ADE - An Architecture Development Environment for Virtual and Robotic Agents

Virgil Andronache Matthias Scheutz
Artificial Intelligence and Robotics Laboratory
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556, USA
e-mail: {vandrona,mscheutz}@cse.nd.edu

Abstract

In this paper we present the agent architecture development environment ADE, intended for the design, implementation, and testing of distributed agent architectures. After a short review of architecture development tools, we discuss ADE's unique features that place it in the intersection of multi-agent systems and development kits for single agent architectures. A detailed discussion of the general properties of ADE, its implementation philosophy, and its user interface is followed by examples from virtual and robotic domains that illustrate how ADE can be used for designing, implementing, testing, and running agent architectures.

1 Introduction

In recent years, several agent toolkits and frameworks have been proposed that are intended to support either the design of multi-agent systems (e.g., JADE [Bellifemine *et al.*2000], RETSINA [Sycara and others2003], AGENTBASE [AgentBase], ZEUS [Nwana *et al.*1997]), or the design of agent architectures for single agents [Sloman1999, Konolige2002]. Currently, there are no systems available that combine and integrate these two realms. To bridge the gap between multi-agent system frameworks and agent architecture toolkits for single virtual and robotic agents, we propose the agent architecture development environment ADE, which provides a homogeneous, user-friendly environment for the development of architectures for virtual and robotic agents in single and multi-agent settings.

In the following, we first give a brief overview of the *agent architecture framework underlying* ADE, which is at the heart of ADE's flexibility. We follow with a description of the user interface and the supporting environment. An example of agent design for a robotic agent, in the context of ADE then illustrates the use of the toolkit. We conclude with by placing ADE in the context of other single-agent and multi-agent development tools and frameworks and summarizing current and planned ADE features.

2 ADE

We start with a brief overview of the theoretical foundation ADE, the APOC architecture framework.

2.1 The APOC Architecture Framework

ADE stands for "APOC Development Environment", where APOC is a general, universal agent architecture framework [Scheutz and Andronache2003a, Scheutz and Andronache2003b], in which any agent architecture can be expressed and defined.

APOC is an acronym for "Activating-Processing-Observing-Components", which summarizes the functionality on which the ADE agent architecture toolkit is built: heterogeneous computational units called "components" which can be connected via four link types to define an agent architecture. [1]

The four link types defined in APOC are intended to cover important interaction types among components in an agent architecture: the "activation link" (A-link) allows components to send messages to and receive messages from other components; the "observation link" (O-link) allows components to observe the state of other components; the "process control link" (P-link) enables components to influence the computation taking place in other components, and finally the "component link" (C-link) allows a component to instantiate other components and connect to them via A-, P-, and O-links.

Components can vary with respect to their complexity and the level of abstraction at which they are defined. They could be as simple as a *connectionist unit* (e.g., a perceptron [Minsky and Papert1969]) and as complex as a full-fledged *condition-action rule interpreter* (e.g., SOAR [Laird *et al.*1987, Rosenbloom *et al.*1993]). Computational components of an agent's architecture can be created and destroyed during the lifetime of an agent by other architectural components.

[1] APOC components are based on the "behavior nodes" described by Scheutz. [Scheutz2001]

Since user-defined algorithms (e.g., search algorithms that browse the web for information) are in general implemented as part of APOC components, ADE allows for distributing computations in terms of asynchronous computational units and communication links among them (see the example in section 8). It is also possible to run different algorithms in different parts of the architecture and to change them over the lifetime of the agent. Thus, it is possible to express and study different designs of various mechanisms within the ADE framework (e.g., how to do behavior arbitration). Furthermore, the resource requirements and computational costs of an architecture can be determined and compared to other architectures implementing different algorithms for the same task in ADE [Scheutz and Andronache2003a].

ADE provides functionality for implementing agent architectures for simulated and robotic agents. An integrated server-client subsystem allows components of the architecture to connect directly to robots (see the example in section 3) or remote agents in a simulated environment in order to control them. ADE was particularly structured with the goal of designing complex agents in mind. Hence, there is support for (1) building more complex components out of simpler ones using a "grouping mechanism" for components, (2) "online inspection and modification" of all parts of the architecture (components and links can be removed and new ones can be added in the running virtual machine), and (3) distribution of the architecture over multiple hosts in a platform independent way.

2.2 ADE: The User Interface

The ADE environment was designed to allow users to access, inspect, and modify an architecture at any time during its development process: from the original design of the architecture, to the testing of components, to the execution of the complete architecture. For this, the system is divided into an architecture layout section and a virtual machine section. In the former, the components of the architecture are specified and the connectivity among them is established, thus defining the overall architecture layout. In the latter, the running architecture is maintained, which is updated dynamically and subject to runtime modifications.

Figure 1 shows a screen shot of the basic run-time environment: the left half shows the architecture layout of the system, while the right half shows the run-time virtual machine. Of note in Figure1 is the *Network Setup* menu item, which provides a variety of functions related to the distributed nature of the environment, such as viewing and modifying the structure of the network on which ADE is running.

In the following, we describe the functionality of each subspace individually.

Architecture View

In the architecture view, boxes represent the types of components that can be present in the run-time virtual machine (i.e., the instantiated architecture). Users can add customized components and display their information.

Links in ADE are created and configured through a link creation tool. In the graphical interface, edges indicate (by the direction of the arrow) the direction of the links in the architecture. Link information can be obtained by clicking on the arrow.

Virtual Machine View

In the running virtual machine, boxes indicate actual computational components present in the instantiated architecture and edges represent instantiated APOC links. Multiple arrows can be present along each edge, indicating each type of instantiated link.

Users can insert components directly into the running virtual machine if the insertion operation does not violate the architectural restriction on the number of components which can be simultaneously present in the instantiated architecture. If a violation is detected, the instantiation operation fails. Links can also be inserted in the running virtual machines. If a link is not available in the description of the architecture then the insertion operation fails.

Operating Modes

ADE has three operating modes: an *editing mode*, a *synchronous agent update mode*, and an *asynchronous agent update mode*.

In edit mode, a user can modify both the architecture layout and the architecture present in the run-time virtual machine. Supported architecture layout operations are: adding/deleting a type of component, modifying the maximum number of allowable components of a type in the running virtual machine, adding a link between two component types, and deleting a link between two component types. In the running virtual machine, a user can add a component, delete a component, add a link between two existing components, and delete an existing link within the constraints imposed by the architecture layout.

The synchronous mode provides the means for a synchronous update of the agent architecture. In this mode, each component in the running virtual machine completes one update cycle and waits for the other components to complete their cycle.

In asynchronous mode, all components (and all links) update asynchronously based on their internal timing without synchronizing their state with other components. This is particularly interesting for distributed applications, where synchronization is not required and would result in a severe performance bottleneck. In some architectures (e.g., subsumption [Brooks1986, Brooks1991]) asynchronous update is even part of the architecture specification, and thus forces the agent designer to make no assumptions about the timely update of states and delivery of information. Note, however, that each node will still attempt to update at its update frequency if permitted by the operat-

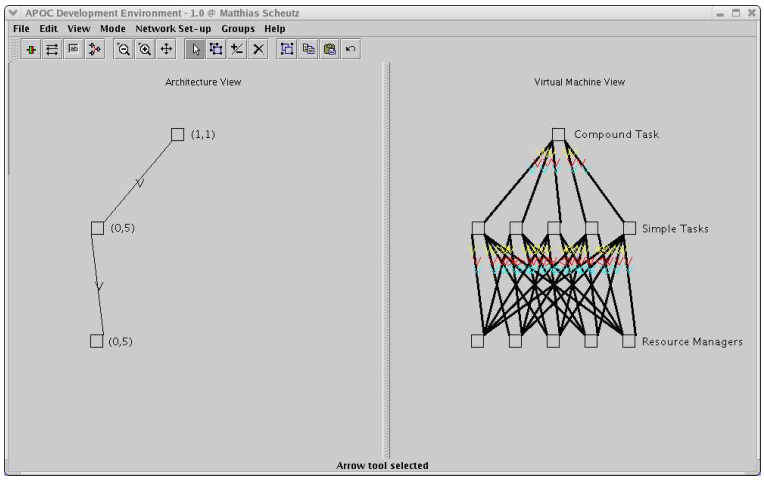

Figure 1: ADE Interface: The left side shows the types of components which can be present in the architecture and their possible interconnections. The right side shows the components (boxes) and communication paths (lines) in the running virtual machine

ing system.[2]

2.3 ADE: The Supporting Environment

ADE's supporting environment provides the infrastructure to distribute agent architectures and to operate virtual as well as robotic agents. It consists of a (global) *registry* (which dynamically keeps track of the elements of the distributed environment) and four types of servers: *system servers* (such as APOC virtual machines and graphical user interfaces), *agent servers* (which provide a "body description" for virtual agents or the interface to robots), and *utility servers* (which provide additional distributed services that are not part of the agent architecture).

Registry

The registry is a repository of available services as provided by the various servers. In particular, it provides updated information of the location of all participating APOC, GUI, agent, and utility servers, and maps APOC components that request a particular service to agent or utility servers, thus acting as a transaction broker between a client requiring a resource and available resources. The client contacts the registry and requests the desired resource either by specifying the type of resource required (if no specific instance is required) or by identifying a specific resource (by virtue of its unique ID or location within ADE).

Servers

A server in ADE is a computational unit that represents a resource of the ADE system. Each ADE server is an independent computational resource that typically runs in its own operating system process and can be started independently. After startup, each server first contacts the *registry* and specifies the maximum number of client connections that it can support (as well as any additional restrictions regarding the connection, e.g., allowed domain names). We describe below each of four server types–APOC server, GUI server, agent server, and utility server–and their functions within ADE.

Each APOC server is an independent entity, with capabilities for instantiating and deleting new components and links. APOC servers control their locally instantiated components and maintain connections to other APOC servers as well as to all available GUI servers in the ADE system. In effect, APOC servers are containers for individual ADE architectural components and resource managers for the ADE system.

Gui servers are visual resources which allow the user to view the architecture and its instantiated virtual machine. Each GUI server maintains connections to all available APOC servers.

Agent servers provide access to the body of a virtual or robotic agent by establishing connections to its sensors and effectors. Upon startup, they automatically connect to the registry announcing their service and then wait for clients to connect.

Utility servers provide a service which may be needed by one or more APOC components. Generally speaking these servers implement computationally expensive operations. Utility servers may require additional connections to agent or utility servers (established through the registry) in order to obtain the data they are supposed to process.

An example of the different relationships among the server types and the registry is illustrated in Section 3.

ADE Configuration File

In order to simplify the startup of an ADE system, ADE uses a global configuration file. In a typical set-

[2] On realtime operating systems, this update frequency can be guaranteed.

ting, this configuration file is run in one GUI server (which needs to be started manually), which then automatically starts the rest of the ADE system from a single host.[3] In the bootstrapping process, the GUI server reads the configuration file and opens connections to each host in the "hosts list". Once a connection is established to a remote host, an APOC server is started there.

Having described the building blocks of ADE, we now focus on the facilities provided by ADE for the design of agent architectures.

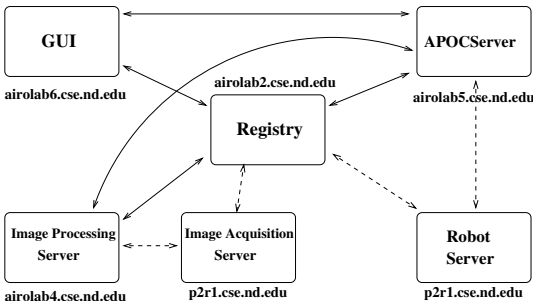

Figure 2: ADE Set-up for ball retrieval in the robot experiment

3 Example of Using ADE for Robotic Agents

Robotic agents pose a challenge for agent designers, as they operate in realtime and timely updating of components is crucial for the proper operation of the architecture. In this section we show how a robotic agent can be implemented in ADE.

3.1 Agent Task

The task for the robot is a target localization task, in which a target (i.e., "orange ball") has to be located in an environment with obstacles (e.g., an office space with boxes, chairs, etc.). The robot needs to locate the ball, and coordinate its position with the position of the ball in such a way that a plan can be created to bring the agent from its current location to the ball location, while avoiding obstacles.

The opening between two obstacles is sometimes not wide enough for the robot to pass through. Therefore, the robot needs to go around one of the obstacles in order to reach its goal. As a result, the robot also needs to be able to handle situations where it loses sight of the ball for a certain period of time. One architecture for such an agent is described in the following section.

3.2 Agent Architecture

The server structure for the experiment is presented in Figure 2. Each box represents a separate ADE server running independently, while links represent communication pathways between servers (dashed links indicate wireless ethernet). The name of the computer on which each component was run is specified below the component.

[3]The registry is started separately by the user, not through the bootstrapping process.

In the robot architecture used for the experiment a *Supervisor* uses visual information from the *VisionSensor* to determine if the robot is stuck in a situation where it is not making progress towards achieving its goal. If such a determination is made, the *Supervisor* switches off *CooperativeDecision* and turns on *CompetitiveDecision*. Unlike *CooperativeDecision*, which combines all the directional information from the *SonarProcessing* nodes to produce an overall directional vector, *CompetitiveDecision* uses a competitive selection mechanism, discarding all but the most relevant information for its task, in this case, wall following. Upon regaining sight of the target, the *Supervisor* shuts off *CompetitiveDecision* and reactivates *CooperativeDecision*. Once the robot has reached the ball, its task is complete. Details of the robot architecture can be found in [Scheutz and Andronache].

3.3 Setup in ADE

The setup steps for the experiment are as follows: (1) start the *Registry*, (2) start the *GUI*, (3) start the *RobotServer* and the *ImageAcquisitionServer* (this step and the previous one are interchangeable), (4) start the *ImageProcessingServer*, (5) define the robot architecture (using the graphical tool and predefined components), (6) run the experiment by switching to *Synchronous* or *Asynchronous* Mode.

The entire set-up process can be completed by using a script, thus allowing the user to issue a single command to initialize an ADE system.

Various components in the architecture require access to robot related resources, such as its sensors and effectors, as well as other information gathered from utility servers. In this experiment, one component requires visual information, a second requires access to robot sonar and motor information, while a third requires access to the robot motor effectors. Each of these components contacts the registry, requests the server it requires and sets up direct communication channels with that server. As a result the registry acts only as a resource manager, and will not become a bottleneck for communication within the system.

For visual processing, a standard blob-detection algorithm was used to identify the ball and run on a separate utility server, the *ImageProcessingServer*, to improve the parallelism of processing and thus the performance of the system. Two of the servers, the *RobotServer* and the *ImageAcquisitionServer*, can only run on the on-board computer of the robot used in the experiment, as they need direct access to hardware components (robot sensor and effectors, as well as camera sensors and effectors). The *Registry* was run on a Sun workstation, while the *GUI* server and the *APOCServer* were run on PCs.

ADE architecture inspection mechanisms can be useful in several ways: (1) inspecting the *VisionNode* allows the user to see what the agent "sees," (2) inspecting the *CooperativeDecision* and *CompetitiveDecision* nodes allows the user to view the commands being sent to the motors and identify discrepancies between expected and actual behavior, (3) inspecting the links between the *VisionNode* and the *SupervisoryNode* allows the user to see the information available to the decision node and to ascertain any delays in communication which may be present in the system.

The user can also stop the robot and analyze its position in the environment. Based on this analysis expectations about the contents of the architectural components (e.g., sonar sensor values, ball location) the user can pinpoint design flaws in either individual components or the layout of the architecture.

The above example illustrates the "real-time" nature of ADE, as its distributed nature enabled it to appropriately control a robotic agent. This feature separates ADE from most existing agent toolkits, where robot control is external to the toolkit itself. It should be noted that due to its distributed nature and its robot control facilities, ADE could also be used to implement RCS-based systems. [Albus1992]

4 Discussion

Many agent software tools are targetted at the design of agents and agent systems. We begin this section by placing ADE in the context of these toolkits.

DACAT [Barber and Lam2002, Barber and Lam2003] is an architecture design tool which provides the user with a customizable set of competencies from which the user can choose the ones relevant to her agent design. Like ADE, it provides a fully graphical environment, in which the relationship among architectural elements is visualized. However, DACAT stops at indicating the structure among components at the level of the functionality of an agent, without actually implementing it, whereas ADE allows for the implementation, running, and testing of any architecture specified within it.

IBM's ABE [ABE] is a tool which provides some architecture design support, e.g., a set of *adapters* (for agent-environment interaction), *engines* (forward chaining inferencing tools), and *libraries* (support for rule and fact authoring tools, to organize, group, and control the inferencing materials that are used by the engine). However, ABE imposes a rule-based design philosophy on its agents, in contrast to ADE, which supports rule-based systems, but also allows for alternative architectures not based on rule interpreters (e.g., subsumption architectures [Brooks1986]).

Many agent systems are concerned with mobile software agents, which can roam the internet. These systems (AGENTBASE [AgentBase], AGLETS [Aglets], BDIM/TOMAS [Busetta and Kotagiri1998], RETSINA [Sycara and others2003], and others) focus on supporting efficient and secure communication among agents as well as improving their mobility. However, only limited support is provided for the design of an agent architecture beyond the communication APIs and virtually no support is present for robotic agents in these systems. In contrast, ADE treats robots and virtual agents the same from a designer's perspective and allows designers to implement any architecture methodology based on its implementation of the universal architecture framework APOC.

The AGENT FACTORY system [Collier and O'Hare1999, Collier *et al.*2003] is an environment for agents which use BDI architectures. [Kinny *et al.*1996] It is similar to ADE in that it provides support for an agent architecture design, from a high level specification of the architecture to its implementation and deployment and, furthermore, allows the definition of agents that are not strictly based on the BDI framework. Still, the main focus of the AGENT FACTORY system is on BDI-based software systems, and thus differs markedly from ADE. Furthermore, it neither provides ADE's seamless support for single and multi-robot systems, nor ADE's capability of distributing architecture components over multiple computers in an OS-independent fashion.

SIMAGENT is a toolkit designed specifically for the exploration of agent architectures. Like ADE it supports the specification of architectures at various levels of complexity (e.g., symbolic mechanisms can coexist and communicate with neural networks). However, SIMAGENT only provides basic library functionality for the design of agent architectures for single and multi-agent systems (e.g., a basic agent class, a condition-action rule interpreter, etc.) and currently has no support for distributing agents over multiple hosts or for controlling robots, both of which are core features in ADE.

In sum, ADE integrates desirable features from different agent systems such as a general architecture framework APOC for the definition of agent architectures, support for distributed architectures which can change dynamically, support for communications among agent and agent mobility. None of the above discussed agent systems combines all of these features within one system. Additionally, ADE provides seamless support for single and multi-agent architectures for virtual and robotic agents and a user-friendly, multi-

user graphical interface that allows multiple designers to work collaboratively on agent architectures.

We demonstrated ADE's utility as design, implementation, test, and run-time tool with a robotic agent The example was primarily intended to illustrate one architectural design and setup within the given space restrictions to show ADE flexibility. Consequently, the robot task was kept simple. Several much more complex architectures have been implemented or are currently under development for a variety of more complex tasks, especially in robotic settings. [Scheutz and Andronache2003a, Scheutz and Andronache2003b]

References

[ABE,] IBM agent building environment. http://rtdcs.hufs.ac.kr/docs/intelligent_java/abe/.

[AgentBase,] Agentbase. http://www.sics.se/~market/toolkit/.

[Aglets,] IBM aglets. http://www.trl.ibm.co.jp/aglets/.

[Albus, 1992] J. S. Albus. A reference model architecture for intelligent systems design. In P. J. Antsaklis and K. M. Passino, editors, *An Introduction to Intelligent and Autonomous Control*, pages 57–64, Boston, MA, 1992. Kluwer Academic Publishers.

[Barber and Lam, 2002] K. S. Barber and D. N. Lam. Architecting agents using core competencies. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 90–91. ACM Press, 2002.

[Barber and Lam, 2003] K.S. Barber and D.N. Lam. Specifying and analyzing agent architectures using the agent competency framework. Technical Report TR2003-UT-LIPS-02, University of Texas at Austin, Austin, TX, 2003.

[Bellifemine *et al.*, 2000] F. Bellifemine, A. Poggi, G. Rimassa, and P. Turci. An object-oriented framework to realize agent systems. In *Proceedings of WOA 2000 Workshop*, pages 52–57, Parma, May 2000.

[Brooks, 1986] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.

[Brooks, 1991] Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.

[Busetta and Kotagiri, 1998] P. Busetta and R. Kotagiri. An architecture for mobile bdi agents. In *Applied Computing 1998, Mobile Computing Track, Proceeding of the 1998 ACM Symposium on Applied Computing (SAC'98)*, pages 445–452, 1998.

[Collier and O'Hare, 1999] R.W. Collier and G.M.P. O'Hare. Agent factory: A revised agent prototyping environment. In *Proceedings of the 10th AICS Conference, Irish Artificial Intelligence and Cognative Science Conference*, 1999.

[Collier *et al.*, 2003] R.W. Collier, G.M.P. O'Hare, T. Lowen, and C.F.B. Rooney. Beyond prototyping in the factory of the agents. In *Proceedings of the 3rd Central and Eastern European Conference on Multi-Agent Systems (CEEMAS'03)*, pages 383–393, 2003.

[Kinny *et al.*, 1996] 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 ina Mult Agent World, MAAMAW 96*, volume 1038 of Lecture Notes in Artificial Intelligence, pages 56–71. Springer-Verlag, 1996.

[Konolige, 2002] Kurt Konolige. Saphira robot control architecture. Technical report, SRI International, Menlo Park, CA, April 2002.

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

[Minsky and Papert, 1969] M. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, 1969.

[Nwana *et al.*, 1997] H. Nwana, D. Ndumu, L. Lee, and J. Collins. Zeus: A collaborative agents toolkit. In *Proceedings of the 2nd UK Workshop on Foundations of Multi-Agent Systems*, pages 45–52, 1997.

[Rosenbloom *et al.*, 1993] P.S. Rosenbloom, J.E. Laird, and A. Newell. *The Soar Papers: Readings on Integrated Intelligence*. MIT Press, Cambridge, MA, 1993.

[Scheutz and Andronache,] Matthias Scheutz and Virgil Andronache. Architectural mechanisms for the dynamic selection and modification of behaviors in behavior-based systems. In Preparation.

[Scheutz and Andronache, 2003a] Matthias Scheutz and Virgil Andronache. Apoc - a framework for complex agents. In *Proceedings of the AAAI Spring Symposium*. AAAI Press, 2003.

[Scheutz and Andronache, 2003b] Matthias Scheutz and Virgil Andronache. Growing agents - an investigation of architectural mechanisms for the specification of "developing" agent architectures. In Rosina Weber, editor, *Proceedings of the 16th International FLAIRS Conference*. AAAI Press, 2003.

[Scheutz, 2001] Matthias Scheutz. Ethology and functionalism: Behavioral descriptions as the link between physical and functional descriptions. *Evolution and Cognition*, 7(2):164–171, 2001.

[Sloman, 1999] A. Sloman. Sim_agent help file, 1999.

[Sycara and others, 2003] Katia Sycara et al. The retsina mas infrastructure. *Autonomous Agents and Multi-Agent Systems*, 7(1):29–48, 2003.