ADE: Filling a Gap Between Single and Multiple Agent Systems

James Kramer and Matthias Scheutz
Artificial Intelligence and Robotics Laboratory
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556, USA
{jkramer3,mscheutz}@cse.nd.edu

Abstract

Agent systems tend to focus either on individual agent architectures or the infrastructure required for multi-agent interaction and distributed computing. Defining a feature set for each type of system highlights a "gap" between them. A system in that gap exhibits additional features not found in either type alone. We argue that these "gap features" are of crucial importance for complex, intelligent, agent-based applications. In addition to specifying the SAS, MAS, and "gap" feature sets, five agent systems are compared accordingly. We then give a description of an implemented robotic application that demonstrates the utility of all the identified "gap features".

1 Introduction

"Agent systems", which we distinguish here as *single-agent* and *multi-agent* systems (SAS and MAS, respectively), for developing complex agent-based applications generally exhibit disparate features. SAS tend to focus on the organization of functional components in an agent architecture. MAS, on the other hand, focus on providing the computing infrastructure–a "social environment" in which multiple agents can interact.

The SAS/MAS distinction is substantial enough to highlight an important niche in the system space that we call the *SAS/MAS gap*. A system in this gap would provide *computing infrastructure plus intelligence*, i.e., support for individual agent architecture development as well as the infrastructure necessary for agent-to-agent interaction. We believe that a *synthesis* of both kinds of system is necessary for the development of complex, intelligent, autonomous applications. For example, a task involving a joint human-robot team might require a wide range of AI techniques, whose computational demands require the distribution of a complex agent architecture over multiple hosts. Furthermore, security (e.g., data privacy and system access), run-time system modification, and reliability (i.e., failure detection and recovery) features may also be required. A system that synthesizes SAS and MAS provides these advanced features, which are not possible in either type of system alone.

2 The SAS/MAS Distinction

An "agent system" facilitates the task of defining and implementing an agent-based application. Differences between SAS and MAS are a recognized research issue; for instance, a similar distinction is made in [Jennings *et al.*, 1998], while in [Zambonelli and Omicini, 2004] it is characterized in terms of *scales of observation* (*micro*, *macro*, and *meso*). However, a novel perspective on the area between them can be gained by first establishing the distinct features exhibited by SAS and MAS, then examining the results of their combination. We note that any list of features is open-ended and necessarily incomplete; we include only those that pertain to the "gap features" presented in Section 3[1].

2.1 Single Agent Systems (SAS)

We identify five SAS features in total. The first, *device abstractions* (**S1**), refers to the interfaces of modules available in an agent system [Vaughan *et al.*, 2003], which can be physical sensors and/or effectors, such as cameras, sonar, or laser hardware, or virtual processing units such as speech or database APIs. Feature **S2**, *software integration*, refers to tools that facilitate the integration of external software, either at the component level (e.g., a localization routine) or a complete application-as-component (e.g., speech production), greatly enhancing development time and effort. *Predefined components* (**S3**) are analogous to software libraries; their inclusion in an agent system allows rapid agent development, saving the designer both time and effort. *Hierarchical structures* (**S4**) refer to the common software engineering technique that promotes organization, functional encapsulation, and component reuse. Finally, feature **S5** concerns *maintaining component state*, which can be used for debugging, repetition of application execution, gathering performance statistics, or detecting improper component operation.

[1] An expanded version of this paper will include more features.

Type	Feature	*Soar*	*Player/Stage*	*Swarm*	*RETSINA*	*ADE*
S A S	**S1: "Device" abstractions**		√			√
	S2: Software integration		√			√
	S3: Predefined components		√			√
	S4: Hierarchical structures	√	√	√		√
	S5: Maintaining component state	√	√	√		√
M A S	**M1: Multiple, Concurrent Services**		√		√	√
	M2: Agent naming service		√		√	√
	M3: Agent management	√		√	√	√
	M4: Authentication and access control				√	√
	M5: Maintaining system state			√	√	√

Table 1: Agent System Feature Satisfaction

2.2 Multi-agent Systems (MAS)

We identify five MAS features in total. The first, *multiple, concurrent services* (**M1**), refers to the ability of an agent to provide its services to multiple other agents simultaneously. Feature **M2**, an *agent naming service*, provides location transparency, mapping an agent's name to a network location to enhance distribution of an application over many computational hosts. *Agent management*, feature **M3**, refers to the ability to externally control an agent's operation, including starting, stopping, suspending or adjusting an agent's parameters. Agent *authentication and access control* (**M4**) are security concerns that ensure an agent has proper credentials to use the services offered by another agent. Finally, feature **M5**, *maintaining system state*, refers to keeping a model of the structure of all parts of the system, whether it is explicitly specified or derived by some means.

2.3 Selected Agent Systems

To make the character of a system exhibiting these features more concrete, an overview of five selected agent systems follows, chosen as examples of the broad range of systems used for developing agent applications. We describe the SAS and MAS features found and lacking in each system, summarized in Table 1.

SOAR [Laird *et al.*, 1987] is an example of a unified cognitive architecture. Its purpose is to provide aspects of a cognitive agent which are constant over time; that is, its structure must be applicable in a variety of domains and not dependent on the knowledge encoded therein. At its heart, Soar is a *production-rule system* that relies on a *rete-network* [Forgy, 1982] for actual implementation.

Soar incorporates *hierarchical structures* and *maintains component state* (**S4** and **S5**), through the use of its knowledge base and its decision procedures. However, being a cognitive architecture (that is, mostly concerned with internal reasoning and problem solving), Soar does not provide *"device" abstractions* or *predefined components* (**S1** and **S3**). While there are mechanisms for external input and output, there is little to no support for *software integration* (**S2**).

The only MAS feature found in Soar is *agent management*. Other MAS features are most likely lacking due to the focus on individual agents operating on a single host. For instance, when developing a single agent, neither an *agent naming service* nor *authentication and access control* mechanisms (**M2** and **M4**) are necessary. Similarly, the ability of an agent to provide *multiple, concurrent services* (**M1**) assumes multiple processes exist to use those services. Finally, there is no notion of *maintaining system state* (**M5**) outside of individual agents.

Player/Stage [Gerkey *et al.*, 2003] project is for developing robotic applications. *Player* is a server interface for application implementation, while *Stage* provides simulation facilities. Rather than treat a robot as the primary unit of agency, it focuses on *devices* [Vaughan *et al.*, 2003], or functional components of a robotic architecture.

Player/Stage, being concerned with the physical necessities of working with robots, provides strong support for *"device" abstractions*, *software integration*, and *predefined components* (**S1**, **S2**, and **S3**). Since the primary unit of agency is a device (i.e., a component), it also provides mechanisms for *maintaining component state* (**S5**). Furthermore, the inclusion of a *passthrough device*, which allows agent designers to consolidate devices in a single interface, satisfies feature **S4** (*hierarchical structures*).

Devices are designed such that they are multithreaded (satisfying **M1**, *multiple, concurrent services*) and a rudimentary *agent naming service* (**M2**) supports distribution of devices across hosts. However, the designers of Player/Stage explicitly avoid adding infrastructure to keep the system "free from the computational and programmatic overhead that is generally associated with [it]" [Gerkey *et al.*, 2003]. Thus, little in the way of *agent management*, *authentication and access control*, and *maintaining system state* (**M3**, **M4**, and **M5**) are supported.

Swarm [Minar *et al.*, 1996; Swarm, 2005] is an agent based modeling (ABM) system used for artificial life and complex systems simulations. ABM relies on defining many simplified individuals by specifying a set of *characteristics*, placing the agents in some *context* (or *environment*), specifying the relations and interactions of the individuals, setting the initial condi-

tions, and then running the simulation.

With its focus on complex systems, Swarm provides support for both *hierarchical structures* and extensive *maintainence of component state* (**S4** and **S5**). However, since agents are simplified models of individuals that interact within the specified environment, no support is given for *"device" abstractions*, *software integration*, nor *predefined components* (**S1**, **S2**, and **S3**).

Similarly, the focus on the study of complex systems indicates support for both *agent management* and *maintaining system state* (**M3** and **M5**). But because agents are simplified models of interacting individuals, there is no need for agents to provide *multiple, concurrent services* (**M1**). Swarm does not provide any infrastructure for distributed computing, so features **M2** and **M4** (an *agent naming service* and *authentication and access control*) are not supported.

RETSINA [Sycara *et al.*, 2003] is a framework that facilitates complex agent interactions by establishing an open environment in which heterogeneous agents can participate. A variety of infrastructure components are linked to form the agents' "environment".

Because RETSINA is only a framework that provides an environment in which agents interact, it does not support any SAS features. Rather, it is agnostic towards individual agents, which are assumed to be implemented independently of the system.

Conversely, exactly because it is a framework for the interaction of multiple agents, RETSINA provides all MAS features. Assuming the individual agent supports it, *multiple, concurrent services* (**M1**) is enabled, made accessible across hosts by the inclusion of an *agent naming service* (**M2**). In addition to simply *maintaining system state* (**M5**), RETSINA also allows *agent management* (**M3**) for high-level control of system operation. Another aspect of system control is robust *authentication and access control* (**M4**) mechanisms (discussed in [Singh and Sycara, 2004]).

`ADE` [Andronache and Scheutz, 2006; Scheutz, 2006] is a programming environment for distributed agent architectures designed with robotic applications in mind as an end-to-end development environment that provides a comprehensive set of design tools integrated with an infrastructure in which to use them. A guiding principle of `ADE` is that functional components of an agent architecture can assume the characteristics of independent agents, while preserving the notion that agents themselves may have internally complex architectures. An "agent" in `ADE` is called an `ADEServer`, which may be a simple component-as-agent or include a mix of components and other `ADEServer`s to form a *complex* `ADEServer`.

Due to its development for robotic applications, `ADE` provides a set of *"device" abstractions*, *predefined components*, and wrappers for *software integration* (**S1**, **S2**, and **S3**). *Hierarchical structures*, feature **S4**, are supported through the ability to treat architectural components as individual agents that can also have complex internal architectures. Finally, *maintaining component state* (**S5**) is supported through logging mechanisms and the use of a *heartbeat* signal that periodically updates a component's status.

Unless specifically restricted, an `ADE` component provides *multiple, concurrent services* (**M1**) to other components. Restriction is controlled by an `ADE` infrastructure component called the `ADERegistry`, which is an enhanced *agent naming service* (**M2**) that organizes, tracks, and controls access to `ADEServer`s. An `ADERegistry` implements *authentication and access control* (**M4**) at the system level by requiring approval for `ADEServer` registration, when an individual `ADEServer` can specify finer grained access information. Since every `ADEServer` is accessible through an `ADERegistry`, both *agent management* and *maintainence of system state* (**M3** and **M5**) are possible.

3 The SAS/MAS Gap

3.1 Gap Features

An agent system with both SAS and MAS features provides an expanded set of design options over either type of system alone. In addition to supporting the individual SAS and MAS features, other features result from their synthesis. We identify five of these "gap features", which we believe are crucial for complex agent-based applications.

G1: *Distributed Architectures* Inclusion of an *agent naming service* (**M2**) facilitates the distribution of agents across many hosts, making knowledge of an agent's location unimportant. Providing the means to decouple a component's location through assignment of a name allows any part of an individual agent architecture to be located on any connected host. In addition, the use of *device abstractions*, *software integration*, and *predefined components* (**S1**, **S2**, and **S3**) can be used to transform architectural components into agents that are assigned names.

G2: *Shared, Location Independent Components* The combination of *device abstractions* with the ability to *service multiple agents* (**S1** and **M1**) allows architectural components to be shared. A further enhancement is made with the inclusion of an *agent naming service* (**M2**) to facilitate distribution. Additionally, using *software integration* mechanisms and *maintaining component state* (**S2** and **S5**) can be used to essentially create a "staging area" in the interface to external software, explicitly providing context switching for multiple use.

G3: *Run-time Component Substitution* The ability to control execution of individual agents in a MAS (*agent management*, **M3**) is due to the fact that agents in a MAS operate independently of one another. SAS components, on the other hand, are often tightly integrated and rely on synchronous operation; halting one typically leads to a total breakdown in system function. Applying agent management mechanisms to components can avoid this issue. Furthermore, given the ability to *maintain component state*

(**S5**), where the state is summarized by a *device abstraction* (**S1**), a component that can be managed like an agent can save its state, be halted, and replace by another component that uses the same abstraction.

G4: *Secure Subsystem Separation* A benefit of using a *hierarchical structure* (**S4**) is to accommodate the breakdown of a system into subsystems that can be more easily understood. Decomposition of a system into subsystems often takes advantage of *device abstractions*, *predefined components*, or *software integration* (**S1**, **S2**, and **S3**) for a clear specification of the architecture. A MAS that incorporates security mechanisms such as *authentication and access control* (**M4**) provides coarse-grained access control at the agent level. The combination of these features restricts access to a given subset of components, yielding *secure subsystem separation*. Furthermore, a system that provides transparently *distributed architectures* (**G1**), which includes an *agent naming service* (**M2**), is able to extend this across multiple hosts.

G5: *Failure Handling* Detecting failures of a component's operation depends on being able to assess the *component state* (**S5**). When using *hierarchical structures* (**S4**), failures may affect components indirectly, as a component may rely on information from a component to which it is not directly connected. Combined with a model of the system, that is, the *system state* (**M5**), detection of failures can propagate throughout the entire architecture, giving each component an "awareness" of far-reaching areas of the system and potentially adjusting their own operation. Robust operation would further be enhanced by mechanisms that support *failure recovery*. Incorporating *agent management* (**M3**) facilities can be used to make attempts to recover from detected failures. Furthermore, the *maintenance of component state* can provide the means to restart a component in the state existing when failure occurred.

3.2 Utility of a "Gap" System

We feel that an agent system that displays all the SAS, MAS and gap features identified is crucial for the development of future intelligent agent applications. In general, such applications will require an infrastructure that supports highly controlled access to different kinds of information stored within a distributed system. Moreover, refined internal monitoring and supervision tools are needed to detect failures of components, initiate recovery, and ensure the long-term, autonomous operation of the application.

At a large enough scale, an application will require distribution (**G1**), due to the high computational demands of the utilized AI technology. Sharing components (**G2**), particularly when done transparently (that is, without concern for component location), is one way to mitigate the demand for resources. During the application's execution lifetime, it may be necessary to upgrade arbitrary components, made possible with run-time component substitution (**G3**). A common concern in distributed systems is the ability to keep potentially sensitive or proprietary data secure; maintaining a separation of subsystems that make use of security mechanisms (**G4**) provides one method for promoting system security. Finally, a system that provides failure detection and recovery mechanisms (**G5**) allows increased application up-time and graceful system degradation.

At the same time, the benefit of these features is not guaranteed. For instance, the network latency involved in distributing a highly reactive control system may harm the application's real-time performance. Nonetheless, each of features **G1-5** enable capabilities that have to be addressed in certain cases; we feel that such cases are far more prevalent than those where they will be detrimental.

3.3 Agent System Comparison

While a system's evaluation is always dependent on the choice of relevant features, no consensus has been reached regarding definitive analysis criteria. As Ricordel [Ricordel and Demazeau, 2000] points out, "any criteria is relevant to a specific outside need, and a platform can only be compared relatively to another one". However, an objective comparison of agent systems can be made by constructing a two-dimensional "agent system space" where dimensions consist of SAS vs. MAS features. Agent systems can then be compared by placing them in that space.

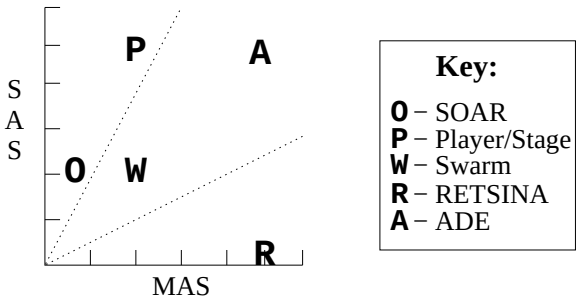

Figure 1: Agent System Placement in SAS/MAS Space

Figure 1 illustrates the agent-system space that results from putting the SAS and MAS features, found in Sections 2.1 and 2.2, on the X- and Y-axes, respectively. The "gap" between the systems is indicated by the area between the dotted lines. The agent systems from Section 2.3 are placed in the agent space according to their "score", which is the sum of features along each dimension that they exhibit (as shown in Table 1). An agent system that occupies a position in the upper-right corner indicates that it exhibits many or all of the SAS and MAS features, and consequently also the "gap" features.

Only Swarm, Player/Stage, and `ADE` have the SAS and MAS features necessary to exhibit any of the "gap" features. Since Swarm is used to study the large scale interactions of generally simple and highly inde-

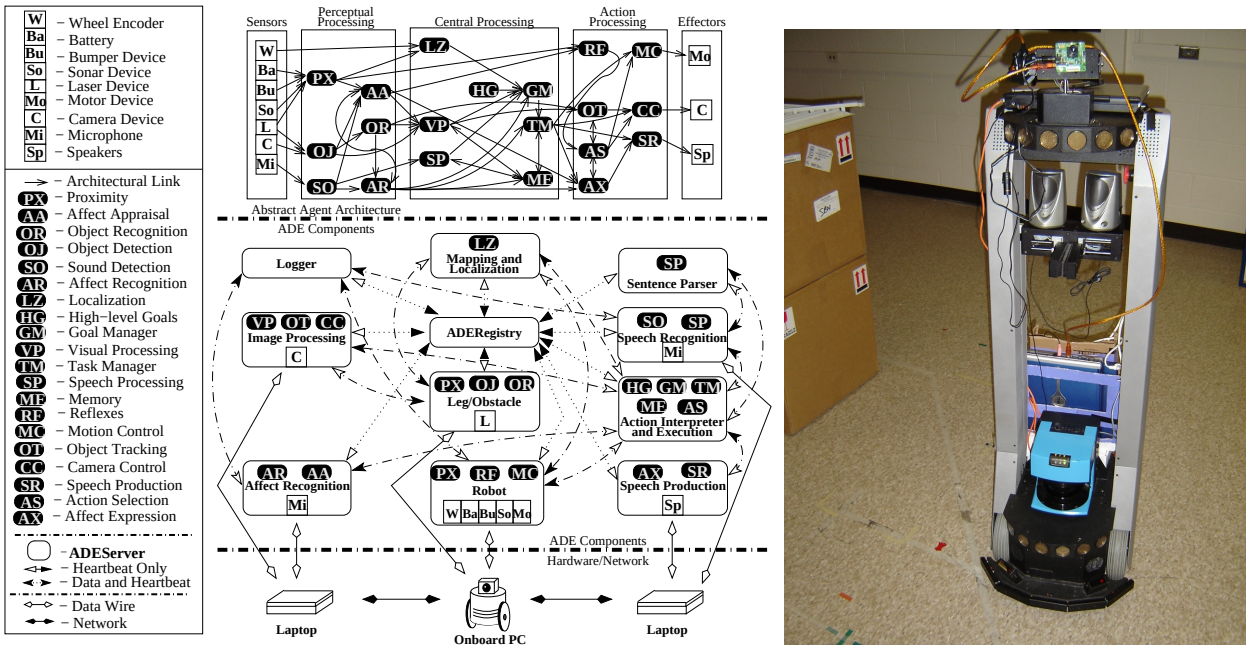

Figure 2: **Left:** The DIARC Architecture. **Right:** The robot performing a task.

pendent agents whose individual failure is not generally a concern, its support of feature **G5** (*failure handling*) is almost incidental to system operation. Player/Stage satisfies both features **G1** and **G2** (*distributed architectures* and *shared, location independent components*), but the designers explicitly shy away from providing the infrastructure necessary for the other gap features (see Section 2.3). For certain applications, this is entirely acceptable; however, a large class of agent applications require more. Only `ADE` supports *all* of the specified "gap" features.

4 A "Gap" Application

We give a description of an implemented application that uses all the gap features listed in Section 3. The DIARC architecture is being used to examine the role of affect in a social robot that interacts with humans using natural language in joint human-robot tasks [Scheutz *et al.*, 2006] and has been demonstrated in a robotic competition [Scheutz *et al.*, 2004]. The left side of Figure 2 shows a "3-level" depiction of the architecture. The bottom, or "Hardware Level", specifies hosts and connections. The middle level, referred to as the "`ADE` Component Level", uses rounded rectangles to show the `ADEServer`s in the application. Two types of lines signify communication connections: a dotted line indicates a client/server connection over which a "heartbeat" is sent, where the solid arrowhead indicates the originating `ADEServer` and the empty arrowhead indicates the component receiving the `ADEClient` (as discussed in Section 2.3), wheras a dashed/dotted line is used to represent a connection over which both a heartbeat signal and other data is transferred. Hardware devices used by an `ADEServer` are depicted by a set of labelled squares within a rectangle. The top level is referred to as the "Abstract Agent Architecture Level", where darkened ovals represent architectural components that are shown in a data flow progression from sensory input on the left to effector output on the right, grouped into five high-level categories. Two relations between the bottom and middle levels are shown: (1) `ADEServer`s are placed in a vertical column directly above the host on which they execute and (2) connections between hardware devices and the `ADEServer`s that use them are indicated by solid lines that cross the separating line. The relation between the middle and top levels consists of including a darkened oval that represents a functional architectural component of an agent within an `ADEServer`'s rectangle.

This example has been implemented using an ActivMedia Peoplebot (shown on the right of Figure 2) with a pan-tilt-zoom camera, a SICK laser range finder, three sonar rings, and an on-board 850 MHz Pentium III. In addition, it is equipped with two PC laptops with 1.3 GHz and 2.0 GHz Pentium M processors, each with a microphone, and one with two external speakers. All three run Linux with a 2.6.x kernel and are connected via an internal wired ethernet; a single wireless interface on the robot enables system access from outside the robot for the purpose of starting and stopping operation. Obstacle detection and avoidance is performed on the on-board computer, while speech recognition and production, action selection, and subject affect recognition are performed on the laptops.

Using three hosts for application execution demonstrates feature **G1**, *distributed architectures*. "Logger Server" execution is an example of feature **G2**, *shared, location independent components*, in that the "Affect Recognition", "Robot", and "Speech Recognition" components all make use of logging; the logger was, at

various times, located on different hosts. Feature **G4**, *secure subsystem separation*, was demonstrated by the "Sentence Parser" server, which only allowed access to the "Action Interpreter and Execution" server. Finally, features **G3** and **G5**, *run-time component substitution* and *failure handling*, were demonstrated throughout application testing and execution.

With the entire architecture executing, various components were purposely interrupted to confirm server inter-dependencies and test the robustness of the system, for which we give three examples. First, the "Action Interpreter and Execution" server was forcibly taken down, at which point a chat-bot program embedded in the "Speech Recognition" server handled conversational duties until a new connection was made. Upon not receiving a heartbeat from the "Action Interpreter and Execution" server, the `ADERegistry` restarted it; once recovered, the new "Action Interpreter and Execution" server re-established connections with the other servers and system operation continued. Second, the "Image Processing" server was stopped. The "Action Interpreter and Execution" server relayed this to the "Robot" server, which does not have a direct connection to the "Image Processing" server, causing the motors to be shut off for safety reasons. Additionally, speech was generated by the "Action Interpreter and Execution" server to alert the user. Finally, the `ADERegistry` was forcefully interrupted and manually restarted. Upon restart, `ADEServer`s re-registered for full restoration of system functionality. In fact, at least once the batteries powering the robot base failed, causing the onboard computer to shut down; when brought back up, the components executing on the laptops automatically reconnected and the system continued operation, demonstrating the robustness of the system.

5 Conclusion

Specification of SAS and MAS features allows comparison of agent systems and highlights a niche in the resultant "agent system space". An agent system in this "gap" provides features that are only possible when all the listed SAS and MAS features are also supported; of the five agent systems in Section 2.3, only `ADE` "fills the gap". While not every agent-based application requires all these features, they are *crucial* for a certain class of agent-based applications, i.e., those that rely on a variety of computationally expensive AI technologies, operate autonomously in a reliable way, and provide secure, fine-grained access control, such as the one presented in Section 4.

References

[Andronache and Scheutz, 2006] V. Andronache and M. Scheutz. ADE - a tool for the development of distributed architectures for virtual and robotic agents. In P. Petta and J. Müller, editors, *Best of AT2AI-4*, volume 20, 2006.

[Forgy, 1982] C. Forgy. RETE: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[Gerkey *et al.*, 2003] B. Gerkey, R. Vaughan, and A. Howard. The Player/Stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th International Conference on Advanced Robotics*, pages 317–323, Coimbra, Portugal, June 2003.

[Jennings *et al.*, 1998] N. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[Laird *et al.*, 1987] J. Laird, A. Newell, and P. Rosenbloom. SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[Minar *et al.*, 1996] N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The swarm simulation system: A toolkit for building multi-agent simulations. Technical Report Working Paper 96-06-042, Santa Fe Institute, 1996.

[Ricordel and Demazeau, 2000] P. Ricordel and Y. Demazeau. From analysis to deployment: A multi-agent platform survey. In *Engineering Societies in the Agents World*, volume 1972 of *LNAI*, pages 93–105. Springer-Verlag, Dec 2000.

[Scheutz *et al.*, 2004] M. Scheutz, V. Andronache, J. Kramer, P. Snowberger, and E. Albert. Rudy: A robotic waiter with personality. In *Proceedings of AAAI Robot Workshop*. AAAI Press, 2004.

[Scheutz *et al.*, 2006] M. Scheutz, P. Schermerhorn, and J. Kramer. The utility of affect expression in natural language interactions in joint human-robot tasks. ACM Conference on Human-Robot Interaction (HRI2006), forthcoming, 2006.

[Scheutz, 2006] M. Scheutz. ADE - steps towards a distributed development and runtime environment for complex robotic agent architectures. Applied Artificial Intelligence, forthcoming, 2006.

[Singh and Sycara, 2004] R. Singh and K. Sycara. Securing multi agent societies. Technical Report CMU-RI-TR-04-02, Carnegie Mellon Robotics Institute, 2004.

[Swarm, 2005] The swarm multi-agent simulation system. http://www.swarm.org/wiki/Main_Page, 2005.

[Sycara *et al.*, 2003] K. Sycara, M. Paolucci, M. Van Velsen, and J. Giampapa. The RETSINA MAS infrastructure. *Autonomous Agents and Multi-Agent Systems*, 7(1):29–48, 2003.

[Vaughan *et al.*, 2003] R. Vaughan, B. Gerkey, and A. Howard. On device abstractions for portable, resuable robot code. In *Proceedings of IROS 2003*, pages 2121–2427, 2003.

[Zambonelli and Omicini, 2004] F. Zambonelli and A. Omicini. Challenges and research directions in agent-oriented software engineering. *Autonomous Agents and Multi-Agent Systems*, 9(3):253–283, 2004.