Panel Summary: Agent Development Tools

Joanna Bryson[1], Keith Decker[2], Scott A. DeLoach[3], Michael Huhns[4], and Michael Wooldridge[5]

[1] Department of Computer Science
Massachusetts Institute of Technology, Cambridge, MA 00000
joanna@ai.mit.edu
[2] Department of Computer and Information Sciences
University of Delaware, Newark, DE 19716-2586
decker@cis.udel.edu
[3] Department of Electrical and Computer Engineering, Air Force Institute of Technology
2950 P Street, Wright-Patterson AFB, OH 45433-7765
sdeloach@computer.org
[4] Electrical and Computer Engineering Department
University of South Carolina, Columbia, SC 29208
huhns@ece.sc.edu
[5] Department of Computer Science
University of Liverpool Liverpool L69 7ZF, UK
m.j.wooldridge@csc.liv.ac.uk

1 Introduction

This panel (and a corresponding paper track) sought to examine the state of the art (or lack thereof) in tools for developing agents and agent systems. In this context, "tools" include complete agent programming environments, testbeds, environment simulators, component libraries, and specification tools. In the past few years, the field has gone from a situation where almost all implementations were created from scratch in general purpose programming languages, through the appearance of the first generally available public libraries (for example, the venerable Lockeed "KAPI" (KQML API) of the mid-90's [10]), to full-blown GUI-supported development environments. For example, http://www.agentbuilder.com/AgentTools/ lists 25 commercial and 40 academic projects, many of which are publicly available. The sheer number of projects brings up many questions beyond those related to the tools themselves, and we put the following to our panel members:

- What are useful metrics or simply feature classes for comparing and contrasting agent development tools or methodologies (especially features that are unique to agent systems)?
- How does a tool suggest/support/enforce a particular design methodology, theory of agency, architecture, or agent language?
- Why, as more and more development tools and methodologies become available, do most systems still seem to be developed without any specialized tools?
- What are the differences in development tools oriented toward the research community versus the agent application community, and are we already seeing a significant lag between theory and practice?

C. Castelfranchi, Y. Lespérance (Eds.): Intelligent Agents VII, LNAI 1986, pp. 331–338, 2001.

- Are there obvious development tools, that are unique to agent-based system development, that have yet to be built?
- Given the resources available to the average basic researcher, what do you think can be done to improve reuse of agent infrastructure / software?
- Is there any evidence that agent development tools actually increase programmer productivity and/or the quality of the resulting systems?
- What do you see as the largest time-sink (wasted time) in your research and in the agent software you have developed (e.g., having to write one's own compiler would be a waste of time)?
- What infrastructure have you succeeded in reusing? Why do you think you were able to do this?
- What (if any) stumbling blocks have you encountered in distributing your technology, e.g., legal, platform dependence, etc.?
- What do you see as the current infrastructure limitations in establishing large agent communities for real world applications?

2 Statement by Mike Wooldridge

Historically, the main obstacle to be overcome in developing multi-agent systems has been one of *infrastructure*. By this, I mean that in order to reach a state where two agents can communicate, (let alone cooperate, coordinate, negotiate, or whatever), there needs to be an underlying, relatively robust communications infrastructure that the agents can make use of. Moreover, simply providing such an infrastructure is not usually enough; as Gasser and colleagues suggested as long ago as 1987, there also needs to be some way of visualizing and monitoring ongoing communication in the system [3]. Until very recently, the provision of such an infrastructure represented a significant challenge for multi-agent system developers.

As an example of this, back in the late 1980s, we developed a multi-agent system platform called MADE (the Multi-Agent Development Environment) [15]. MADE allowed agents implemented using several different AI-oriented languages (LISP, PROLOG, and POP-11) to communicate using performatives such as `request` and `inform`, provided some simple tools to monitor and track the behavior of the system, and provided support for distributing agents across a local area network. MADE was implemented in C, on Sun 3/50 and Sun 3/60 UNIX workstations. Agents in MADE were UNIX processes, (which rather limited the number of agents it was possible to run at any one time), and communication between agents was handled using a mixture of UNIX SVID interprocess communication mechanisms (for communication between agents on the same physical machine), and sockets (for communication between different machines). Our experience with MADE was that handling communications, and the problems of handling a system composed of multiple processes, took up the vast majority of our time. We implemented a stream of utilities that were intended to help us launch our agents, and even such apparently trivial tasks as stopping the system were complicated by the distributed, multi-process nature of the system. In addition, the cost of computer hardware at the time meant that developing a multi-agent system was an expensive process.

In the past decade, three things have happened to change this situation fundamentally:

- The first is that computer power today is considerably cheaper than it was even a decade ago.
- The second is that, whereas in the late 1980s comparatively few computers had network connections, today it is rather rare to find computers that are *not* networked.
- The third is that high-power communications and programming environments are available much more cheaply and widely than they were previously (I am thinking here of, for example, the Java language and its associated packages, as well as communication frameworks such as CORBA and RMI).

This means that many of the infrastructure hurdles that were in place a decade ago are no longer present: any medium power PC sold today has the software and the processing capability to run a respectably sized multi-agent system. There are also many powerful, freely available software platforms for implementing sophisticated agent systems, and these can be leveraged to develop agent systems in a time scale that was unthinkable even five or six years ago.

Looking to the future, perhaps the most important trend I anticipate is that the "non-agent" part of computing (in which I include the object-oriented world) will gradually expand to encompass more and more agent features. There are several good examples of this already. Sun's Jini system is one; the ability of software components to advertise their capabilities to other components, as provided by Jini, was until recently the province of the agent community. Another good example is the reflection API provided by the Java language, which allows objects to reflect on their capabilities; again, this type of behavior, which is now provided in the Java language for free, would until recently have been regarded as an agent-like feature. In much the same way, we can expect software development platforms and tools to provide ever more "agent-like" features, thus blurring the line between what is an what is not an agent even more (as if the line needed more blurring).

3 Statement by Joanna Bryson

The questions Keith set are all interesting and important, but the one I will concentrate on is "Why aren't more systems developed with established tools?" The answers to this question have serious implications for the others.

For tool sets to be successful, *tools need to be needed.* There are two sides to this statement. The first is an issue of user education. People who don't understand a problem won't recognize its solution, and people that don't know solutions exist won't bother to purchase or download them. The other side is of course design. Tools that don't address frequently occurring problems in clear ways can't be useful.

A major problem with utility is this: *Needs change.* Our field moves very quickly. Consumer demands change, applications change, new methodological insights occur. Consequently, tools need to be easy to update and extend. Also, we should expect that throwing out old toolsets should happen regularly. This means that for tools to be useful, they have to be quick to write and quick to learn. For any ongoing project, there should be developers dedicated to the tools' continuous updating and support.

A surprisingly large problem is that *Productivity has to matter.* You wouldn't think this would be a problem, but it is. Of course, productivity matters ultimately, but it also

has to matter in the heads of decision makers. People who budget money have to think buying tools is worthwhile. This means software engineers have to tell these decision makers the value of tools. With decent tools, a developer can be three to ten times more productive, so tool budgets should be considered part of staffing budgets. How much more would you pay to have an excellent programmer rather than a pretty good one? That's how much a good tool set is worth. And don't forget the lessons of the Mythical Man-Month [7]. It's better to have a few excellent programmers than a lot of pretty good ones, because of the inefficiencies involved in teams and communication.

Unfortunately, programmers themselves don't always value learning and using tools. For some of them, this is due to past experience with tool benches, which is a serious consideration, but not the one I'm addressing right here. I'm still complaining about people not recognizing that you have to take time to save time, or spend money to make money. People who budget time have to think learning tools is worthwhile. Getting stuff done well and quickly has to matter more than early milestones, product loyalty or inertia.

Now I will get back to the issue of programmers who have experienced tool sets and don't find them worth the effort of learning. *Tool developers have to do good work.* We need to know what's already out there, and what the alternatives are. We need to understand and minimize the costs of transitioning to our tools. No one should design a tool set that hasn't tried to use at least two of the sets that are already out there, and also already tried to do the work from scratch. And *any* product developer needs to know the work habits and latent knowledge of their target users.

Being on the ATAL tools panel made me very aware that both the problems of tool quality *and* of getting people to use tools are enormous. Here's a suggestion for debugging your own toolbench: try using someone else's. Then go back and think about how many of the problems you had a novice user of your system would share. Common problems are: poor accessibility, poor documentation, no source code, no examples, and not facilitating users to help each other (I can't believe tool sets that don't have mailing lists).

But there has to be more to the underutilization of tool sets than these problems. Some products get these things right, and they still struggle. In mainstream software engineering, I would swear by Center Line C/C++ (formerly Saber), PARC Place Smalltalk VisualWorks80, NextStep (ObjectiveC — now possibly resurrected in Mac X), and to a lesser extent XANALYS (formerly Harlequin) LispWorks. These are all excellent programming tool sets, and they are all struggling or bankrupt. Why??? Is it because the Microsoft antitrust action came too late? In an incredible coincidence—do all these companies lack basic business skills? Or is there something fundamental lacking in corporate culture that can't recognize, value or use good tool sets? I don't know the answer, but I know it matters. I wouldn't expect *any* agent toolkit to succeed if a product like Center Line can't. A lot more people use C/C++ than use agents, and Center Line is probably the best software product I've ever used.

To end on a positive note, I'd like to suggest that the best thing we can do to address these issues is to set up an agent toolset comparison server. This way, for those of us who have used more than one agent toolset (at least 20% of the people who attended ATAL said they have), if we stumble on a good one, we can let people know. New

users would then have a better chance of having a good first experience, if they could make an informed choice. Comments on the server might also provide information for toolmakers: they could see what users valued or disparaged in various toolkits. As a community, we should cooperate to both police and promote all of our agent toolkits. This is the best way to make sure we make targeted, useful, easy-to-adopt tools, and that these tools get to their intended consumers.

4 Statement by Scott DeLoach

Multi-agent systems development methodologies and tools to support them are still in their infancy. While there have been several methods proposed for analyzing, designing, and building multi-agent systems, most of these have come out of the academic community [1,2,5,6,8,9,13] and there has not been wide-spread industry support for any particular approach. Because agents provide a unique perspective on distributed, intelligent systems, our traditional ways of thinking about and designing software do not fit the multi-agent paradigm. These unique characteristics cry out for methodologies, techniques, and tools to support multi-agent systems development. Providing this capability requires a three part solution: an appropriate agent modeling language, a set of agent-unique methodologies for analyzing and designing multi-agent systems, and a set of automated tools to support modeling language and methodologies.

Developing an agent modeling language that can capture the wide range of concepts associated with multi-agent systems is a challenge at best. One place to start is with the Unified Modeling Language (UML), which is rapidly becoming the standard in the analysis and design of object-oriented systems [12]. There are many similarities including the encapsulation of state and services within a single entity and the passing of messages between these entities. While much of the syntax of UML can be adapted to multi-agent systems, the semantics would be inherently different. The concept of an object class can be equated to an agent class; however, there are many differences. First of all, classes are not decomposable. They have attributes and methods, but no internal architecture that further describes them. Agents, on the other hand, can, and often are, decomposed into more detailed, often complex, architectures. Also, in object-orientation, messages almost always become method calls between objects whereas in multi-agent systems there usually are messages that are transmitted between distributed systems and have all the associated problems. Therefore, multi-agent systems must include the ability to define message passing protocols that include such things as error handling and timeout conditions.

A concept associated with many of the proposed agent methodologies is the use of roles and role models. These concepts are lacking in the object-oriented paradigm and require a special modeling component. An issue related to the modeling language is that of how to evolve the models of multi-agent systems from the requirements specification into a detailed design. This is the area of agent methodologies. Modeling of agent capabilities is a start; however, a methodology provides a map that shows the multi-agent system developer how to move from a set of requirements to design to implementation. Most of the current methodologies focus either on high-level multi-agent aspects, such as the organization and coordination of the agents, or on low-level issues such as how to

develop internal agent reasoning. The key to modeling the internal reasoning of agents in a multi-agent system is that it must be consistent with the external interface defined at the organizational level. For instance, if the external interface of an agent states that the agent will respond to a particular message type with either message type A or B, the internal design of the agent must adhere to this as well. This can be much more complex than it first appears. Since an agent, by popular definition, is autonomous, it must be able reason about its actions. The problem comes in guaranteeing that the reasoning will adhere to the external interface definition. Learning brings with it even more difficulties in this area.

The last area necessary for advancement of multi-agent methodologies and techniques is the tools themselves. Many toolsets have been developed to support the development of individual agents as well as multi-agent systems [1,11]. However, most of these tools are either implementation toolkits or limited tools that support only a particular agent architecture working within a specific agent environment or framework. What is needed are tools that support the developer in using an appropriate modeling language and methodology. Generally, tools that support particular methodologies are more useful than simple drawing tools. Good methodologies define relationships between modeling components that can be enforced by a toolset. The methodology should not only describe the order in which the models should be developed, but should define how to derive information from one model to the next. While the existence of tools would greatly increase the ability of developers to design and build multi-agent systems, the development of the languages and methodologies must precede the development of the tools. Once a language is defined for expressing multi-agent designs, development of rudimentary, drawing level, toolsets can be undertaken. However, it is not until complete multi-agent system development methodologies are defined that the power of such toolsets can be realized. Toolsets can then go far beyond the current crop of tools to actually help the designer by making suggestions and actually performing many of the mundane steps while allowing the designer to concentrate on the more critical analysis and design decisions.

5 Conclusion by Keith Decker

Our panelists attacked different areas of the questions surrounding agent development tools—the "agentization" of traditional software; quality, support and distribution issues; the need for tools to complement a development methodology. At Delaware, our own experiences with building the DECAF toolkit bears out many of these observations[4]. While the earliest publicly available agent implementation toolkits focussed mostly on providing increasingly well-thought-out APIs for agent communications, in order to actually build agents, programmers needed to piece together those APIs to create some kind of complete agent architecture from scratch. While this made supporting different research goals easy, it also made it harder for students, multi-agent application programmers, or researchers interested in only some agent architectural components to develop their ideas quickly and efficiently. From the standpoint of non-researchers or those new to the field, the focus of an agent toolkit might just as well be on programming agents and building multi-agent systems, and not on designing new internal agent architectures

from scratch for each project.[1] Another important goal for an agent toolkit is to support that which makes agents different from arbitrary software objects: flexible (reactive, proactive, and social) behavior [14]. The other goals of toolkits may be to develop a modular platform suitable for research activities, allow for rapid development of third-party domain agents, and provide a means to quickly develop complete multi-agent solutions using combinations of domain-specific agents and standard middle-agents. All in all, the toolkit should both take advantage of the features of the underlying programming language and provide an efficient framework that adds value for the developer.

References

1. Scott A. DeLoach and Mark Wood. Developing multiagent systems with agentTool. In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents VII. Agent Theories, Architectures, and Languages — 7th. International Workshop, ATAL-2000, Boston, MA, USA, July 7–9, 2000, Proceedings*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2001. In this volume.
2. A. Drogoul and A. Collinot. Applying an agent oriented methodology to the design of artificial organizations: A case study in robotic soccer. *Autonomous Agents and Multi-Agent Systems*, 1(1):113–129, 1998.
3. L. Gasser, C. Braganza, and N. Hermann. MACE: A flexible testbed for distributed AI research. In M. Huhns, editor, *Distributed Artificial Intelligence*, pages 119–152. Pitman/Morgan Kaufmann, 1987.
4. J. Graham and K.S. Decker. Towards a distributed, environment-centered agent framework. In N.R. Jennings and Y. Lespérance, editors, *Intelligent Agents VI*, LNAI-1757, pages 290–304. Springer Verlag, 2000.
5. C. Iglesias, M. Garijo, and J. González. A survey of agent-oriented methodologies. In J. P. Muller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V. Agents Theories, Architectures, and Languages*. Springer-Verlag, 1998. Lecture Notes in Computer Science, vol. 1555.
6. C. Iglesias, M. Garijo, J. González, and J. Velasco. Analysis and design of multiagent systems using MAS-CommonKADS. In *INTELLIGENT AGENTS IV: Agent Theories, Architectures, and Languages*. Springer Verlag, 1998.
7. Frederick P. Brooks Jr. *The Mythical Man-month: Essays on Software Engineering*. Addison-Wesley Publishing Company, Reading, MA, 20th anniversary edition edition, 1995.
8. E. Kendall, U. Palanivelan, and S. Kalikivayi. Capturing and structuring goals: Analysis patterns. In *Proceedings of the Third European Conference on Pattern Languages of Programming and Computing*, July 1998.
9. D. Kinny, M. Georgeff, and A. Rao. A methodology and modelling technique for systems of BDI agents. In *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW '96*. Springer-Verlag, 1996. Lecture Notes in Artificial Intelligence, vol. 1038.
10. D. Kuokka and L. Harada. On using KQML for matchmaking. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 239–245, San Francisco, June 1995. AAAI Press.
11. H. Nwana, D. Ndumu, L. Lee, and J. Collis J. ZEUS: A toolkit for building distributed multi-agent systems. *Applied Artificial Intelligence Journal*, 13(1):129–185, 1999.

[1] Designing new agent architectures is certainly an important *research* goal, but fraught with peril for beginning students and programmers wanting to work with the agent concept.

12. J. Odell, H. V. D. Parunak, and B. Bauer. Representing agent interaction protocols in UML. In *Proceedings of the First International Workshop on Agent-Oriented Engineering*, June 2000.
13. M. Wooldridge, N. Jennings, and D. Kinny. The gaia methodology for agent-oriented analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems*, 3(3), 2000.
14. M. Wooldridge and N.R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.
15. M. Wooldridge, G. M. P. O'Hare, and R. Elks. FELINE — a case study in the design and implementation of a co-operating expert system. In *Proceedings of the Eleventh European Conference on Expert Systems and Their Applications*, Avignon, France, May 1991.