ADE - Steps Towards a Distributed Development and Runtime Environment for Complex Robotic Agent Architectures

Matthias Scheutz

Artificial Intelligence and Robotics Laboratory

Department of Computer Science and Engineering

University of Notre Dame

Notre Dame, IN 46556, USA

email: 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 robotic agent architectures. ADE is unique among robotic architecture development environments in that it is based on a universal agent architecture framework called APOC, which allows it to implement architectures in any design methodology, and in that it uses an underlying multi-agent system to allow for the the distribution of architectural components over multiple host computers. After a short exposition of the theory behind ADE, we present the multi-agent system setup and give an example of using ADE in a multi-robot setting. A general discussion then highlights some of the novel features of ADE and illustrates how ADE can be used for designing, implementing, testing, and running agent architectures.

1 Introduction

Virtual and robotic agents share many features, from the need to be able to deal with various kinds of incoming information, to the generation of appropriate outputs or behaviors. *Qua agents*, both kinds process information

autonomously and need to ensure their proper operation throughout their lifetime. Yet, robotic agents are particular in that they operate in physical space under real-time conditions, i.e., the timing of their sensory processing and action selection is determined and constrained by the real-world environment in which they operate. Consequently, robotic agents have specific computational and architectural demands that are usually not required for virtual agents. Mobile robots, for example, typically have an obstacle avoidance mechanism that is connected to the robots' distance sensors and will have to be able to interrupt the current action in order to be able to take control of the robot's motors if the robot is about to bump into an obstacle.

Although provisions in the architectural design can help ensure the reactivity for single-CPU robotic systems to some limited extent, the computational limits of such systems are quickly reached when information from multiple sources needs to be processed and integrated in real-time to produce complex behaviors based on multi-modal incoming information and previously stored knowledge. A complex waiter robot, for example, that serves drinks and appetizers at receptions (e.g., Maxwell et al., 1999), has to perform several tasks at any given time: (1) it has to find and recognize people (to serve drinks and appetizers to), yet avoid bumping into them or other "obstacles" such as chairs, tables, etc. (2) it has to have knowledge about its environment, needs to remember locations, and needs to be able to navigate from one location to another and update its current position in the environment (e.g., via a combination of perceptual or proprioceptive mechanisms); and most importantly (3) it has to interact with people in natural language, which itself involves a whole set of different competences, from speech processing, to syntactic parsing, semantic analysis and language understanding, to reasoning, planning, text generation and speech production, and many more. Obviously, the computational demands of all these "functions" by far exceed the computational power of a single-CPU system.

Common programming environments for robotic agents such as *Saphira* (Konolige, 2002), *Carmen* (Thrun, Fox, Burgard, & Dellaert, 2000), or *Player/Stage* (Gerkey, 2003) acknowledge the need for distribution of robotic architecture by allowing different components of an architecture to be distributed over multiple computers. Yet, they do not themselves support the design, testing, and running of distributed agent architectures for complex robotic agents. What is needed is an environment that hides the implementation details of the underlying middleware necessary to distribute architectural components, while providing tools that allow for easy access of robotic devices and support the development of architectural components that need to operate in real-time. In this paper,

we report our attempts at developing such a system. Specifically, we propose the ADE system (under development in our lab) as a first step towards a fully distributed development and runtime environment for complex robotic agents. 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 4) 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) integration of existing architectural components in a distributed architecture, (2) "online" inspection and modification of all parts of the architecture (components and their connections 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. ADE also provides a multi-user graphical user interface environment, in which agent architectures can be defined and implemented at different levels of abstraction. Through this interface, ADE allows users to monitor components in running distributed architectures in real-time. Finally, ADE incorporates security, failure detection and recovery mechanisms that are crucial to distributed robotic systems and are unique among robotic development environments.

In the following, we will first provide an overview of the theoretical underpinnings of ADE–the agent architecture framework APOC–which is at the heart of ADE's flexibility as an architecture development environment. Then we will discuss ADE's underlying multi-agent system (MAS) infrastructure, which allows for the distribution of components of the robotic architecture over multiple computers in a platform independent way. An example of a multi-robot system, in which two robots need to find and follow a ball, is then used to illustrate various aspects of ADE in a practical setting. The subsequent discussion further highlights some of ADE's features that distinguish it from other robotic architecture development environments and agent systems. Finally, the conclusion discussion summarizes ADE's current status and provides a brief outlook for future development directions.

2 ADE's Theory: The APOC Architecture Framework

One major problem with current robotic development systems is that they are either based on a particular architecture methodology (e.g., the schema-based approach of AuRA (Arkin & Balch, 1997) or the subsumption-based approach of the behavior-language (R. Brooks, 1990)), or that they do not provide particular support for any archi-

tecture (e.g., Player/Stage, etc.). The approach taken in ADE was to steer a middle line between the two, i.e., to provide as many tools as possible to support the development of robotic architecture while not forcing the designer to commit to any particular architecture. It was possible to achieve this flexibility by building ADE around the agent architecture framework APOC (Andronache & Scheutz, 2002, 2003; Scheutz & Andronache, 2003; Scheutz, 2005; Andronache & Scheutz, 2005).

APOC is a general, universal agent architecture framework in which any agent architecture can be expressed and defined. The APOC framework views agent architectures as networks of (possibly heterogeneous) computational *components* connected by four types of *links* called *A*ctivation, *P*riority, *O*bserver, and *C*omponent links (hence the name "APOC"). The four link types 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-links, P-links, and O-links.

In the following, we will briefly describe the functionality of APOC components and links, which is directly implemented in ADE.

2.1 APOC Components

Each *component* in APOC is an autonomous computational unit with activation and priority values, which are used for process management (see Section 2.3). 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 & Papert, 1969)) and as complex as a full-fledged *condition-action rule interpreter* (e.g., SOAR (Laird, Newell, & Rosenbloom, 1987; Rosenbloom, Laird, & Newell, 1993)) and can be created and destroyed during the life-time of an agent.

Components can receive inputs from and send outputs to other components via links connected to their input and output ports. Inputs (from incoming links) are processed and outputs (to outgoing links) are produced according to an *update function F*, which determines the functionality that the component implements, i.e., the mapping

from inputs and internal component states (e.g., the current activation and priority values) to outputs and updated internal component states (e.g., new activation and priority values). The update function thus provides the specification for a computational process that in an instantiated component continuously updates the component's state. The particular algorithm for implementing the update function F has to be defined separately for each component type employed in an architecture. Figure 1 summarizes the basic structure of an APOC component.

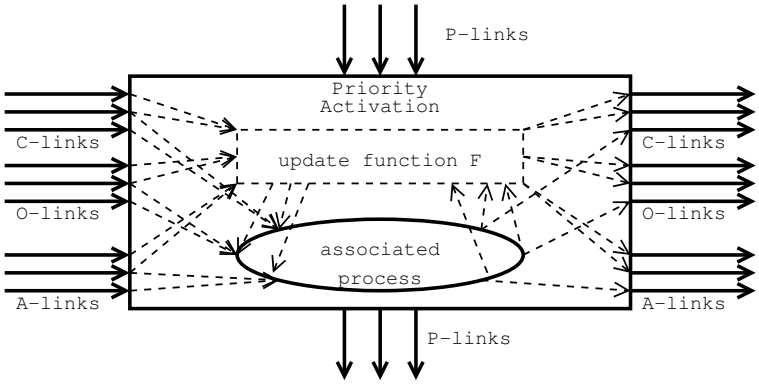

Figure 1: The basic structure of an APOC component.

Different from components in other formalisms such as schemas in *RS* (Lyons & Arbib, 1989.) or the "augmented finite state machines" (AFSMs) in the subsumption architecture (R. A. Brooks, 1986), an APOC component can also have an "associated process" (in addition to the computational process updating the state of the component), which it can *start*, *interrupt*, *resume*, or *terminate*. The associated process can either be a physical process external to the architecture (e.g., a process controlling the motors in a wheeled robot), or a self-contained computational process that takes inputs from the component and delivers outputs to it (e.g., a process implementing a blob detection algorithm that takes an image and returns all blobs of a particular color). One way of construing the APOC component concept is thus to view it (i.e., the update function F) as a *process manager* of its associated process. This construal makes it possible to separate information concerned with architecture-internal processing, i.e., *control information*, from other information (e.g., sensory information that is processed in various stages).

APOC distinguishes between component and link types, as they are specified in the architecture diagram, and the corresponding run-time entities, i.e., instances of components and links in the APOC virtual machine. The architecture diagram defines the generic structure of the run-time architecture instance and thus imposes limits and restrictions on the kinds of components and links that can be instantiated at any given time. It is,

for example, not possible to instantiate a link type between two components if that type has not been specified in the architecture digram. Moreover, APOC specifies a simple resource control mechanism at the architecture level: for each component in the architecture diagram, a maximum number of instances needs to be specified, which fixes the upper limit of simultaneously present instances of that type in the running virtual machine. For example, a component implementing a one-step lookup-ahead of a planning mechanisms may be able to create only 10 instances, thus allowing for plan sequences of at most 10 steps. While it may not be possible to predict the exact resource requirements of an architecture without knowing the specific inputs at any given time, this mechanism nevertheless allows agent designers to define an upper bound on the computational requirements of an architecture, which cannot be exceeded by any running architecture instance. This is critical for autonomous robots, where unbounded resource requirements might cause the control system to slow down and eventually fail to function properly.

In sum, the state of an APOC component can be characterized as an 8-tuple:

$$\langle pri, act, inst_{def}, inst_{max}, links_{in}, links_{out}, update, process \rangle$$

where *pri* is the priority of the node, *act* is the activation of the node, $inst_{def}$ is the number of instances of the node which are instantiated automatically by the system (with $inst_{def} \geq 0$), $inst_{max}$ is the maximum number of instances of the node which can exist simultaneously in the system. (with $inst_{max} \geq inst_{def}$ and $inst_{max} > 0$), $links_{in}$ and $links_{out}$ are incoming and outgoing links from and to other nodes respectively, $update$ is the component's update function, and finally *process* is its associated process. Some members of the tuple ($links_{in}$, $links_{out}$, $update$, and *process*) are themselves structures. $links_{in}$ and $links_{out}$, for example, can be broken down according to link type, while $update$ and *process* are pairs containing the data they act on and the operations performed on that data. The update function F specifying the evolution of a component state over time is a mapping

$$F : \mathcal{ST} \times \mathcal{PST} \times \mathcal{IN}^k \mapsto \mathcal{ST} \times \mathcal{OUT}^l \times \mathcal{OP}^{3+k+l}$$

where $\mathcal{ST}$ is the set of internal states of the component, $\mathcal{PST} = \{$RUNNING, INTERRUPTED, FINISHED, READY, NOPROC$\}$ is the set of possible states of the associated process as mentioned above (NOPROC indicates that no

process is associated with the node), $\mathcal{IN}$ is the set of input states, $\mathcal{OUT}$ is the set of output states, and $\mathcal{OP}$ the set of operations a component can perform. The set of operations can be subdivided into operations that a component can perform on its associated process and possibly the processes of other components $\mathcal{POP}$ ={START, INTERRUPT, RESUME, NOOP}, those it can perform on itself and other components *per se* $\mathcal{COP}$ ={INSTANTIATE, TERMINATE, NOOP}, and finally, those it can only perform to manipulate its priority $\mathcal{SOP}$ ={INCR, DECR, NOOP}, such that $\mathcal{OP} = \mathcal{POP} \cup \mathcal{COP} \cup \mathcal{SOP}$. Note that the NOOP is used formally if no operation is performed. All these sets together fix the set of possible architecture topologies of components as well as their operations, input and output state types.

We now briefly describe the four types of links that are used to connect APOC components. "V_s" will be used to denote the source component of a link and "V_d" will denote the destination component.

2.2 The Activation Link (A-link)

The first type of link, the A-link, is a generalized version of the "activation" passing links by Tyrrell (1993) and Maes (1989) used for general communication between components. It is defined as a quadruplet $(V_s, V_d, data, op)$, where $data$ is an arbitrary data structure to be sent through the link, and op is an operation performed on $data$ as it is passed through the link. The operation op performed on $data$ can either be the identity mapping, or can be used to perform transductions of $data$ (e.g., op can be a weighting of a numerical value, a mapping from one representational space into another, etc.).

2.3 The Priority Link (P-link)

The second type of link, the P-link, is defined as $\langle V_s, V_d, pri, act, S \rangle$, where act and pri are the activation and priority of the source component, respectively (with $act, pri \in [0,1]$), and S is a signal (i.e., *start*, *interrupt*, *resume*, or *terminate*). P-links are used for the control of associated processes of components based on a simple arbitration mechanism to determine which of the components with outgoing P-links to a given component C gets to control the associated process of C: the signal on the incoming P-link to C with the highest priority determines the action to be performed on the associated process. If two or more components have the same priority, then their activation values will be used as a tie breaker. If the activations are also the same, then no signal will be sent to the process.

This simple priority-based process control mechanism can then be used to implement other process arbitration strategy by virtue of the being able to control priority and activation values of components (for details, see Scheutz & Andronache, 2004).

2.4 The Observer Link (O-link)

The observer link is defined as $\langle V_s, V_d, val \rangle$ where *val* is the information passed through the link from V_d to V_s. The purpose of O-links is to provide a way for nodes to observe the state of other nodes in an "non-intrusive" fashion (i.e., through a one-way communication, which does not affect the functioning or state of the observed node). This mechanism can be used to implement elaborate monitoring mechanisms in an agent architecture. In ADE it is used to implement inspection tools directly at the level of the architecture (see Section 5.3).

2.5 The Component Link (C-link)

The component link is the most complex link in that it allows for the instantiation of other links and components at runtime subject to the resource limits specified in the architecture diagram. It is defined as $\langle V_s, V_d, port, S \rangle$, where *port* is the port of V_s the component on which the operation (*instantiate* or *terminate*) specified by signal S is to be performed. What will be instantiated depends on the state of the architecture and the type of link which connects the *port* of V_s and V_d in the architecture diagram. For example, if two components A and B are connected via an A-link on port 10 in the architecture diagram, then the signal *instantiate* on port 10 will cause the creation of an A-link between component A and B if B is already instantiated, or will first instantiate B and then create the A-link. In both cases, instantiation is subject to the specified resource constrains (e.g., if the maximum number of instances of a component of type B has already been reached in the architecture instance, then the C-link operation will fail and component V_s will be notified).

2.6 Implementing the APOC Specification

APOC as a formalism, in which to cast agent architectures in a unified way, does not suggest particular implementations nor does it limit the kinds of programming languages or environments that can be used to implement it. The choice for an appropriate environment will largely depend on the application domain, i.e., the kinds of agents

for which architectures will be defined in APOC. In the present context of distributed architectures for complex robots, two major constraints are imposed on an implementation environment: (1) it needs to support networking and distribution of objects in order to allow for distributed agent architectures, and (2) it needs to be able to deal with situations that commonly come up in the context of robotic systems. Here we only give three simple examples together with possible solutions (as provided by ADE):

Robot Scenario A: Suppose an autonomous robot using an architecture distributed over several computers is intended to perform a given task, but does not exhibit the right behavior without any apparent cause for the failure. In such a case, online inspection tools that allow to monitor and log the states of different parts of the architecture (including the states of the robot's sensors and the effectors) in real-time can help to track down the problem.

Robot Scenario B: Suppose the architecture of an autonomous robot is distributed over two computers. One of the computers, which is connected to the robot's motor devices, suddenly crashes, leaving the system in a state where no more movements can be performed. In such a case, a supervisory mechanisms that periodically checks whether all participating hosts are up can be used to restart components of the architectures either on the very same host that crashed or on another host if the previous one is not available any longer.

Robot Scenario C: Suppose a mobile robot is running a "robot server" program on its on-board computer, which is responsible for providing access to the robot's sensors and effectors. The server is connected to an off-board computer running the main architecture via a wireless link. During a test run, the robot drives outside the wireless range leaving the server disconnect from the main architecture, while the motors are still running. In such a case, a mechanism on the server side that detects the failure to connect to host running the architecture can trigger emergency behaviors or shutdown of the mobile robot.

These are only a few of the many situations that a runtime environment for distributed robotic architectures has to address and be able to cope with.

While it is possible to start with an existing multi-agent system such as JADE (Bellifemine, Poggi, & Rimassa, 1999) or RETSINA (Sycara et al., 2003), which provide all the necessary infrastructure to account for (1), these systems would have been modified and adapted to meet the requirements of (2) (illustrated in the three robot

scenarios above), as they lack the features to support real-time, fault-tolerant architectures for robotic systems. Moreover, most multi-agent systems provide features that are not required in the context of distributed agent architectures (e.g., an agent communication language, an implementation-neutral definition of distributed or mobile objects such as CORBA, etc.) and would only cause computational overhead. Hence, it was decided to implement the necessary MAS infrastructure from scratch in JAVA, which already supplies crucial primitives (such as object serialization and RMI) for dealing with distributed and remote objects. The platform independence of JAVA, the process synchronization mechanisms at the level of the programming language as well as the reflection capabilities for online, dynamic class inspection are features that allow for an efficient, yet open design of the development environment that can be easily adapted and extended to suit particular processing requirements (e.g., the number of available computers, the robotic sensory and effector equipment, etc.).

In the following, we will provide an overview of the ADE implementation of APOC and the various design choices made to facilitate broad applicability of the system.

3 ADE Implementation: The Underlying Multi-Agent System

The basic methodology behind ADE is that of a network of connected client and server objects, where clients, i.e., objects on one host, are "local" representations of "remote" servers, i.e., objects on another host. The client-server subsystem is then used to implement various kinds of ADE *agents*, i.e., autonomous computational processes consisting of one or more clients and/or a server that communicate with other agents in the system (via client-server connections). For example, there are agents, called ADE *registry*, that implement a system-wide yellow page service, where other agents can register and advertise their services. Other agents, the APOC *servers*, implement "agent container" (cp. to JADE) for a subset of mobile agents called APOC *components* and APOC *links*, as well as a white page service, which individual APOC components use to announce their presence locally within the network of APOC servers. APOC servers are in charge of initiating the update of component and link agents, synchronizing information among them and across other servers, and providing basic service for user interaction.

ADE also provides several other types of agents: *robot servers* (which provide a "body description" for virtual agents or the interface to robots), *GUI servers* (which provide a graphical user interface to the complete ADE system), and *utility servers* (which provide additional distributed services that are not part of the agent architecture).

The different kinds of ADE agents together provide the infrastructure for a flexible, effective, user-friendly implementation of distributed robotic architectures: components of a robotic architecture (such as sensor and effector representations, feedback controllers, perceptual processors, planners, rule interpreters, reasoning engines, action selection units, etc.) as well as their various communication links are implemented by special APOC component agents and APOC link agents, which exclusively reside within APOC servers. APOC servers, in turn, provide the necessary run-time infrastructure and environment for APOC components and APOC links–the "APOC virtual machine"–according to the APOC agent architecture framework specification. Note that the term "agent" is used in two different senses: in ADE "agents" are the autonomous computational entities constituting the distributed multi-agent system, which implements robotic architectures; in APOC, "agent" refers to an entity whose control system is specified by an "agent architecture". Although we in general favor expositions that avoid terminological overloads, we believe that in context of this paper it is important to remain faithful to the established terminology, which directly reflects the research interests and objects of two distinct research communities in AI (i.e., the subfields of "distributed AI" and "agent architectures").

We will will now describe the main types of ADE agents in more detail.

3.1 The ADE Registry

An ADE registry is a agent that maintains the configuration of an ADE system. It provides a "yellow pages" service to all parts of the system: whenever a new server comes online, it registers with one registry and thus makes available information about its services. Clients can then contact the registry and request a desired resource either by specifying the *type* of the required resource (if no specific instance is required such as a server running on a particular host) or by identifying a specific resource (by virtue of its unique ID or location within ADE).

Registries also provide select access control to these services and keep track of the number of available and used connections to each server (all servers have a maximum number of connections they allow simultaneously in order to avoid computational overload). Moreover, ADE registries can save configurations of an ADE system and reload them if needed. Most importantly, they implement a monitoring system to check whether registered servers are still "up". If the failure of a server is detected, a registry can subsequently initiate a remote failure recovery process, which will attempt to bring up the server again on the remote host, or, if the host is not available

anymore, on another host from a set of possible hosts (if the set is specified). Since registries are servers themselves, registries can also bring up other registries in case of failures or system crashes, thus increasing the fault tolerance and reliability of an ADE system. All registries register with each other and automatically synchronize their registration information, again to be able to recover from crashes.

There are two ways to initialize an ADE system: (1) all registries are brought up first and then all servers are started (with the registry host information as startup arguments to allow them to register subsequently with one of the registries). Alternatively, (2) the first registry can be started with a startup script, which defines the configuration of the whole ADE system. The registry will then bring up other (optional) registries, start all servers, and load an (optionally) specified architecture file for execution. This second mode does not only allow for automatic startup of the ADE system, but also for dynamic reconfiguration (e.g., based on availability of resources or based on host or device failures–we will address some of the error detection and recovery features of ADE in Section 5.2, a detailed description of all mechanisms available in the MAS underlying ADE is currently under review (Scheutz & Kramer, under review)).

3.2 APOC servers

As already mentioned, APOC servers are containers for components and links, which provide the necessary mechanisms for instantiating, deleting, and updating both APOC components and APOC links. Each APOC server is in control of its locally instantiated components and their outgoing links. It maintains a list of all APOC component types that can be instantiated on it and registers these types with the registry. Components can only be instantiated on servers that registered their type. If a server has already instantiated the maximum number of components of a given type, it will forward the request to another server that registered that type and request instantiation on that server. If no such server is available or if the maximum number of component instances of that type in the whole ADE system has been reached, the instantiation fails.

APOC servers support two update modes for APOC components and links: *synchronous* and *asynchronous* update. In the former, an update cycle is complete only after all servers have updated all their components and links. In the latter, no restriction is placed on the update speed, i.e., inputs are processed and the updated information is relayed as quickly as possible followed by the next update cycle regardless of whether other servers were able

to update all their components in time. While asynchronous mode is not appropriate for applications that share a global state (e.g., an airplane reservation system which keeps track of seat reservations), it can be useful in robotic applications where state is mostly local to a component and information transfer does not critically depend on individual transactions, but rather requires the information to arrive at least once within a given time interval. Take a robotic vision system, for example, that directly controls the motors of a pan-tilt unit on which the camera is mounted. The image information taken from the camera is preprocessed locally on the camera unit in order to detect moving regions in the image, which might have to be tracked. If the moving region is close the image boundaries, a camera movement is typically initiated to center on the moving region in order to be able to track it. In such a system, waiting for the response of the unit that receives the image information to decide (after additional intensive visual processing) whether the object in the moving region was worth tracking is often not an option, as by the time the feedback is received, the moving object is out of view. Hence, a good strategy for the camera unit to track moving regions is to asynchronously update the camera's position as often as possible, while using any feedback information as soon as it becomes available for deciding whether or not to track the current moving region. In fact, some behavior-based architectures (e.g., Brook's subsumption architecture) are predicated on asynchronous data transfer between architectural components and thus lend themselves directly to distributed implementations without any additional synchronization requirements. Rather than guaranteeing consistency at the implementation layer (e.g., the middle-ware providing the high-level architectural primitives such as component types and communication links), it is the architecture designer's responsibility to develop a system that is capable of achieving its tasks without making any assumptions about the timely exchange of information across distributed components (e.g., in subsumption architectures timers will have to be employed to detect missing incoming data or timeouts on connections).

3.3 Robot servers

Robot servers in ADE provide easy access to the various physical devices attached to a robotic system, from the sensory devices such as cameras and frame grabbers, distance sensors (e.g., sonar, infrared, and laser sensors), bumper and pressure sensors, etc. to effectors like various motors for locomotion and grasping. Most importantly, robot servers provide high-level programming language access to embedded low-level robot controller boards

(e.g., JAVA language function calls such as FORWARD(10), which will make a robot move forward at a speed of 10 mm/sec.). As such, robot servers hide the implementation details of the controller board primitive and controller communication from the architecture designer and provide a unified programming interface to robotic agents that is independent of particular robotic systems (similar to systems like Player/Stage or Carmen). This abstraction is achieved in two steps: (1) first a set of generic JAVA interfaces is defined for common sensory and effector devices (e.g., a "sonar interface", or a "wheel motor interface"); (2) a generic robot server is defined consisting of a set of sensors and effectors (with their individual properties and update rates). It is then possible to specify particular sensory devices (e.g., a sonar ring consisting of 16 sonar devices, or a camera) or effector devices (e.g., a speech synthesizer or a gripper) and define a robot server that consists of exactly those devices and the additional, device-specific code that will provide access to them.

Robot servers register, like all other servers, with the ADE registry on startup, thus making themselves available for clients to connect. There are several ways of how robot servers can be used in an APOC architecture. For example, one APOC component could represent the entire robot with all its devices, in which case the device information can be made available on outgoing A-links or O-links (depending on whether data needs to be pre-processed and/or should be sent on a regular basis rather than being actively monitored as in the case of O-links). Alternatively, each robotic device could have its own representation at the architecture level, i.e., it could be implemented in a separate APOC component. The advantage of the latter approach is that devices with different update rates (e.g., a frame grabber vs. a laser range finder) will be able to run individually (and isochronously) at their maximum speed without slowing each other down.

One problem with distributed components accessing devices in parallel is that this can lead to inconsistent device states. For example, it is possible and very likely that erratic behavior will occur in a robot if multiple components send commands to the robot's motors in parallel. In ADE this can be prevented at two levels. At the level of the robot server, mutual exclusion is implemented in JAVA programming language primitives (i.e., "synchronized" writes to the motor control board are used as part of the function that implements motor primitives in JAVA). At the architecture level, motor devices can be represented by only one APOC component, which can then take requests for motor commands from other APOC components and arbitrate among them (Scheutz & Andronache, 2004). The latter approach allows for a better, much finer-grained control of behavior arbitration.

The device model used in robot servers in ADE has also other advantages (in addition to supporting the individual update rates of devices and allowing for controlled access to them). In particular, it allows for a fine-grained access model to devices according to device specific properties in a multi-user multi-robot environment (e.g., multiple read access to sensory devices, exclusive write access to effector devices), which guarantees a high level of parallelism wherever possible, while providing access restriction when needed. Moreover, it makes it possible to share physically separated devices across different architectures (e.g., a sound card installed on one robot could be accessed and used in the architecture of another robot in parallel to the first, thus allowing both robots to process sound information independently).

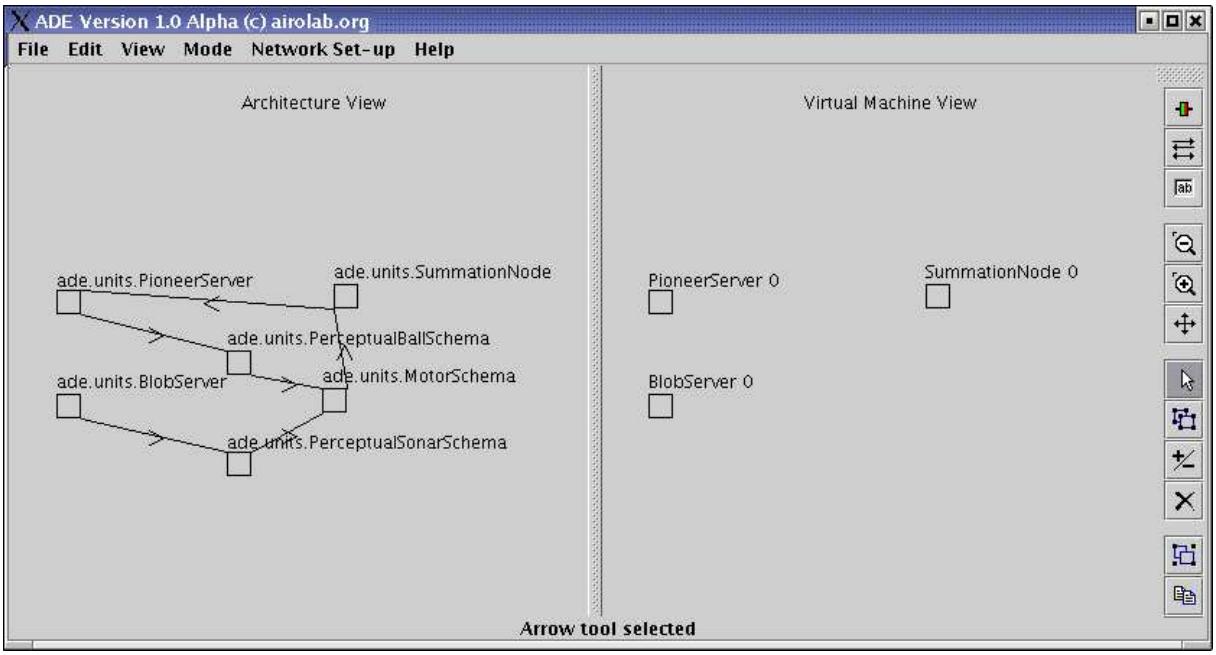

Figure 2: The ADE Workspace: The left side shows the architecture view with an architecture diagram consisting of different types of components that can be present in the virtual machine together with their possible interconnections. The right side shows the current state of the virtual machine running the components (boxes) and their communication links (lines). Of note is also the *Network Setup* menu item, which provides easy access to a variety of functions related to the distributed nature of the environment, such as viewing and modifying the underlying MAS network on which the distributed architecture is running.

3.4 GUI servers

GUI servers provide a multi-user graphical interface to an ADE system. They allow users to design, implement, and test architectures and, moreover, to inspect and debug distributed running instances in real-time. Multiple

instances of GUI servers can be active at the same time, thus allowing multiple users to work jointly on the same architecture or on different architectures, while being able to share resources (e.g., APOC, robot, and utility servers). The main graphical workspace is divided into an *architecture view* and a *virtual machine view*. In the former, the components of the architecture are specified and the general link connectivity among components (and thus run-time restrictions thereof) is established. In the latter, the running instance of an architecture is maintained and its graphical representation is updated dynamically. Figure 2 shows the divided workspace with the architecture view on the left, and the virtual machine view on the right.

In the architecture view, boxes represent the *types of components* that can be instantiated in the run-time virtual machine. Users can add customized components and display their information by double-clicking on them. 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.

In the virtual machine view, boxes indicate instantiated APOC components (present in the instantiated architecture) and edges represent instantiated APOC links (all of which can reside on different hosts). Multiple arrows can be present along each edge, indicating different types of instantiated links. Users can insert components directly into the running virtual machine if the insertion operation does not violate the architectural restriction on the number of simultaneously present components of a given type. If a violation is detected, the instantiation operation fails. Links can also be inserted in the running virtual machines. As with components, a link insertion operation fails if the type of link is not available between the two types of components the architecture diagram.

To support multi-user operations and prevent accidental destructive modifications of a given architecture, the access to all GUI components can be configured individually via a *permissions file* (which will typically defined by the system administrator or the project manager and loaded by the GUI server on startup). All users receive personal permissions files, with which they can start their individual GUI instances. Depending on the preset permissions, three main classes of user operations can be defined: *edit only*, *run only*, *edit and run*. In each of these modes, more fine-grained permissions can be set. For example, in *run mode* asynchronous update can be disallowed, thus allowing users to run architectures in synchronized mode only. Another example, in *edit mode*, would be to allow users to add only links, but not components. For both modes, it is possible to restrict access

to the distributed MAS infrastructure (i.e., the hosts on which to run APOC servers or the way architectural components can be distributed). In addition to and independent of their GUI instances, users can also be allowed controlled access to ADE registries (via a separate administration tool) in order to be able to configure parts of an ADE system (e.g., the hosts on which to run utility servers, the IP addresses of robots connected through wireless networks, etc.).

3.5 Utility servers

Utility servers are intended to reduce overall processing redundancy and increase system performance by providing services typically needed by multiple APOC components. Often times utility servers implement computationally expensive operations and run on dedicated hosts. For example, a vision processing system that takes images from a frame grabber and computes color blobs, motion blobs, and edge information in the image at the highest frame rate possible (e.g., 30 Hz say), will typically exhaust the computational power of a 3 GHz Pentium IV processor for images with 640 x 480 pixels. Hence, it is often better to run such services in utility servers on dedicated machines and provide the processed data via remote access than to provide it locally, yet having to share the CPU with other parts of the architecture. In ADE, multiple APOC components can represent the same utility server in an architecture, thus allowing for parallel access to the data processed in the utility server. In the case of the vision processor, for example, one APOC component could analyze the color information in the image looking for skin-like color in an effort to determine if faces are present, while another APOC component could process the edge information in order to detect shapes, yet another component could try to track moving objects based on motion blobs. All three components could either run on the same machine or could be distributed across multiple APOC servers.

As already mentioned above, utility servers in conjunction with robot servers and the ADE registry allow for a very fine-grained access model to devices in the ADE system. Suppose, for example, that two programming teams jointly develop an architecture for an autonomous robot and the first team is in charge of the visual processing, while the second team is in charge of the developing appropriate motor behaviors for the robot. Both teams share one robot, which is equipped with an on-board PC that has a frame grabber board connected to the robot's camera. It is then possible to run a "frame grabber" server on the robot, in addition to the "robot server". The frame grabber

server is configured in the ADE registry such that multiple APOC components (implementing frame grabber clients) can gain access to it, while the robot server is configured to allow only one client to access it at any given time. It is then possible for the behavior team to start their architecture requesting access to the robot and get the only client connection allowed for it, while the vision team can still access the frame grabber and process visual information at the same time. In a slightly different vein, suppose that the robot team now requires multiple connections to the robot server (as their architecture consists of multiple distributed client representations of the robot). It is possible to configure the ADE registry to allow multiple connections to the robot for the user ID of the behavior group, while still disallowing other groups from accessing the robot.[1]

This kind of access control is particularly useful in research or educational setting, where multiple teams of researchers or students need to share one robot, which they use for experiments. In such a case the ADE registry can be configured to allow only exclusive access to the robot's devices for each team, while allowing multi-client access within each team.

4 A Robotic Example of Using ADE

The previous sections provided information about the theory behind ADE as well as its design as a multi-agent system. In this section, we will take a brief look at the practical side of ADE as a tool for the design of distributed architectures for robots. We use a multi-robot ball following task to illustrate the flexibility of ADE as an architecture development and run-time environment for single and multi-robot applications (for other applications of ADE, see Scheutz, McRaven, & Cserey, 2004; Andronache & Scheutz, 2004; Scheutz & Andronache, 2004, 2003; Andronache & Scheutz, 2005). In this task, two robots have to locate and follow independently an orange ball placed in the environment. The robots used for this task are mobile Pioneer P2DXe robots with onboards 400 MHz Pentium II PC 104 boards running Redhat LINUX. They are both equipped with SONY ZOOM cameras mounted on pan-tilt units, PC 104 frame grabber boards, and wireless 802.11b Ethernet connections.

[1] Note that all reconfigurations of an ADE system can be accomplished dynamically without having to recompile any JAVA code.

4.1 The Robotic Architecture and its Implementation

We used a schema-based robotic architecture (Arkin, 1989) for the task, where sensory information coming from the camera and the sonar sensors is processed in "perceptual schemas": whenever a round orange color blob is detected in the image, a perceptual "ball schema" representing the orange ball is instantiated in the APOC server. The perceptual schema contains information about the centroid and bounding box of the ball. Similarly, whenever the distance reading from a sonar sensor is below a preset threshold θ, an "obstacle schema" is instantiated representing an obstruction of some kind at the sensed distance in the direction of the sensor. Perceptual information is then sent to motor schemas (again implemented as APOC components) via APOC A-links, which for each perceptual schema produce a directional vector: the vector either points in the direction of the perceived object (if it is "attractive" as in the case of the orange ball) or away from it (if it is "repulsive" as in the case of obstacles). The outputs of all motor schemas are then combined in a summation component (using a weighted vector sum), converted to motor commands, and sent to the motors.

The ADE implementation of the architecture (depicted in Figure 2) uses an ADE component to represent the robot together with all of its devices ("ade.units.PioneerServer"). An instance of the sonar perceptual schema component ("ade.units.SonarPerceptualSchema") receives the state of the robot's sonar sensors via an A-link and instantiates new sonar perceptual schemas ("ade.units.SonarPerceptualSchema") and motor schemas ("ade.units.SonarMotorSchema") based on the readings as described above via C-links. For visual processing, the chain is a bit more complicated as observing the state of the robot's camera (i.e., the image) does not directly yield information about round colored objects. For this, a color blob and shape detection program is needed. In our setup, this program runs as a separate ADE utility server that on startup connects to the specified ADE robot server, from which it obtains images as soon as they are available. For each image, the server (based on its configuration) computes color blob and shape information, which it subsequently makes available to other agents in the ADE system. We use an architectural representation of the blob server ("ade.units.BlobServer") to obtain blob information, which is connected via an A-link to an instance of a perceptual ball schema component ("ade.units.PerceptualBallSchema"), which will instantiate more ball schema components and motor schema ("ade.units.MotorSchema") depending on the number of detected balls and connect them via A-links analogous to the sonar schemas. All motor schemas are connected to the summation component ("ade.units.SummationUnit")

via A-links, which in turn is connected to the robot representation ("ade.unit.PioneerServer").

4.2 ADE Configuration

To demonstrate ADE's platform-independence and dynamic configurability, we will use two configurations of the ADE system: in the first configuration (see the left part of Figure 3), both robots each run three servers on the on-board computer: the APOC server running their architecture, the blob utility server performing blob and shape detection, and the robot server providing access to the robot's device. The ADE registry and the ADE GUI are run off-board; the registry on a Sparc Ultra V workstation running SOLARIS, and the GUI on a LINUX Desktop PC. The robots are connected to the registry via their wireless ethernet connection.

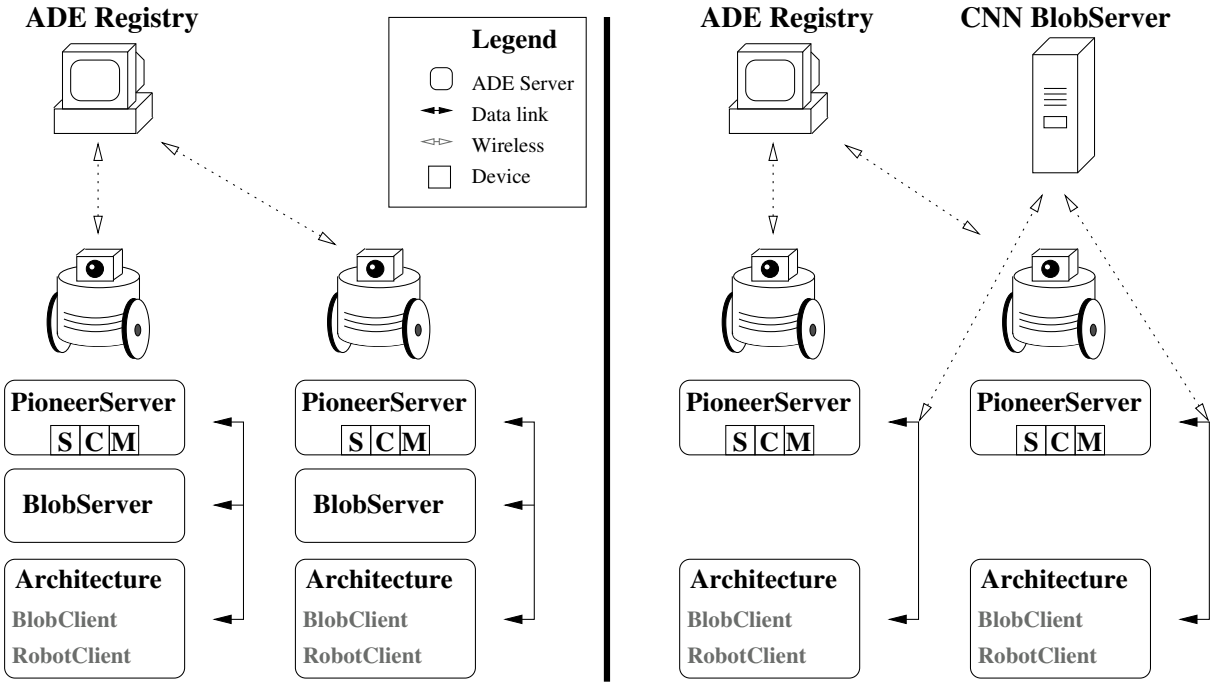

Figure 3: The ADE setup for the two configurations of the ball following task with on-board blobservers (left) and with shared CNN blob server (right). Each robot server comprises three devices: sonars (S), camera (C), and motors (M). For each host indicated by a box the host name, the operating system, as well as the ADE agents running on it are indicated. Solid lines indicate 100 Mbit ethernet connections, dashed lines indicated wireless 5 Mbit ethernet connections, and arrows indicate directional information flow.

For the second configuration we employ special vision processing hardware to perform fast blob and shape detection: a CNN-UM ACE4K chip (Chua & Yang, 1988; Chua & Roska, 2002). The CNN-UM is an analog computer that is particularly well suited to perform operations on pixel images at very high frame rates. In this

case, a blob detection algorithm has been implemented whose update rate is at least as high as that of the employed frame grabbers of 30 Hz. The CNN board (with the CNN chip) is part of a WINDOWS XP PC, which runs an ADE utility server that communicates with the CNN board via local socket connections. Effectively, this CNN blob server wraps the functionality, protocol, and runtime environment of the CNN board and provides the same interface as the ADE blob server to the rest of the ADE system. Hence, instead of using local blob servers, both robots can share the CNN blob server. Upon startup, the CNN blob server is configured to connect to each of the two robot servers (running on the robots) and obtain compressed images from them as soon as they become available. Each image is sent across the wireless connection from each robot to the CNN server as a compressed 160 x 120 pixel JPEG image to increase throughput. The server keeps track of the origin of the images and stores the detected blobs accordingly. The perceptual ball schema components of each robot then observe the state of the CNN blob server, which behind the scenes relays the right blob information to the right component based on the unique ID of the client representation (part of each blob server component).

4.3 ADE Runtime Inspection and Monitoring

Figure 2 depicts the graphical representation of the above described architecture in "Edit Mode", with the type diagram on the left and the initial state of the instantiated architecture on the right. Figure 4 shows the same architecture in "Run Mode" together with some of the available runtime inspection and monitoring tools, in particular, tools that provide the status of the robotic devices. For example, the graphical representation that can be obtained by double-clicking on an instance of the "ade.units.PioneerServer" component (in the virtual machine view) displays the basic information about the component (e.g., its process state, instantiation number, activation value, priority, and some additional text information) and furthermore provides quick access to the robot's sonar, bumper, motor, and camera devices (as displayed here), all of which can be obtained by clicking on the buttons in the panel. The circles in the sonar ring indicate then the current distances reading obtained from each sonar sensor. The circles in the bumper window indicate the status of the bumper sensor (currently, no bumper is pressed). The motor panel displays the current wheel velocities (currently the robot is not moving). Finally, the camera panel displays the status of the robot's camera device (top right image) as reported from the pioneer server as well as that of two other servers (if they are running): the center image displays the color blobs detected by the blob server

(in this case, one blob was detected, which is enclosed by a bounding box), while the bottom image displays any motion blobs detected in the image (in this case, there is no motion blob server running, so the image is the same as the camera image). Note that the camera panel allows for easy recording of the camera images as well as the color and motion blob images to separate files, which can be subsequently used for debugging or for demonstration videos.

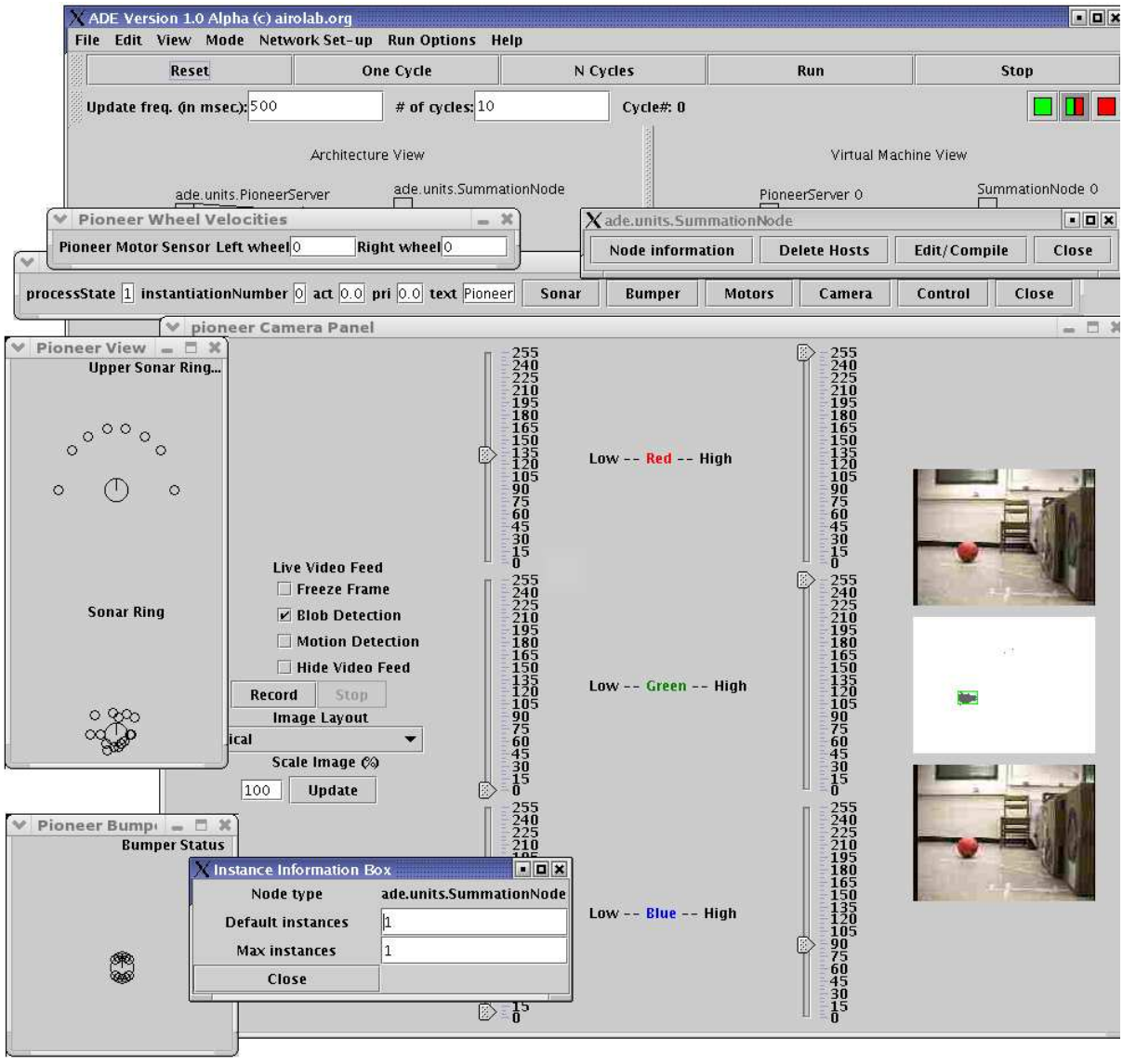

Figure 4: The robotic architecture in "Run Mode", showing several of the panels that provide information about the state of different parts of the ADE system (e.g., the camera panel showing the current image received by the robot's camera and the result of detecting color blobs in blob server–see the text for more details.

Also shown in Figure 4 are panels that can be obtained by double-clicking on the type representation of the "ade.units.SummationNode" (in the architecture view), where additional instance information can be obtained (see the "Instance Information Box"). The type panel also provides access to the hosts for which the component is registered as well as an editor/compiler interface (no depicted for space reasons), in which the component's source code can be edited and compiled dynamically. This facilitates rapid prototyping within ADE. All panels are updated automatically in all running GUI servers whenever they are open. Many additional graphical tools are available for runtime inspection and monitoring (e.g., for information passed through any type of link, for the dynamic host configuration of the ADE system, for the status of ADE servers, etc.). Moreover, the user interface has been designed to allow for easy extension of the graphical displays associated with each component (e.g., for user-defined components that need to display component specific information). That way it is possible to display the state of all components in the system (regardless on which APOC server they reside on) in a unified way without having to worry about host configurations or remote access (we will briefly return to the ADE's support for extensions in the next section).

5 Discussion

The above example demonstrates several important features of ADE, among which are its platform independence, its ability to run and integrate other programs within the ADE environment, and its graphical tools for setup, inspection and monitoring of architectural components and component instances. In this section, we will briefly touch on some other important aspects of ADE, some of which give it flexibility beyond what other comparable toolkits for robotic architectures (such as Saphira/Aria, Player/Stage, Carmen, and others) have to offer. Specifically, we will discuss three issues: (1) real-time processing and performance, (2) fault detection, security, and failure recovery, and (3) usability and extendability of ADE, after which we will briefly compare ADE to other agent systems and toolkits.

5.1 Real-time Processing and Performance

ADE was implemented in JAVA to achieve platform independence and allow for easy interoperability and data exchange between ADE agents by virtue of serializable JAVA objects via RMI. There are four potential performance

bottlenecks connected to the JAVA implementation of ADE. The first has to do with the overall JAVA performance (e.g., compared to other programming language), the second is connected to the performance of JAVA's RMI subsystem, the third is a result of ADE's implementation of agents as JAVA threads, and the fourth is a general issue about real-time processing in JAVA.

Regarding the first, it is a well-known fact that JAVA, even with optimized just-in-time compilation, does typically not reach the performance of efficiently compiled C++ or C code. Especially for time-critical applications or computationally restricted environments (such as embedded controllers) this might cause a problem. Fortunately, the APOC architecture specification, which ADE implements, offers a simple, yet elegant solution: time critical code can be implemented "natively" on each target platform and run as an associated process in an APOC component agent, where the JAVA-based part merely serves the architectural control function of providing inputs to the native process and relaying outputs from it to other components in the architecture without performing any other computation. For example, we have implemented a quick image segmentation algorithm in C for LINUX, SOLARIS and WINDOWS operating systems, which is "wrapped" by an ADE component. The ADE component, which runs as a thread in the JAVA virtual machine, determines the current operating system on instantiation and automatically loads the right precompiled executable for the platform (i.e., the code for the "associated process" of the component), which is then executed as a native operating system process.

A second potential performance bottleneck is imposed by JAVA's RMI system, which is generally much slower than comparable implementations that build directly on socket connections between two JAVA virtual machines. Especially for local method calls (i.e., for calls within the same virtual machine) the RMI overhead is significant. To reduce this overhead, ADE automatically checks for any RMI server-client connection whether the two computational objects are physically residing on different hosts. If both objects reside on the same host, ADE uses direct function calls (instead of RMI).[2]

The third potential performance problem is a result of ADE's implementation of agents: each ADE agent (including APOC components and APOC links) has at least one thread of control associated with it (ADE servers could have many simultaneously running threads associated with them depending on the number of allowed par-

[2]The assumption in the current version of ADE is that only one JAVA virtual machine, and consequently APOC virtual machine, can run on each host. For multi-processor environments, the short-cut mechanism has to be disabled to allow components to communicate across virtual machines on the same host.

allel connections). For large architectures this can create a significant bookkeeping overhead at the level of the operating system (both in terms of thread switching time as well as required virtual memory), especially if the architecture is not distributed, but runs completely within only one JAVA virtual machine. For example, an architecture with 50 APOC components fully connected via A-links requires more than 5000 threads, each with at least 128k of virtual stack memory (in SUN's JDK 1.4) associated with it. While such architectures are not feasible in all operating systems (e.g., 2.4 LINUX kernels severely limit the number of the currently existing threads) and all implementations of the JAVA virtual machine, mappings of JAVA threads to POSIX kernel threads (as implemented in SUN's JDK 1.4 for SOLARIS systems and 1.5 for LINUX with 2.6 kernels) allow for an efficient implementations of systems with large numbers of threads. In fact, practical experiments with large architectures (containing 5000 threads and more) on single CPU systems in our lab have shown that the performance loss of a standard ADE implementation, where all architectural components are implemented in separate threads and updated by virtue of OS thread switches, over an optimized loop implementation, where all agents are updated sequentially by virtue of explicit update function calls from the server, is only about 10 percent (of the duration of the overall loop). Moreover, the virtual memory overhead typically does not matter in practice as most threads in an ADE system are used to update links, which only perform simple computations without recursive function calls.

Finally, the fourth problem is a general concern related to real-time performance: JAVA by itself can at best operate in "soft real-time", i.e., guarantee performance to the point supported by the underlying operating system.[3] Hence, designers of robotic architectures have to ensure at the architecture level (as with other agent architecture toolkits) that their design is robust enough to be able to deal with temporal fluctuations of update times and cycles, timeouts on connections, etc.. However, ADE provides several mechanisms at the level of ADE agents to facilitate real-time operations. In addition to timed APOC component and link updates (which are based on system timers and will complete in a timely fashion if the computational resources are sufficient), ADE servers provide mechanisms to help maintain the communication throughput in distributed architectures and to detect and recover from failures, which we will discuss next.

[3] If run on a real-time OS, however, it is possible to guarantee hard real-time by virtue JAVA's real-time (see http://www.rtj.org/latest.pdf for details).

5.2 Security, Reliability, and Error Recovery

The ADE server model was designed for distributed dynamic multi-OS computing environments, where connections between any two hosts participating in the ADE system cannot assumed to be present throughout the lifetime of an architecture. Consequently, mechanisms were required for ADE servers to detect failures of connections and recover from failure. Moreover, since ADE was not assumed to be run in a dedicated computing environment, mechanisms to ensure the authenticity of service providers and service consumers needed to be integrated. Both requirements are met by ADE's "heartbeat mechanism", which is implemented by every ADE server: right after registration with an ADE registry, an ADE server starts sending status packets at regular intervals. Each status packet contains the server's ID, its unique credentials, which are created dynamically by the registry in response to each packet and need to be included in the followup heartbeat packet, and the status of its currently active connections. When a client requests a server connection, the registry forwards the request to the server passing along the client's ID and credentials. If the request is accepted, the server will return a remote reference to itself to the registry, which is then returned to client, and a separate peer-to-peer heartbeat connection is started between client and server (in addition to the one between server and registry). Note that while all ADE servers and clients containing references to servers must implement the heartbeat mechanisms, APOC component and link agents do not themselves need to implement it (unless they contain clients) as their containers, the APOC servers, are already connected via heartbeats. Hence, if an APOC server fails to send a heartbeat signal, all of its instantiated components and links are automatically assumed to be defunct and recovery steps can be taken by other APOC servers to recreate the dysfunctional parts of the architecture.[4]

Whenever a heartbeat times out (i.e., when a packet fails to arrive in time), a special function is called within the agents on each side of the connection to deal with the failure. While ADE enforces the implementation of error recovery methods (by virtue of JAVA's abstract methods), the details of the recovery procedures are left to the user. This flexibility is important as depending on the kind of agent, different steps are appropriate. For servers losing the connection to a registry, the first step typically is to try to connect to other registries. If none are present or reachable, the server will keep trying to reconnect to the host on which the registry was last running. Otherwise,

[4]Currently, the implemented mechanism only restarts the APOC server. A more detailed recovery would be possible, if the APOC server could save its state including the state of all of its component and link agents. Suitable efficient, yet computationally low-cost mechanisms are currently explored.

a new registry will take over the responsibility for managing the server. Registries losing server connections will either simply unregister the server or, if required to maintain the service (e.g., via the startup script), they will attempt to start the server on another host. Clients losing connections to servers can either discontinue the connection or request a new connection to another server of the same type (if available) through the registry. Servers losing connections to clients can either simply free the connection (informing the registry) or start a shutdown or reset process. This is, for example, crucial for servers running on robots connected through wireless ethernet, when the ethernet connection gets interrupted, yet the remote client controlling the robot had issued motor commands.

5.3 Using and Extending ADE

One of the main design requirements of ADE was to make the system as versatile as possible (despite its conceptual foundation on the APOC framework), allowing users to (1) reuse already developed components, (2) build architectures in their favorite programming language and design paradigm, and in general (3) customize and extend ADE according to their needs. The first two features are directly supported by APOC's notion of associated process. An already developed component can be used within ADE by running it as an associated process of a special APOC component, whose sole purpose is to start, interrupt, resume, and terminate the planner, supplying it with inputs during its operation and relaying its outputs to other parts of the architecture (cp. to 5.1). Depending on whether the component is written in JAVA or another programming language, it can run as a JAVA thread (either directly or via a JNI wrapper) or as a separate OS process (e.g., a planner written in C++). In the former case, ADE uses JAVA process synchronization mechanisms to control the process (to the extent possible), while in the latter it employs operating system primitives (on LINUX and SOLARIS, associated OS processes are not fully supported under WINDOWS) to suspend and resume non-JAVA-based associated processes. Hence, one possible application of ADE that does not require architecture development in APOC is to use its interprocess communication infrastructure to link programs on different OS platforms written in different programming languages, which otherwise would require additional low-level socket programming. Since all data exchange in ADE is based on serialized JAVA data types, data type conversions are only needed once between each involved programming language and JAVA.

Another application is to utilize ADE's robot server model to connect existing architecture, which again run

as associated processes in APOC components, to (remote) robots. In this case, some JAVA programming will be involved in designing the interfaces between the robot's devices and the employed architecture. Finally, all ADE mechanisms can be exploited for architectures that are translated into the APOC framework, which not only allows for a direct implementation, but also for an easy extension of the base APOC component class to support user-defined extensions, including fine-grained inspection and monitoring of different parts of the architecture.

Users can define new APOC components by extending existing components and modifying their link handling procedures (for incoming A-links and O-links, as well as outgoing A-links, P-links, and C-links) as well as their associated processes. To allow for graphical access to the components state, ADE will automatically add every instance variable of a derived APOC component class that is declared "ADEObservables" (i.e., entities that can be observed via O-links) to the graphical status display associated with the component, which can be displayed by double-clicking on the component's graphical representation in the user interface. ADE finds out about such variables using JAVA's reflection API, which allows for dynamic inspection of class properties that are unknown at compile time.[5]

Finally, it should be noted that due to its distributed nature and its robot control facilities, ADE itself could also be used to implement other frameworks intended for real-time operations of robotic agents (e.g., RCS-based systems (Albus, 1992) or RS-based system (Lyons & Arbib, 1989.)).

5.4 Related Work

Most the above discussed features distinguish ADE from other architecture development environments for robots (such as Aria/Saphira, Player/Stage, Carmen, and others), thus making ADE unique among robotic agent systems. Yet, several of the discussed features are present in other multi-agent systems, which are not targeted at robotic domains. We will briefly place ADE in the more general context of agent systems and architecture development toolkits.

DACAT (Barber & Lam, 2002, 2003) 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

[5] Several JAVA-specific APIs including the Reflection API are at the heart of the multi-user graphical user interface, which needs to link potentially remote objects to their local graphical representations, updating the display of their state without causing the a performance loss on the update cycle on the remote server–the implementation details of ADE's GUI model are beyond the scope of this paper.

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 [6] 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.

Many agent systems are concerned with mobile software agents, which can roam the Internet (e.g., AGENT-BASE[7],AGLETS[8], RETSINA, JADE, BDIM/TOMAS (Busetta & Kotagiri, 1998), ZEUS (Nwana, Ndumu, Lee, & Collins, 1997)). These systems 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 in APOC).

The AGENT FACTORY system (Collier & O'Hare, 1999; Collier, O'Hare, Lowen, & Rooney, 2003) is an environment for agents which use BDI architectures (Kinny, Georgeff, & Rao, 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 for 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.

[6] http://www.research.ibm.com/iagents/
[7] http://www.sics.se/isl/agentbase/
[8] http://aglets.sourceforge.net/

6 Conclusion

ADE is a nascent environment for distributed robotic architectures, which combines in a unique way the flexibility of the general architecture framework APOC and the communication infrastructure of an underlying multi-agent system, both of which form the basis for the implementation of distributed robotic architectures. In addition to the support for distributed robotic architectures, which can change dynamically, ADE provides mechanisms for error detection and recovery, and a customizable distributed multi-user graphical interface for designing, testing, and running robotic architectures. None of the current robotics systems integrates all of these features. We demonstrated ADE's utility as a design, implementation, testing, and run-time tool in a simple multi-robot setting, where two robots had to perform a ball finding and following task. Several much more complex architectures have been implemented or are currently under development for a variety of more complex tasks, especially in robotic settings.

Development on ADE is still ongoing.[9] While in its current form ADE is still far from being a full-fledged industrial application (contrary to some of the above mentioned agent systems such as JADE or RETSINA that have been used successfully outside academia), it has already been used successfully in several research projects. In summer of 2004, it was for the first time demonstrated publicly in several real-time applications at the AAAI 2004 intelligent systems demonstration. Feedback from these demonstrations will be incorporated into the first public release, which is planned for the end of 2004.

7 Acknowledgements

I would like to thank Virgil Andronache, John McRaven, and Gyorgy Cserey for their help with implementing and running the robotics example, and James Kramer for many discussions about the MAS underlying ADE and his help with the graphics in Figure 3.

References

Albus, J. S. (1992). A reference model architecture for intelligent systems design. In P. J. Antsaklis & K. M. Passino (Eds.), *An introduction to intelligent and autonomous control* (pp. 57–64). Boston, MA: Kluwer

[9] The first alpha version is available for download from our lab's web page at http://www.nd.edu/ airolab/software/

Academic Publishers.

Andronache, V., & Scheutz, M. (2002, April). Contention scheduling: A viable action-selection mechanism for robotics? In S. Conlon (Ed.), *Proceedings of the thirteenth midwest artificial intelligence and cognitive science conference, MAICS 2002* (pp. 122–129). Chicago, Illinois: AAAI Press.

Andronache, V., & Scheutz, M. (2003). APOC - a framework for complex agents. In *Proceedings of the AAAI spring symposium* (p. 18-25). AAAI Press.

Andronache, V., & Scheutz, M. (2004). Integration theory and practice: The agent architecture framework APOC and its development environment ADE. In *Proceedings of the AAMAS* (p. 1014-1021). ACM Press.

Andronache, V., & Scheutz, M. (2005). ADE - an architecture development environment for virtual and robotic agents. *International Journal of Artificial Intelligence Tools*, to appear.

Arkin, R. C. (1989). Motor schema-based mobile robot navigation. *International Journal of Robotic Research*, *8*(4), 92–112.

Arkin, R. C., & Balch, T. R. (1997). AuRA: principles and practice in review. *JETAI*, *9*(2-3), 175-189.

Barber, K. S., & Lam, D. N. (2002). Architecting agents using core competencies. In *Proceedings of the first international joint conference on autonomous agents and multiagent systems* (pp. 90–91). ACM Press.

Barber, K. S., & Lam, D. N. (2003). Specifying and analyzing agent architectures using the agent competency framework. In *Proceedings of the 15th international conference on software engineering and knowledge engineering (seke-2003)* (pp. 232–239).

Bellifemine, F., Poggi, A., & Rimassa, G. (1999, April). JADE - a FIPA-compliant agent framework. In *Proceedings of the 4th international conference and exhibition on the practical application of intelligent agents and multi-agents* (pp. 97–108). London.

Brooks, R. (1990). *The behavior language: User's guide* (Tech. Rep. No. AIM-1227). MIT.

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

Busetta, P., & Kotagiri, R. (1998). An architecture for mobile BDI agents. In *Applied computing 1998, mobile computing track, proceeding of the 1998 acm symposium on applied computing (sac'98)* (pp. 445–452).

Chua, L. O., & Roska, T. (2002). *Cellular neural networks and visual computing, foundations and applications*.

Cambridge University Press.

Chua, L. O., & Yang, L. (1988). Cellular neural networks: Theory. *IEEE Trans. on Circuits and Systems*, *35*, 1273–1290.

Collier, R., & O'Hare, G. (1999). Agent factory: A revised agent prototyping environment. In *Proceedings of the 10th aics conference, irish artificial intelligence and cognative science conference.*

Collier, R., O'Hare, G., Lowen, T., & Rooney, C. (2003). Beyond prototyping in the factory of the agents. In *Proceedings of the 3rd central and eastern european conference on multi-agent systems (ceemas'03)* (pp. 383–393).

Gerkey, B. (2003, June). The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th international conference on advanced robotics* (pp. 317–323). Coimbra, Portugal.

Kinny, D., Georgeff, M., & Rao, A. (1996). 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* (Vol. 1038 of Lecture Notes in Artificial Intelligence, p. 56-71). Springer-Verlag.

Konolige, K. (2002, April). *Saphira robot control architecture* (Tech. Rep.). Menlo Park, CA: SRI International.

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). SOAR: An architecture for general intelligence. *Artificial Intelligence*, *33*, 1–64.

Lyons, D. M., & Arbib, M. A. (1989., June). A formal model of computation for sensory-based robotics. *IEEE Transactions on Robotics and Automation*, *5*(3), 280–293.

Maes, P. (1989). How to do the right thing. *Connection Science Journal*, *1*, 291–323.

Maxwell, B. A., Meeden, L. A., Addo, N., Brown, L., Dickson, P., Ng, J., et al. (1999, July). Alfred: The robot waiter who remembers you. In *Proceedings of AAAI workshop on robotics.*

Minsky, M., & Papert, S. (1969). *Perceptrons: An introduction to computational geometry*. Cambridge, MA: MIT Press.

Nwana, H., Ndumu, D., Lee, L., & Collins, J. (1997). Zeus: A collaborative agents toolkit. In *Proceedings of the 2nd uk workshop on foundations of multi-agent systems* (pp. 45–52).

Rosenbloom, P., Laird, J., & Newell, A. (1993). *The soar papers: Readings on integrated intelligence*. Cambridge,

MA: MIT Press.

Scheutz, M. (2005). APOC - an architecture for the analysis and design of complex agents. In D. Davis (Ed.), *Visions of mind* (p. forthcoming). Idea Group Inc.

Scheutz, M., & Andronache, V. (2003). Growing agents - an investigation of architectural mechanisms for the specification of "developing" agent architectures. In R. Weber (Ed.), *Proceedings of the 16th international flairs conference.* AAAI Press.

Scheutz, M., & Andronache, V. (2004). Architectural mechanisms for dynamic changes of behavior selection strategies in behavior-based systems. *IEEE Transactions of System, Man, and Cybernetics Part B: Cybernetics*, *34*(6).

Scheutz, M., & Kramer, J. (under review). AgeS - a distributed agent system.

Scheutz, M., McRaven, J., & Cserey, G. (2004). Fast, reliable, adaptive, bimodal people tracking for indoor environments. In *IEEE/RSJ international conference on intelligent robots and systems (IROS).*

Sycara, K., et al. (2003). The RETSINA MAS infrastructure. *Autonomous Agents and Multi-Agent Systems*, *7*(1), 29–48.

Thrun, S., Fox, D., Burgard, W., & Dellaert, F. (2000). Robust monte carlo localization for mobile robots. *Artificial Intelligence*, *128*(2).

Tyrrell, T. (1993). *Computational mechanisms for action selection*. Unpublished doctoral dissertation, University of Edinburgh.