representations of classical first-order and modal systems). More importantly, though, OpenProof also includes a sophisticated framework for linking the various component subsystems together to yield heterogeneous proof trees. Its design supports adding proof managers for different logics, and also supplies a method to define inference rules that bridge the different deductive subsystems.

The CDT will bind together a particular, identified set of logical subsystems that are found to be useful for reasoning about and modeling conversations in particular ACLs. Because part of our research involves identifying these useful deductive subsystems, the precise collection of components in the CDT has not yet been finalized. Our strategy will be to select a base set of logical subsystems for the CDT, and to evaluate this selection using a group of KAoS developers at Boeing. On the basis of their feedback about usability and appropriateness to their problem domain, we will expand and modify our base set of representations. In its initial incarnation, the CDT will provide the following types of deductive systems to its users:

- *A natural deduction reasoning system for standard first-order logic.* This will let

Pragmatics in agent communication

Pragmatics is implicitly a part of the semantic analysis of agent communication languages in that the semantics of message types (for example, `request` and `tell`) is typically based on the pragmatic analysis of the ACL speech act designators associated with the eponymous natural language performatives.[1] (We note, however, that specifying the semantics of the communicative act designators in *not* equivalent to specifying the semantics of the ACL expressions that use these designators.) As we will argue, interoperability is not just a matter of using the same set of speech act designators with the same meaning and a common syntax for the messages that include them. It also involves being as cooperative and sensitive to context as possible so that the intended meaning of an ACL message can be correctly and efficiently inferred. This is one of several areas where an understanding of the role of the pragmatics can provide valuable guidance to agent designers.

Developers of current ACLs have exploited the parallels between agent communication and human communication in developing the syntax and semantics of their languages. The area of syntax is relatively noncontroversial, the major ACL-related requirement being that the syntactic scheme adopted be sufficiently expressive to capture the structure of what is being communicated. The major new development in ACL syntax is that, due to the ubiquity of XML content and ready availability of XML parsers, developers are increasingly abandoning Lisp-like ACL syntax in favor of richly structured markup languages.

In contrast, the area of semantics has generated a diversity of approaches.[2-4] The approach we describe in this article is consistent with the Cohen-Levesque analysis of joint intentions, although some of the concepts could certainly be adapted to other types of semantics.[2,5] Joint-intention theory supplies a denotational semantics based on ascribing certain mental states to the communicating agents. A basic principle of this style of semantics is *compositionality,* meaning that we can derive the semantics of a complex act from the semantics of the acts that are its syntactic components. Using this kind of model, we can define an ACL that allows for principled extensibility. In other words, we can define a language with a small core set of operators and define additional operators in terms of this core set; the semantics of the newly added operators are defined to be the compositional result of the component core operators in its definition. This type of semantics stands in contrast to the original operational semantics proposed for KQML.[3,6] Because KQML's message types were defined independently, it is not possible to determine the exact relationship between a pair of messages, or to understand how new messages can be reused in existing conversation structures.

Both semantics and pragmatics are involved in the meaning of ACL expressions. (Note that the entire ACL expression, not just the communicative act, is in the proper subject of pragmatic analysis.) In the study of natural languages, semantics is often viewed as an account of the core truth conditions of a sentence—the basic conditions under which the sentence or the proposition it expresses is true independent of context. An expression's semantics thus defines its literal meaning—that aspect of a sentence's meaning that is common across every context of usage. The pragmatics of natural-language expressions, on the other hand, is concerned with that aspect of meaning that arises from *the context of use*, and how that context contributes to both the total meaning and the effects of an utterance. For example, the statement "It is cold in this room" has a syntactic analysis and a literal, semantic meaning that is constant across all of its possible uses,[7] namely that the temperature in the room is cold relative to the speaker. However, given a specific context, the sentence could be used to state a fact, request that the listener close a window, warn the listener not to enter the room, or some combination of the above.

Although analyses of some of the KAoS core conversation policies and the application of the joint-intention theory semantics to conversation policy design have been published previously,[8] we have only recently begun to investigate the role of pragmatics.[1] Like many agent communication languages, KAoS makes a distinction between communication, content, and contextual portions of agent messages. The communication portion encodes information enabling proper message routing, such as the identity of the sender and recipient. The content portion contains the actual gist of the message (for example, the specific request or information being communicated). The contextual portion describes the act intended by sending the message (this is done by tagging the message with a communication act designator such as `request` or `inform`), a reference to the governing conversation policy or policies, a unique identifier for the conversation instance, and the various conversation option parameters proposed or currently in force. In addition, this message context might also contain other descriptive information useful for the agent interpreting the message, such as the language used to express the content and references to particular ontologies associated with it.

One way we think pragmatics comes into play in ACL expressions is in the selection of an appropriate communicative act designator to convey the agent's intention or purpose for a particular message in the context of a conversation policy. Given, for example, that a certain agent wants to request X, it seems intuitive that the best way to do this would be through an ACL expression with something like `REQUEST` as the communicative act designator and a description of X as the propositional content. However, there is no reason from a strictly semantic or syntactic perspective why the agent could not instead use `INFORM` as the communicative act designator and something like "I ask you to give me X" or even "I want you to give me X" as the propositional content to perform the request. Because these two forms are semantically interchangeable, it is a *pragmatic* principle of ACL usage that would account for one or the other messages (typically the one with `REQUEST`) being preferred.

The philosopher H.P. Grice suggested that there is an overall pragmatic principle—the Cooperative Principle—that speakers of natural languages implicitly follow as they engage in conversation. This principle relies on the assumption that communication succeeds because speakers and hearers assume they are cooperating to achieve communicative goals:

users of the CDT perform reasoning using classic natural deduction introduction and elimination techniques over a first-order language with equality. This subsystem will also include automatic theorem provers for this logic. OpenProof currently includes two such theorem provers: one based on resolution and one based on the matrix method, as well as a specialized variant of the matrix method prover that incorporates domain knowledge to increase its speed in a certain class of domains. We might extend this subsystem to support simple reasoning (though not theorem-proving) in modal logic, as many of the important semantic properties of agent conversations are naturally expressed via modalities.

- *A Petri net deductive system*. Petri nets are a common graphical formalism for modeling systems that are driven by discrete events and whose analysis essentially involves issues in concurrency and synchronization. They have a fairly simple and intuitive semantics based on state transition graphs, and a well-understood set of update rules. They are an important technical tool with which to investigate communication and coordination in agent systems. The basic CDT will contain a Petri net reasoning tool integrated as an OpenProof subsystem.
- *An enhanced Dooley graph deductive system*. Enhanced Dooley graphs are a variety of FSM diagram H. Van Dyke Parunak developed for the analysis of agent dialogue at a speech-act level.[14] They occupy an attractive middle ground between FSMs that label nodes with par-

> Make your conversational contribution such as is required, at the stage at which it occurs, by the accepted purpose or direction of the talk exchange in which you are engaged.[9]

The Cooperative Principle is admittedly vague and does not by itself account for much of the form and content of human conversation. However, since the time this principle was first formulated it has become generally accepted that speakers and hearers implicitly follow this and related principles when they use and interpret natural language.

Meaningful agent conversation is also subject to pragmatic principles. What might these pragmatic principles be? First, we think it is reasonable to assume that agents are at their most cooperative when they are most direct in their communication. (Note that this assumption is not always valid for human communication where conditions such as politeness enter in.) We propose the following *agent cooperative principle* as a specialization of the Cooperative Principle for agents using speech-act-based ACLs:

> Make your conversational contribution as direct as possible by using an ACL expression whose communicative act designator *most directly* represents the intended communicative act.[1]

For the ACP to be useful, it is necessary to give criteria for determining what is the most "direct" communicative act designator in a given context. One promising idea involves an analysis of the logical entailment relations between different communicative acts (illocutionary acts) with respect to their conditions of success and satisfaction. (The conditions of satisfaction are based on the semantics of the communicative act and constitute those conditions that must obtain in a context for the act to count as having been fulfilled in the context.) For example, Daniel Vanderveken notes that

> many illocutionary acts have *stronger conditions of satisfaction* than others, so that whenever they are satisfied in a possible context of utterance, the other illocutionary acts are also satisfied. For example, if a promise to be nice is kept then the assertion that the speaker can be nice is *eo ipso* true.[10]

A speech act whose formal conditions of satisfaction are stronger than those of another speech act would thus be the more direct. As applied to agent communication, consider the following abstract ACL expressions:

`REQUEST`: Send me four widgets by tomorrow
`INFORM`: I want you to send me four widgets by tomorrow

The ACL expression with `REQUEST` has stronger conditions of satisfaction than the ACL expression with `INFORM`, because the conditions of satisfaction of the former include those of the latter (that is, part of satisfying a request to send four widgets is that the sender is informed of the receiver's desire for four widgets, but not vice versa). Thus, if the goal is to be sent the four widgets, then the ACP would dictate that `REQUEST` should be the ACL message chosen. However, as should be obvious, if A's communicative intent is to just make its desires known to B, then the `REQUEST` is too strong for the communicative intent and an `INFORM` should be sent. In this fashion, the ACP, in conjunction with criteria for judging directness, provides principled grounds for selecting one type of message over another.

Following the ACP necessarily limits the freedom of the developer or agent by restricting the kind of content that can be put in a given message. For example, it would exclude the less direct requests within the content of an `INFORM` message mentioned above. But recall that one of the purposes of making the communicative act designator explicit within ACLs was to lessen the inferences necessary by the message recipient.[11] Without the limitations provided by the ACP, this burden might well increase because the content language might be powerful enough to express any number of indirect communicative acts. Previous analysis has shown that this can happen in subtle ways.[8] We acknowledge that the pragmatic restrictions provided by the ACP definitely involve a trade-off of this sort over the expressive power of the agent content language, but see this as a necessary consideration for agent researchers interested in the practicalities of fielding agent systems in large-scale applications.

Beyond the ACP, pragmatic considerations determine how contextual factors (for example, time constraints, resource availability, trust between the participating agents) come into play in agent conversations and con-

ticipants and FSMs that label nodes with dialog states. M.P. Singh has done some recent work using EDGs as a descriptive mechanism for his ACL semantic theories.[15] We are hoping to implement an EDG subsystem and editor as an OpenProof plug-in module for the CDT.

Once the CDT's logical subsystems are fully developed, we will populate them with the necessary aspects of semantic and pragmatic theory to allow reasoning about various ACL properties. For example, one of our initial goals is to explore the logical consequences of the theory of joint intentions.[8,16] We would like to verify, for example, that the semantic properties of formalized KAoS basic conversation policies would result in team formation. The results of these and other analyses will serve as the foundation for a library containing a starter set of proven conversation policies that agent designers can reuse or specialize. When agent designers using CDT are satisfied that the models they have built adequately capture their assumptions and intent for agent communication within a particular setting, they will use the CDT to generate the actual conversation policy definitions required.

The CDT's integration of an appropriate set of representational and deductive tools along with the availability of a starter set of proven conversation policies will make the job of designing sophisticated agents easier. Currently, reasoning about ACLs requires familiarity with modal logic, plus a fair amount of comfort with formal proof techniques. However, by importing these same problems into the reasoning environment of the CDT, we hope to see decreases in proof complexity, coupled with increases in proof readability and usability for those not trained in logic. Essentially, we believe that using structured graphics, or *mediating representations*,[17–19] to carry part of the cognitive load in reasoning will result in a simpler and more intuitive environment in which to explore the conversational possibilities in a given ACL. Because of this, we hope that agent designers and developers who have had only a minimum of training will be able to use the CDT to explore and verify individual conversation policies.

Framework and tools for agent management

The second part of our research is aimed at creating agent management tools based on policy-based mechanisms. This will require substantial enhancements to basic Java security and resource management platform features to better support agent technology requirements. Although Java is currently the most popular and arguably the most security-conscious mainstream language for agent development, current versions fail to address many of the unique challenges posed by agent software. While few if any requirements for Java security and resource management are entirely unique to agent software, typical approaches used in nonagent software for defining and executing security policies and mechanisms are usually hard-coded. They do not allow the degree of on-demand config-

versation policies. Contextual factors might affect the selection or composition of appropriate conversation policies, the relevance and sequencing of individual messages within those policies (and whether or not certain kinds of optional message insertions are allowed), the number of iterations in an iterative sequence (for example, offer, counter-offer, counter-counter-offer, and so forth), the need for agreement on explicit nonperformance penalties, and the maximum duration of the overall conversation. Some pragmatic conditions are unique to particular kinds of messages. For example, in human conversations offers typically have implicit or explicit expiration conditions. Identification of the types of usage conditions that are likely to be part of a given message type will reduce the likelihood of errors of omission by conversation-policy designers.

Other pragmatic considerations cut across all conversation-policy types. For example, when communicating over noisy or unreliable radio channels, people use explicit requests for acknowledgment ("Do you copy?"), special conversational delimiters ("over") and stereotyped opening and closing statements ("over and out") to compensate. Other mechanisms come into play when adults converse with small children who might or might not understand what they are being told. Similarly, in agent applications certain agents might require more frequent acknowledgments that their messages have been heard and understood, depending on the moment-to-moment reliability of the communications channel and mutual perceptions of competence among the communicating agents. Another general pragmatic issue is to determine how agents interrupt each other. Though some agent conversation policies might not allow intentional interruption at all, those that do should include a specification of when and how the interruption can occur and the available mechanisms for recovery.

Given that pragmatics and semantics can both contribute to the design of conversation policies, the core question of this article arises anew: What kind of tools can we provide that will assist agent conversation policy developers who are not specialists in semantic and pragmatic theory? We have begun the development of a *conversation design tool* as a first prototype of such a tool.[12] Though as of this writing, the CDT does not exist as an integrated whole; important pieces of it have been written over the years in the course of various other projects.

References

1. H. Holmback, M. Greaves, and J.M. Bradshaw, "A Pragmatic Principle for Agent Communication," to be published in *Proc. Autonomous Agents '99*, ACM Press, New York, 1999.
2. P.R. Cohen and H. Levesque, "Communicative Actions for Artificial Agents, *Software Agents*, J.M. Bradshaw, ed. AAAI Press/MIT Press, Cambridge, Mass., 1997, pp. 419–436.
3. Y. Labrou, *Semantics for an Agent Communication Language*, doctoral dissertation, Univ. of Maryland, Baltimore County, 1996.
4. M.P. Singh, *Conceptual Foundations for Agent Communication Languages: Evaluation Criteria and Challenges*, tech. report, Dept. of Computer Science, Univ. of North Carolina, 1998.
5. P.R. Cohen and H. Levesque, "Intention is Choice with Commitment," *Artificial Intelligence*, Vol. 42, No. 3, 1990, pp. 213–261.
6. Y. Labrou and T. Finin, "A Semantics Approach for KQML—A General Purpose Communication Language for Software Agents," *Proc. Third Int'l Conf. Information and Knowledge Management*, ACM Press, 1994, pp. 447–455.
7. J. Searle, "Indirect Speech Acts," *Syntax and Semantics 3: Speech Acts*, P. Cole and J.L. Morgan, eds., Academic Press, New York, 1975.
8. I.A. Smith et al., "Designing Conversation Policies Using Joint Intention Theory," *Proc. Third Int'l Conf. Multi-Agent Systems (ICMAS-98)*, IEEE Computer Society Press, Los Alamitos, Calif., 1998, pp. 269–276.
9. H.P. Grice, "Logic and Conversation," *Syntax and Semantics 3: Speech Acts*, P. Cole and J.L. Morgan, eds. Academic Press, 1975.
10. D. Vanderveken, *Meaning and Speech Acts*, Cambridge Univ. Press, Cambridge, UK, 1990.
11. M.R. Genesereth, "An Agent-Based Framework for Interoperability," *Software Agents*, 1997, pp. 317–345.
12. M.T. Greaves et al., "CDT: A Tool for Agent Conversation Design," *Proc. Nat'l Conf. AI (AAAI-98) Workshop on Software Tools for Developing Agents*, AAAI Press, 1998, pp. 83–88.

urability, extensibility, and fine-grained control required by agent-based systems. Moreover, at first glance, there are seemingly opposing demands to be reconciled. The need for people to be in control of software running on their machines argues for hiding agent management policies and mechanisms from agent software and putting them fully under human supervision. On the other hand, the need for agents to act autonomously in the face of moment-to-moment contingencies argues for exposing at least some control mechanisms to trusted agents who can partially assume the responsibility for agent management.

The model in Java is rapidly evolving to provide the increased flexibility and fine-grained control required for agents. Early versions of Java featured a typed pointerless virtual machine instruction set, a bytecode verifier, class loaders, a security manager, and the concept of a "sandbox" to prevent applets from accessing "dangerous" methods. Version 1.1 added an API for user security features such as signing of JAR archives. A major feature of the security model in the Java 2 release is that it is permission-based. Unlike the previous all-or-nothing approach, Java applets and applications can be given varying amounts of access to system resources, based upon security policies created by the developer, system or network administrator, the end user, or even a Java program.

Despite these improvements, much work remains for creating robust extensible industrywide agent-specific management policies, mechanisms, and tools that can accommodate the most demanding of agent application settings. Moreover, as in the domain of agent conversation, we assume that the design and operation of secure agents will be done increasingly by people without specialized backgrounds. Hence our motivation to research and develop a prototype *agent management tool* (see the "Agent management tool" sidebar).

The AMT and accompanying framework will provide for at least the following basic scenarios.[20,21] A public-key infrastructure would be provided whereby two arbitrary agents could reliably authenticate each other's identity and the authority by which they are acting. Standardized message-encryption mechanisms would allow arbitrary sets of agents to safely exchange confidential information. The resource use of mobile agents can be guaranteed or constrained at a fine-grained level at design time or runtime, and can also be accounted for by the hosting agent system. Through the use of secure transparent Java checkpointing, "anytime" agent mobility would be transparently available at the demand of the server or the agent rather than just as specifically pre-determined entry points. At runtime, various levels of monitoring and dynamic control would be available to track and manage agent behavior and resource consumption.

Authentication and encryption. Java's security model currently emphasizes static control of resource access according to the source of a class file more than it does dynamic control based on authorized roles or the identity of individuals. While suitable for short-lived pro-

Agent-management tool

In the AMT, we aim to provide a graphical interface for the configuration of security policies for agents and hosts. Unlike the basic `policytool` Java currently provides to assist users in editing policy files, we are enhancing the initial AMT implementation to contain domain knowledge and conceptual abstractions to allow agent designers to focus their attention more on high-level policy intent than on the details of implementation. For example, resource usage policies for memory, threads, and file space may be specified by simply typing limit parameters into the appropriate graphical field. Mobility policies describing the conditions under which the host is permitted to move the agent can be specified, ranging from informed consent, to notification, to complete transparency to the agent being moved.

Domain knowledge in the AMT can help agent designers determine what kinds of policies are appropriate for a given situation. For example, when does it makes sense for an agent to be mobile, and, in the case of several agents acting on behalf of a single principal, which agents should, could, and should not be mobile? Another special problem involves mobile agents that would be at risk from potentially hostile hosts when traveling with private keys. Given the state of current agent frameworks, we do not think that an agent should ever carry private keys when it travels, as safer alternatives can be found for most motivating situations. For instance, if the desire to carry a private key is to increase fault tolerance (that is, in case the machine where the agent's private key is stored fails), perhaps it would be better to have another agent on another machine under the same security domain provide signing services on the mobile agent's behalf. Thus it may make sense for *negotiation agents* to be mobile, while *contracting agents* remain static so they can be available to review and sign agreements in a secure local environment.

We are exploring policies for various configurations of hybrid static-mobile agent ensembles to determine optimal policies and performance trade-offs. In our *agent-minion* approach, simpler lightweight mobile agents (minions) are sent on missions to perform shorter-term tasks by more intelligent static agents. Jini could provide an infrastructure for the distribution of minions, while higher-level coordination and management mechanisms would be used at the agent level.

The AMT prototypes we have been building also provide a graphical interface for the monitoring, visualization, and dynamic control of resource usage at runtime so that certain agents in an application can have greater access to resources than others. The goal of the runtime interface is threefold: to guarantee some specified level of agent access or quality of service to agents providing critical functions; to minimize the possibility of unauthorized access or reduce the impact of denial-of-service attacks; and to provide the possibility of detailed resource accounting. We are exploring the tradeoff between intrusiveness on the agent and level of control. For example, our most basic level of management would allow limited control of agents using mechanisms that would work regardless of how the agents were coded; additional levels of management would require minimal code modifications to support advanced features (for example, certificate management) but would in return allow the agent more full resource access and perhaps better performance. Use of a special VM such as Aroma could be regarded as a high level of intrusiveness on an agent, but would allow the finest grain possible of resource monitoring and control without necessarily requiring any code changes by the agent developer.

grams whose instances do not differ significantly from one another, such a model is insufficient for long-lived agent programs whose shifting roles and accumulating knowledge require strong authentication mechanisms that can identify precisely which unique instance is requesting access rights or communication privileges.

We currently base our agent authentication and encryption mechanisms on the extensive Java support for public-key certificate technology.[22] Public-key technology is an advance over conventional forms of protection—because certificates are public information, no sensitive data (for example, passwords or private keys) need ever flow over the network where it could be intercepted. Additionally, a chain of certificates can be used to establish the line of authority of a particular agent and to resolve liability issues resulting from misbehaving agents. In our framework, JAR files containing agent code are digitally signed using a certificate issued by a trusted third party. Additionally, agents who desire secure transmission of confidential information may communicate using a *Secure Sockets Layer,* which, following an exchange of certificates between the agents, sets up an encrypted channel for messages between them.

We are studying various approaches to building a complete *public-key infrastructure* tailored for agent systems that would provide management of keys and certificates. Beyond the basic functions, the PKI must also provide support for certificate revocation, key backup and recovery, and updating of key pairs and certificates. We also plan to evaluate the notion of flexible, adaptive management of agent policies and privileges by attribute certificate chains. Our work will complement that of other agent researchers seeking to extend ACLs with additional language security primitives—effectively allowing agents to communicate and reason about security policies by elevating the visibility of selected security mechanisms to the knowledge level.

Host-resource management. Mechanisms for monitoring and controlling agent use of host resources are important for three reasons:[20]

- It is essential that access to critical host resources such as the hard disk be denied to unauthorized agents.
- The use of resources to which access has been granted must be kept within reasonable bounds, assuring a specific quality of service for each agent. Denial-of-service conditions resulting from a poorly programmed or malicious agent's overuse of critical resources are impossible to detect and interrupt without monitoring and control mechanisms for individual agents.
- Tracking of resource use enables accounting and billing mechanisms that hosts can use to calculate charges for resident agents.

Resource-protection mechanisms are available at several levels to software developers, including those provided by the networking environment, hardware, the operating system, and the features of a high-level language. With agent technology, however, a persuasive case can be made for the advantages of an approach based on language-based protection primitives.[23] While such an approach limits the developer to a restricted set of languages that can be supported, the increased precision in specification of rights, the relative efficiency of rights amplification, the ability to analyze programs statically and not just at runtime, and the portability of the language-based approach argue strongly in its favor. Also, in the context of our work with DARPA and FIPA, most of the language-based mechanisms we describe below can be incorporated transparently into any Java-based agent framework with little or no code modification.

Given its status as the most popular and most rapidly evolving general-purpose safe language for Internet and agent applications, Java is the best first target for a language-based resource-management approach for the agent community. However, there is still

much to do to make it suitable for the industrial-strength agent applications of the future. Although the Java 2 security model is a step in the right direction, we anticipate that agent developers will require ever greater levels of flexibility and host systems will need ever greater protection against vulnerabilities that could be exploited by malicious agents. It is likely that some of these features will ultimately require changes to the Java architecture, such as the inclusion of an explicit Resource Manager to complement the current Class Loader and Security Manager.[24] For example, while new iterations of the Java security model will increasingly support configurable directory access by supplying the equivalent of access control lists to the Java Security Manager, there is no way to impose limits on how much disk storage or how many I/O operations or how many simultaneous print jobs might be performed by agents. Nor are there ways of controlling thread and process priorities, memory allocation, or even basic functions such as the number of windows that can be opened. A unique opportunity of our research is to explore techniques for dynamic negotiation of resource constraints between agents and the host. We are taking a two-pronged approach: one prong relying on features provided by standard Java mechanisms and security policies, and the other relying on whatever extensions are currently possible through clever programming.

The most important standard Java security mechanisms to exploit are permission classes and policy files. Unlike previous versions, the Java 2 security model defines security policies as distinct from implementation mechanism. Access to resources is controlled by a Security Manager, which relies on a security policy object to dictate whether class X has permission to access system resource Y. The policies themselves are expressed in a persistent format such as text so they can be viewed and edited by any tools that support the policy syntax specification. This approach allows policies to be configurable, and relatively more flexible, fine-grained, and extensible. Developers of applications (such as agents) no longer have to subclass the Security Manager and hard-code the application's policies into the subclass. Agents can make use of the policy file and the extensible permission object to build an application whose security policy can change without requiring changes in source code.

We have launched an effort to show how a limited form of dynamic security services for an agent host can be provided by standard Java security mechanisms. For example, although the virtual machine's Security Manager cannot be replaced at runtime, the instance of the policy class can. The permissions in the policy file can be rewritten at any time to reflect new contingencies such as events that require overall tightened or relaxed security restrictions. The policy object's `refresh()` method can then call for an immediate reload of the policy file and will immediately change its behavior. The policy object must be granted by the Security Manager to the calling code in the policy file and, of course, extreme care must be taken to protect this code from anything that might try to hijack its ability to modify the policy file.

WE HAVE LAUNCHED AN EFFORT TO SHOW HOW A LIMITED FORM OF DYNAMIC SECURITY SERVICES FOR AN AGENT HOST CAN BE PROVIDED BY STANDARD JAVA SECURITY MECHANISMS.

Approaches relying on standard Java security mechanisms can currently do little more than either grant or deny access to a particular service. A more sophisticated approach would guarantee some specified level of agent quality of service, minimize the impact of denial-of-service attacks, or provide meaningful resource accounting. To this end, we are investigating mechanisms and tools for more adequate resource management. Some of these mechanisms, such as load-time byte code rewriting, can be implemented without severely impacting performance or requiring changes to the Java Virtual Machine.[24] However, the need to evaluate and test other important mechanisms (such as those requiring changes to the thread-scheduling algorithm), as well as the desire to support "anytime" mobility, motivated the design and implementation of a custom virtual machine tailored for investigation of safe agent execution.

"Anytime" mobility. Until recently, each mobile-agent system has defined its own approach to agent mobility. Though new proposals such as FIPA's agent-mobility standards and OMG's Mobile Agent Facility are a step forward, some of the required elements of security cannot be implemented without foundational support in the Java language standard. The ultimate goal is to define a set of standard underlying Java security policies and mechanisms that will make agent mobility as safe as possible for both the agent and its host. The mobile agent must be able to deal with situations where it has been shipped off to the wrong address, or to a place where needed resources are not available, or to what turns out to be a hostile environment.[20] Agent hosts might become unavailable or compromised at a moment's notice, and the agent might need to immediately migrate to a safe place or "die." Also, there is the very real possibility of unauthorized inspection or tampering while the agent is traveling. Agent hosts, on the other hand, must deal with all the resource-management issues we described earlier.

To provide the most flexible and robust approach to these problems, agent mobility must be made fully transparent. This means that mobility must become an "anytime" concept, meaning that an agent can in principle (and in accordance with its unique policies) move or be moved on demand, in the middle of an arbitrary point of execution. There are many Java-based mobile agent systems currently available, such as ObjectSpace Voyager (*www.objectspace.com/products/voyager/index. html*), Concordia from Mitsubishi Electric ITA Horizon Labs (*www.meitca.com/HSL/Projects/Concordia/whatsnew.htm*), Odyssey from General Magic (*www.genmagic.com/technology/odyssey. html*), Jumping Beans from Ad Astra (*www.JumpingBeans.com*), and Aglets from IBM.[25] While all of these systems provide the ability to transport an agent from one server to another across a network connection, none of these transport mechanisms are completely transparent. The security measures provided in these systems are also not mature enough to enforce the level of fine-grained security and resource control we desire.

Anytime mobility requires that the entire state of the running agent, including its execution stack, be saved prior to a move so that it can be restored once the agent has moved to its new location. The standard term describing this process is *checkpointing*.[26] Over the last few years, the more general concept of *orthogonal persistence* has also been developed by the research community.[27] The goal

of orthogonal persistence research is to define language-independent principles and language-specific mechanisms by which persistence can be made available for all data, irrespective of type. Ideally, the approach would not require any special work by the programmer (for example, implementing serialization methods in Java or using transaction interfaces in conjunction with object databases), and there would be no distinction made between short-lived and long-lived data.

One of Java's powerful features as a programming language is that its bytecode format, which is interpreted or compiled on the fly by the Java Virtual Machine (VM) residing on the host platform, enables checkpointing in a machine-independent format. This allows the bytecode in principle to be restored on machines of differing architecture. A similar but somewhat less general approach was originally implemented in General Magic's Telescript language.[28] While it is possible to achieve some measure of transparent persistence by techniques such having a special class loader insert `read` and `write` barriers into the source code before execution, such an approach poses many problems:[29]

- the transformed bytecodes could not be reused outside of a particular persistence framework, defeating the Java platform goal of code portability;
- such an approach would not be applicable to the core classes, which cannot be loaded by this mechanism; and
- the code transformations would be exposed to debuggers, performance monitoring tools, the reflection system, and so forth, compromising the goal of complete transparency.

To make it possible to explore issues of anytime mobility and of agent security and resource control not possible to do with current Java VMs, we have developed a new VM (Aroma) from scratch.[21] The new virtual machine supports checkpointing at almost any moment. This capability lets an agent-based system implement fully transparent agent mobility in Java, a feature not available in any other commercial agent framework. This feature can be extended to provide anytime mobility, where the agent or the system can initiate the move of an agent from one host to another while retaining the complete state of the agent. The new VM also supports a finer grain of security and resource control than is available using the current Java standard implementation, allowing the system or user to dynamically monitor and control resource usage rates and permissions according to the behavior of the agent, the availability of resources, and demands by other agents. We hope that research results deriving from our implementation of the VM and its application in an agent context will help spur the adoption of required features for anytime mobility and fine-grained resource control in future versions of the standard Java platform. We are coordinating our work with related research efforts in persistent Java, such as those underway at Sun Laboratories.[29]

Secure transparent mobility involves other concerns beyond simply saving and restoring execution state. Encryption mechanisms must ensure that the mobile agent not be available for inspection or alteration en route to its new destination and that the running agent be securely transported along with the agent from the current host to the new host. The agent also must deal with issues of reliably releasing resources on the old host, acquiring them on the new one, and handling the situation gracefully if expected resources are unavailable. Moreover, a mobile agent cannot always know which classes it will need to take with it. If the agent's host is no longer available after the move, additional required classes will need to be found elsewhere, which might introduce versioning problems as well as some new security and liability issues.

WE AGREE WITH THE OBSERVATION OF KURT LEWIN, WHO SAID THAT THERE IS NOTHING QUITE SO PRACTICAL AS A GOOD THEORY.

Transparent anytime mobility is vital for situations where there are long-running or long-lived agents and, for reasons external to the agents, they need to suddenly move or be moved from one host to another. In principle, such a transparent mechanism would allow the agents to continue running without any loss of their ongoing computation and, depending on circumstances, the agents need not even be aware of the fact that they have been moved. Such an approach will be useful in building distributed systems with complex load-balancing requirements. The same mechanism could also be used to replicate agents without their explicit knowledge. This would allow the support system to replicate agents and execute them possibly on different hosts for safety, redundancy, performance, or other reasons.

A TRULY POWERFUL TOOL CAN change its user. That, in a nutshell, is what we hope we, along with the rest of the agent research community, can achieve: a change for the better in the process of agent development. As computer scientists, this research casts us in a familiar role: we create the tools that implement the underlying theory, which in turn will leverage and extend the capabilities of the domain experts who will develop the individual agents. Just as the advent of verification tools led to digital circuits that were more dependable, we expect that tools such as the ones we have described for conversation and security design will lead to agent-based systems that are more robust and reliable.

As can be seen by the tentative and general nature of some of our conclusions above, there is still much work to be done in the realm of agent theory to support the development of good tools. We agree with the observation of Kurt Lewin, who said that there is nothing quite so practical as a good theory.[30] We expect an agent-design tool to prove useful to the extent that we have serviceable theories to explain its basis of operation and delineate its scope of application.

> For example, tool-makers can exploit theory as a basis for clarifying their underlying assumptions, and also as an infrastructure upon which to build integrated collections of tools and techniques. Tool-users, on the other hand, need a robust theory to serve as the conceptual rationale for the principled application of their tools: an operator's manual alone is not sufficient.[17]

Good tools can also benefit those who are developing agent theory. By creating powerful tools that allow us to explore specific decisions about appropriate management or conversation policies in the context of a given application, we will have thereby created a mechanism with which to explore the theories themselves. ■

Acknowledgments

The work we've described is supported in part by a contract from DARPA's Control of Agent Based Systems (CoABS) Program (Contract F30602-98-C-0170); supported in part by grant R01 HS09407 from the Agency for Health Care Policy Research to the Fred Hutchinson Cancer Research Center; and by the Aviation Extranet joint-sponsored research agreement between NASA Ames, The Boeing Company, and the University of West Florida (Contract NCA2-2005). OpenProof is the result of a joint research project between the Logic Software Group at Stanford's Center for the Study of Language and Information and Indiana University's Visual Inference Laboratory. We extend our appreciation to other members of the Boeing Intelligent Agent Technology group (Bob Carpenter, Rob Cranfill, Renia Jeffers, Mike Kerstetter, Luis Poblete, and Amy Sun); to Ian Angus, Geoff Arnold, Isabelle Bichindaritz, Michael Brooks, Alberto Cañas, Kathryn Chalfan, Phil Cohen, Scott Cost, Pam Drew, Tim Finin, Ken Ford, Yuri Gawdiak, Jim Hendler, Jim Hoard, Dick Jones, Cathy Kitto, Yannis Labrou, Ken Neves, Ira Smith, Kate Stout, Keith Sullivan, and Steve Whitlock; and to faculty (Jack Woolley, David A. Umphress, and William Bricken) and members of the Seattle University software engineering program who have collaborated with Boeing on various aspects of agent development.

References

1. J.M. Bradshaw, "Everything I Know About Systems Design I Learned from My Architect: Building Systems that Work with Time Rather than against It," *Education and Smart Machines*, K. Forbus, P. Feltovich, and A.J. Cañas, eds., AAAI Press/MIT Press, Cambridge, Mass., 1999.
2. I. Foster and C. Kesselman, eds., *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, San Francisco, 1999.
3. B. Logan and J.M. Bradshaw, eds., Special issue on Software Tools for Agent Development, to be published in *Int'l J. Human-Computer Systems*, 1999.
4. J. Hendler and R. Metzger, "Putting It All Together: The DARPA Control of Agent-Based Systems (CoABS) Program," this issue.
5. J.M. Bradshaw et al., KAoS: Toward an Industrial-Strength Generic Agent Architecture, *Software Agents*, J.M. Bradshaw, ed., AAAI Press/MIT Press, 1997, pp. 375–418.
6. T. Winograd and F. Flores, *Understanding Computers and Cognition*, Ablex, Norwood, N.J., 1986
7. M. Greaves, H. Holmback, and J.M. Bradshaw, "Agent Conversation Policies," to be published in *Handbook of Agent Technology*, J.M. Bradshaw, ed., AAAI Press/The MIT Press, 1999.
8. I.A. Smith et al., "Designing Conversation Policies Using Joint Intention Theory," *Proc. Third Int'l Conf. Multi-Agent Systems (ICMAS-98)*, IEEE Computer Society Press, Los Alamitos, Calif., 1998, pp. 269–276.
9. P. Breiter and M.D. Sadek, "A Rational Agent as a Kernel of a Cooperative Dialogue System: Implementing a Logical Theory of Interaction," *Proc. ECAI-96 Workshop on Agent Theories, Architectures, and Languages*, Springer-Verlag, Berlin, 1996, pp. 261–276.
10. G. Allwein and J. Barwise, eds., *Logical Reasoning with Diagrams*, Oxford Univ. Press, New York, 1996.
11. D. Barker-Plummer and M. Greaves, "Architectures for Heterogeneous Reasoning: On Interlinguae," *Proc. First Conf. Inference in Multimedia and Multimodal Interfaces (IMMI-1)*, Edinburgh, 1994.
12. J. Barwise and J. Etchemendy, *Hyperproof*, CSLI Publications, Stanford, Calif., 1994.
13. M. Greaves, *The Philosophical Status of Diagrams*, doctoral dissertation, Stanford Univ., Stanford, Calif., 1997.
14. H.V.D. Parunak, "Visualizing Agent Conversations: Using Enhanced Dooley Graphs for Agent Design and Analysis," *Proc. ICMAS-96*, IEEE CS Press, 1996.
15. M.P. Singh, *Developing Formal Specifications to Coordinate Heterogeneous Autonomous Agents*, tech. report, Dept. of Computer Science, University of North Carolina, Raleigh, N.C., 1998.
16. P. Cohen and H. Levesque, "Communicative Actions for Artificial Agents," *Software Agents*, J.M. Bradshaw, ed., MIT Press, Cambridge, Mass., 1997, pp. 419–436.
17. J.M. Bradshaw et al., "Beyond the Repertory Grid: New Approaches to Constructivist Knowledge Acquisition Tool Development," *Knowledge Acquisition as Modeling*, K.M. Ford and J.M. Bradshaw, eds., John Wiley & Sons, New York, 1993, pp. 287–333.
18. K.M. Ford et al., "Knowledge Acquisition as a Constructive Modeling Activity," *Knowledge Acquisition as Modeling*, K.M. Ford and J.M. Bradshaw, eds., John Wiley & Sons, 1993, pp. 9–32.
19. N.E. Johnson, "Mediating Representations in Knowledge Elicitation," *Knowledge Elicitation: Principles, Techniques and Applications*, D. Diaper, ed., John Wiley & Sons, 1989.
20. K.A. Neuenhofen and M. Thompson, "Contemplations on a Secure Marketplace for Mobile Java Agents," *Proc. Autonomous Agents 98*, ACM Press, New York, 1998.
21. N. Suri, J.M. Bradshaw, and A. Wong, "Experiences with Implementing a Java Virtual Machine and API for Agent Support," tech. report, Inst. for Human and Machine Cognition, Univ. West Florida, 1999.
22. J.M. Bradshaw et al., "Extranet Applications of Software Agents," to be published in *ACM Interactions*, 1999.
23. C. Hawblitzel and T. von Eicken, *A Case for Language-Based Protection*, Tech. Report 98-1670, Dept. of Computer Science, Cornell Univ., Ithaca, New York, 1998.
24. G. Czajkowki and T. von Eicken, "JRes: A Resource Accounting Interface for Java," *Proc. 1998 ACM OOPSLA Conf.*, ACM Press, 1998.
25. D.B. Lange and M. Oshima, *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesley, Reading, Mass., 1998.
26. J.S. Plank, *An Overview of Checkpointing in Uniprocessor and Distributed Systems Focusing on Implementation and Performance*, Tech. Report UT-CS-97-372, Dept. of Computer Science, Univ. of Tennessee, Knoxville, Tenn., 1997.
27. M.P. Atkinson and R. Morrison, "Orthogonally Persistent Object Systems, *VLDB J.*, Vol. 4, No. 3, 1995, pp. 319–401.
28. J. White, "Mobile Agents," *Software Agents*, J.M. Bradshaw, ed., AAAI Press/MIT Press, 1997, pp. 437–472.
29. M. Jordan and M. Atkinson, *Orthogonal Persistence for Java—A Mid-Term Report*, Sun Microsystems Labs, 1998.
30. K. Lewin, *A Dynamic Theory of Personality*, McGraw-Hill, New York, 1935.

Jeffrey M. Bradshaw is an associate technical fellow at the Boeing Company, where he leads the Intelligent Agent Technology program. He is general chair of the Autonomous Agents '99 Conference and editor of *Software Agents* (AAAI/MIT Press, 1997) and the forthcoming *Handbook of Agent Technology* (AAAI/MIT Press). Contact him at the Boeing Co., PO Box 3707, M/S 7L-44, Seattle, WA 98124-2207; jeffrey.m.bradshaw@boeing.com; www.coginst.uwf.edu/~jbradsha/.

Mark Greaves is a member of the Natural Language Processing Group at Boeing's Applied Research and Technology Division. Contact him at mark.t.greaves@boeing.com.

Heather Holmback leads a project to develop word-sense disambiguation for language-checking applications at Boeing. Contact her at heather.holmback@pss.boeing.com.

Wayne Jansen is a computer scientist at the National Institute of Standards and Technology. Contact him at jansen@nist.gov.

Tom Karygiannis is a member of the National Institute of Standards and Technologies' Computer Security Division. Contact him at karygiannis@nist.gov.

Barry G. Silverman is a professor of systems engineering and computers and information science at the University of Pennsylvania. Contact him at www.seas.upenn.edu/~barryg/index.html.

Niranjan Suri is a research scientist at the Institute for Human and Machine Cognition at the University of West Florida. Contact him at nsuri@nuts.coginst.uwf.edu.

Alex Wong is a senior project engineer at Sun Microsystems. Contact him at alexw@toolshed.org.