ADE - AN ARCHITECTURE DEVELOPMENT ENVIROMNET FOR VIRTUAL AND ROBOTIC AGENTS

VIRGIL ANDRONACHE
Artificial Intelligence and Robotics Laborator, University of Notre Dame
Notre Dame, Indiana 46556, USA
vandrona@nd.edu

MATTHIAS SCHEUTZ
Artificial Intelligence and Robotics Laborator, University of Notre Dame
Notre Dame, Indiana 46556, USA
mscheutz@nd.edu

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,[12] RETSINA,[37] AGENTBASE,[3] ZEUS[29]), or the design of agent architectures for single agents.[24,36] 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.

Multi-agent systems typically provide the distributed infrastructure that allows agents to reside on different hosting computers and/or move from host to host in a way that is transparent to the user. Furthermore, some of the toolkits also support distributed agents (i.e., agents residing on multiple hosts at the same time[37]). These systems are typically implemented as middleware that provides APIs for the agent designer. Yet, because these systems are intended as a framework in which to develop the multiagent system they typically do not provide tools required for the development of the architecture of an agent.

Single-agent systems, on the other, focus on support for the development of the agent's architecture, typically by providing libraries for common agent functionality (e.g., condition-action rule interpreters[36] or basic components of a particular architecture such as the BDI architecture[17]). They may also provide additional functionality for running multiple virtual agents (as in SIM-Agent[36] or Swarm[27]) or for operating robots (e.g., Saphira[24]). These systems are usually either designed for virtual agents or for robotic agents, and in the latter case typically only for single agents. Furthermore, they do not provide tools to implement and run multiple, distributed agent architectures (as would be possible in a multi-agent system).

Most importantly for our purposes, most architecture development toolkits either provide libraries that designers of agent architectures need to integrate into their code or they are based on a particular architecture design. While the former setup is very flexible and allows for a variety of architecture types to be specified, the agent architecture designer will have to acquire detailed knowledge of the libraries and the underlying assumptions to be able to use the libraries effectively. Furthermore, a significant amount of programming is required to design even simple agents. Although the latter alleviates this problem by providing a specific architecture paradigm, in which new architectures can be designed with more ease, it makes it impossible to implement other architecture types that are not based on the given paradigm.

We believe that versatile agent architecture design tools should be open with respect to the employed architecture paradigm, and should furthermore allow for a design environment for architectures that is appealing to users and easy to use, rather than requiring designers to understand (possibly very complicated) library calls. Furthermore, tools should also allow for distributing the architecture over multiple hosts in a user-transparent way to make it possible to run complex, computationally demanding architectures in parallel, thus reducing the overall computation time. Finally, tools should allow for the design of virtual and robotic agents alike, as well as single and multi-agent systems, where the same architecture could control a robot or a virtual agent in a multi-agent simulation environment without having to restructure the architecture or recompile the control code.

Based on these desiderata, we propose a novel tool, called ADE, which meets the above requirements. ADE is an integrated architecture development tool for both virtual and robotic single and multi-agent systems. It provides a fully graphical, distributed development environment that allows for interactive design of possibly distributed single and multi-agent architectures.

In the following, we first give a brief overview of the ADE toolkit and place ADE in the context of other single-agent and multi-agent development tools and frameworks. We then present details about the *agent architecture framework underlying* ADE, which is at the heart of ADE's flexibility, followed by a description of the user interface and the supporting environment. We also provide two extensive, practical examples of agent design in ADE, one for a group of virtual agents and one for a robot, which are intended to demonstrate the utility, flexibility, and user-friendliness of the tool. We then conclude with a summary of the ADE features and an outlook at possible future developments that are currently in the planning.

2. Background

We start with a brief overview of the main characteristics of ADE, which we then compare to the features of several other agent architecture environments.

2.1. *The Basic Characteristics of* ADE

ADE stands for "APOC Development Environment", where APOC is a general, universal agent architecture framework,[7,33,34] 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. *

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[28]) and as complex as a full-fledged *condition-action rule interpreter* (e.g., SOAR[25,30]). The implementation of the APOC framework is at the heart of ADE in order to guarantee the independence of ADE from specific architecture paradigms and to allow architecture designers to work in their preferred paradigm.

Computational components of an agent's architecture can be created and destroyed during the life-time of an agent by other architectural components. Hence, agent developers can specify agents that use minimal resources for task completion, develop specialized subsystems, or are adaptive (e.g., become more deliberative over their life-span).

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

*APOC components are based on the "behavior nodes" described by Scheutz.[31]

to run different algorithms in different parts of the architecture and to change them over the lifetime of the agent. Hence, as a design tool, ADE allows for the definition of a large variety of different mechanisms even within the same agent architecture. Concepts from one formalism can often be transferred to another by virtue of a unified representation in (e.g., semantic nets, neural nets, conditions-action rules, or conceptual hierarchies can all be defined in a unified way with ADE). It is thus possible to express and study different designs of various mechanisms within the ADE framework (e.g., how to do behavior arbitration, or how to actively manage finite resources at the architecture level). 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.[33]

In addition to expressing and implementing existing architectures, ADE can be also used to define new concepts and implement new architectures. For example, we recently introduced "dynamic architectures" that are capable of modifying themselves over time by altering their own description (e.g., as part of a learning process) and demonstrated their implementation in ADE.[34]

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 9) or remote agents in a simulated environment in order to control them (e.g., a single distributed architecture could control multiple agents or devices). 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 (as real parallelism is required for fast, real-time processing in complex agents).

A graphical user interface allows for easy access to different parts of the architecture to provide a closer level of control for the developer over the structure of the agent as well as making this control as direct as possible. APOC components can be "dropped into" a workspace where they are depicted as nodes in a graph (whose edges are links, as described above). Relevant information about each computational component can be viewed and modified by clicking on its graphical representation. The presence of four types of links indicates that different modalities of communication take place between components, which can be seen in the graphical representation of the architecture. The actual data carried through each link can also be displayed in the graphical interface. Most importantly, multiple designers can work with ADE at the same time, allowing for a distributed, collaborative design, test, and run-time environment.

It is also worth mentioning that ADE is not limited to architectures of single agents. Rather, it is possible to define multi-agent systems at the level of individual perceptions and actions in terms of the ADE tool: each individual agent is modelled by a subset of APOC components, which in turn have O-links (modelling the perceptions of the agent) and A-links (modelling the actions of the agent). To model procreation in biological systems, C-links can be used to allow agents to instantiate copies of themselves. In general, ADE could be used to model both centralized and distributed control systems, providing additional evidence to the flexibility of the tool and its potential usefulness.[33]

In sum, ADE combines a broad range of existing and new features, from single-agent toolkits to multi-agent frameworks (to be discussed in more detail in later sections describing the user-interface and supporting environment for the framework), all of which are aimed at simplifying the process of architecture development for virtual and robotic agents in single and multi-agent environments.

2.2. *Other Agent Tools Compared to* ADE

DACAT[10,11] is an architecture design tool which provides the user with a set of competencies from which the user can choose the ones relevant to her agent design. The set of competencies can be augmented by the user to provide arbitrary functionality. The user can then group the competencies and resources into modules and produce a description of the agent architecture. 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[1] 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[14]).

Many agent systems are concerned with mobile software agents, which can roam the internet. These systems (AgentBase,[3] AdventNet Agent Toolkit,[2] Aglets,[4] BDIM/TOMAS,[16] Retsina,[37] and others) focus on supporting efficient and secure communication among agents as well as improving their mobility. However, they do not provide support for the distribution of the components of an architecture over multiple computers unless these components are implemented as agents themselves (i.e., each component as a "complete agent architecture", which not only complicates the architectural design, but can also lead to reduced efficiency due to communication overhead). Furthermore, 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[17,18] is an environment for agents which use BDI architectures.[22] 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. Hence, like ADE it does not require or impose a particular architecture paradigm. Rather, 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.

Saphira[23,24] is a hierarchical robot control system designed for the ActivMedia Amigobot and Pioneer Operating System. Functionalities provided by the Saphira system are access to robot sensors and predefined routines for tasks such as gradient-based navigation. ADE provides facilities for accessing a multitude of robots, with code already available for the Pioneer and Peoplebot robots. ADE also allows for the definition of reusable routine. Various navigation and visual processing routines have been implemented and are available with the ADE distribution.

The Player/Stage[19,20] system uses devices to represent various sensors and effectors. A robotic agent is a collection of devices, although Player/Stage does not require the sensors and effectors to be located on a single robot. Devices operate independently of one another and communicate through sockets, making Player/Stage a distributed environment. ADE uses a similar approach: a JAVA interface is defined for each sensor type. A specific robot is accessed by defining a class which implements the interfaces corresponding to that robot's sensors and effectors. Through the use of the JAVA remote method invocation (RMI) mechanism, ADE also provides a distributed environment.

The Java Agent Development (JADE) framework (JADE)[12,13] provides similar distributed-environment functionality to RETSINA.[37] However, its internal organization deserves a closer look. JADE uses agent containers, which use RMI for communication. These containers are used to create, start, and suspend agents. ADE uses a similar approach, through the use of APOC servers, as described in Section 5.

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. In the following, we will first present details about the implementation of the APOC framework, which gives ADE agents their architectural flexibility, and then describe the graphical user interface and the supporting environment.

3. ADE Building Blocks I: The APOC Framework

The APOC agent architecture framework consists of *components* and *links* among them. In this section, we describe the basic functionality of components and links in APOC. In particular, we discuss how components can be extended and connected to form an agent architecture (more details about APOC can be found in.[7, 32–34]

3.1. APOC *Components*

APOC components are very general autonomous control units that are capable of (1) updating their own state, (2) influencing each other, and (3) controlling an associated physical or computational process. The process associated with an APOC component can be used for such functions as sending motor commands to a robot or running a parsing algorithm in a virtual agent that checks web pages for particular content. For physical processes, an APOC component can be viewed as a *controller* (in the sense of control theory) and for computational processes as a *process manager* (in the sense of operating systems).

APOC components have input and output ports, which can be connected to output and input ports of other components, respectively, via APOC links. A set of connected components, then, forms a network of components, i.e., an *architecture.*

Once components are running (i.e., they are instantiated in a virtual machine), they are self-sufficient entities that behave according to their specification as determined by their *initial state*, their *associated process*, and their *update function.* The state of an APOC component can be defined as

$$\langle act, pri, pro, inst, F, in, out\rangle$$

where *act* is the activation level, *pri* is a pair containing the current and the maximum priority level, *pro* is a triple containing the process state and the process associated with node as well as the operation performed on that process, *inst* is a pair containing the current instantiation number and the maximum number of instances of a node of that type, *F* is the update function, *in* and *out* are respectively sets of input and output links of the node.

The basic functionality of APOC components is implemented in the *APOCNode* class, from which all user-defined components are derived. A user-defined subclass can redefine several predefined functions to deal with the states of links and the state of the associated process. While the state transition of the associated process is determined by its current state–RUNNING, INTERRUPTED, or STOPPED–and the incoming information on the P-links according to the APOC specification (e.g., a running process that receives the SUSPEND signal will be interrupted), user-defined classes can redefine the methods that will be called after each state transition is complete. Furthermore, methods for processing incoming A-links and O-links, as well as outgoing A-links, P-links, and C-links can be defined by subclasses of *APOCNode*. Incoming C-links and outgoing O-links are again processed automatically according to the APOC specification.

The basic template of a user-defined class derived from *APOCNode* is given below:

```
import apoc.APOCNode
import apoc.ActivationLinkInterface;
import apoc.PriorityLinkInterface;
import apoc.ObserverLinkInterface;
import apoc.ComponentLinkInterface;

public class UserNode extends APOCNode
                         implements Serializable, Runnable, Remote {

     /* functions to control the associated process */
     public void processNoop() { ... }
     public void processResume() { ... }
     public void processStart() { ... }
     public void processSuspend() { ... }
     public void processReset() { ... }

     /* functions to process information on incoming links */
     public void inputProcessingA() { ... }
     public void inputProcessingO() { ... }

     /* functions to send information to outgoing links */
     public void outputProcessingA() { ... }
     public void outputProcessingP() { ... }
     public void outputProcessingC() { ... }

     /* additional update of the state of the node */
     public void selfUpdate() { ... }

     /* list of graphical entities to be displayed for this component */
     public Vector extendComponent() throws RemoteException { ... }
}
```

Each node also provides the user with the function *extendComponent*, which specifies the data of the component that can be observed, displayed, and subsequently modified through the ADE graphical interface.

3.2. APOC *Links*

APOC components are connected to other components through one of four APOC links: *activation link* (A-link), *priority link* (P-link), *observer link* (O-link), and *component link* (C-link).

Since each component needs to keep track of its incoming and outgoing links, ADE provides eight *JAVA vectors* to that end (four for each link type): *inActivations*, *outActivations*, *inPriorities*, *outPriorities*, *inObservers*, *outObservers*, *inComponents*, and *outComponents*. Each vector has as elements other vectors, which in turn contain individual links.

In the following, we briefly describe the functionality of each link type.

3.2.1. *A-links*

Activation links are the most general means by which nodes can exchange information. The state of an A-link is given by the tuple

$$\langle S, R, act, F, t \rangle$$

where S is the node providing the data to the link, R is the node receiving the output, act the data transmitted through the link, F the operation performed on that data, and t is the time it takes for data to traverse the link. The purpose of an A-link is to connect two APOC nodes and serve as a transducer.

A-links can be used in a variety of different ways. In the simplest case, they function as mere connections between input and output ports of APOC nodes (i.e., inputs to links are identical to their outputs). Furthermore, an A-link can be used transform the input, e.g., in case of numerical values it could “scale” the input by a particular factor (analogous to the “weights” on connections in neural networks).

A-links provide two functions: *setData* to place data on an element of the *outActivations* Vector, and *getData* to retrieve data from a link. Data passed along links must implement the *APOCObservable* interface.

3.2.2. *P-links*

Priority links are intended to explicate the capacity of components to control other components’ associated processes. They are the only means by which APOC nodes can control processes of other nodes (since no link has a process associated with it and nodes can only be connected to other nodes via links, APOC nodes could not control any process otherwise). The state of a P-link is given by the tuple

$$\langle S, R, pri, op, t \rangle$$

where S is the node attempting to take control of the process associated with R, pri is the priority of S, op the operation that S attempts to effect on the process associated with R, and t is the time it takes for data to traverse the link. ADE provides five operations to be performed on processes, using the constants: *PriorityLink*.RESET, *PriorityLink*.START, *PriorityLink*.SUSPEND, *PriorityLink*.RESUME, and *PriorityLink*.NO_OP.

A P-link effectively passes the process control request of an ADE node on to the node it is connected to through the P-link. Priorities can be used to implement all kinds of control mechanisms, in particular, hierarchical preemptive process control. ADE handles priority signals by computing the maximum of the priorities received on incoming P-links. If the maximum priority is greater than the priority of the component itself, then the request received from the node of greatest priority is honored. If there is a tie in nodes of maximum priority and their requests conflict, *PriorityLink*.NO_OP is performed.

In embodied agents, such as robots, P-links could be used to implement emergency behaviors: the node with the associated emergency process would have the highest priority in the network and be connected to all the other nodes controlling the agents behavior, which it could suppress in case of emergency (thus implementing a "global alarm mechanism" as described by Sloman[35]).

Analogous to A-links, P-links provide *setData* and *getData* functions to place on and retrieve data from a P-link.

3.2.3. *The O-link*

Observer links are intended to allow components to observe other components' inner states without affecting them. The state of an O-link is given by the tuple

$$\langle S, R, D, t \rangle$$

where S is the node observed by R, D is the information passed from S to R, and t is the time it takes for data to traverse the link.

The O-link operates independently of the update function in components; it retrieves information by observing a component, not by having data placed on the link. Thus, the only operation available on the O-link is the *getData*.

3.2.4. *The C-link*

Component links are used to instantiate and remove instances of APOC nodes at run-time (they are the only type of component that can instantiate or terminate an APOC node), and are themselves only instantiated by APOC nodes. The state of a C-link is given by the tuple

$$\langle S, R, D, L, t \rangle$$

where R is the node instantiated by S, D is information about the links which can be instantiated through this C-link, L is the set of links already instantiated through the C-link, and t is the time between the activation of the link and the creation of the component.

A C-link contains information about the type of component it can instantiate and the kinds of links it can create to that component (in addition to itself). A component can create and trigger the function of a C-link by issuing a *createCLink* command. This command creates a new C-link, triggers the link to create a new component and automatically adds the newly created link to the *outComponents* vector of the component.

Two operations are available on a C-link: activation and deactivation. On activation it can create a component and/or create a link to the component created by a previous activation command to that link. To separate functionality, several functions are provided for C-link activation and deactivation, some of which are illustrated below:

```
((ComponentLinkInterface)outcomponents.elementAt(0)).activateNode();

((ComponentLinkInterface)outcomponents.elementAt(0)).
     activateLink(ComponentLink.PLink);

((ComponentLinkInterface)outcomponents.elementAt(0)).activateAll();
```

The first example creates a component and connects the controlling component to the new one via a C-link. The second example presupposes an existing component created through the C-link. Thus the C-link simply creates a P-link to the created component. If such a component does not exist, the function call fails. Finally, the third example creates a new component and all the links defined as being available for instantiation through the C-link.

Since C-links are the only mechanism through which architectural changes are effected, they play an important part in resource allocation and arbitration. If A-links or P-links are used in conjunction with a C-link, activation and priority based mechanisms can be used to trigger the action of the newly instantiated component.

To "undo" the action of a C-link, several options are again available. *deactivateNode* deletes the component instantiated through the C-link if the only incoming link remaining to that component is the current C-link, otherwise the operation fails. The second deactivation example deletes a link created through the C-link, while the final example deletes all components–components and links instantiated through the C-link whose *deactivateAll* method is called.

```
((ComponentLinkInterface)outcomponents.elementAt(0)).deactivateNode();

((ComponentLinkInterface)outcomponents.elementAt(0)).
     deactivateLink(ComponentLink.PLink);

((ComponentLinkInterface)outcomponents.elementAt(0)).deactivateAll();
```

4. ADE Building Blocks II: 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.

To reflect this conceptual division into architecture layout and running virtual machine, the workspace of the graphical user interface is divided into two subspaces, which can be manipulated and viewed independently: the left half shows the architecture layout of the system, while the right half shows the run-time virtual machine. Figure 1 shows a screen shot of the basic run-time environment.

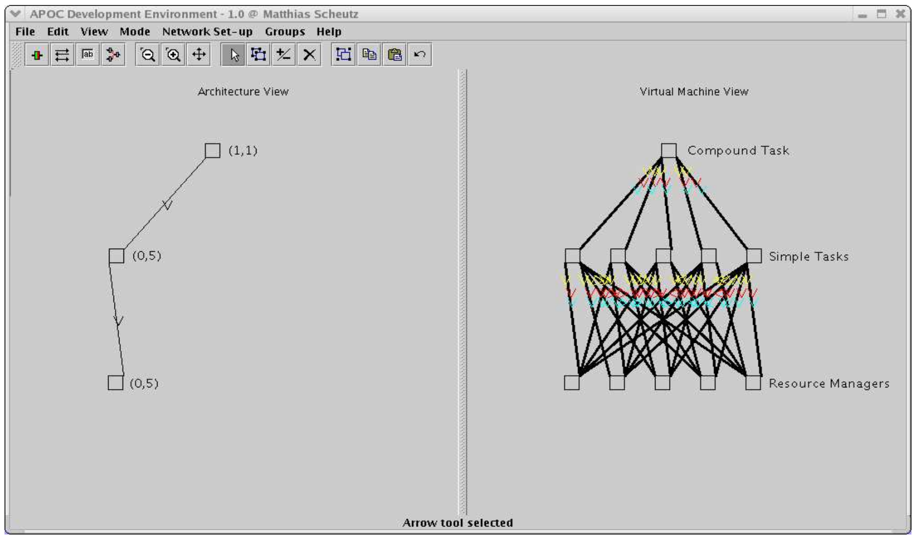


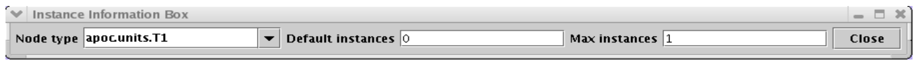
Fig. 1. ADE Interface

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

4.1. *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 components with a "component tool" and display their information by double-clicking on them. For each component, ADE shows at least three parameters which need to be specified by the user: the type that the component represents, the number of components of that type present when the architecture is first instantiated, and the maximum number of components of that type which can be present simultaneously in the running virtual machine (see Figure 2). For user-defined components, which are extensions of the basic component (as defined by APOC), ADE can display additional information about the component (e.g., the image taken by a robot's camera, see section 9).

Links in ADE are created through a link creation tool that allows users to specify links of each of the four available types simultaneously. Each link can be configured individually as specified below. In the graphical interface, edges indicate (by the direction of the arrow) the direction of the links in the architecture, which is the same as the direction of information flow (except for O-link edges, where the information flow is contrary to the direction of the arrow). By clicking on the arrow, information can be obtained about the types of links (e.g., A-link) that can be instantiated between the connecting components as well as the link parameters (e.g., delay, operator type for an A-link, etc.) in the run-time virtual machine.


Instance Information Box

Node type apoc.units.T1 Default instances 0 Max instances 1

Fig. 2. APOC Component Specification Prompt. The three fields indicate the component class, the number of components present in the initial virtual machine and the maximum number of components simultaneously present in the virtual machine

All APOC links share the characteristics below:

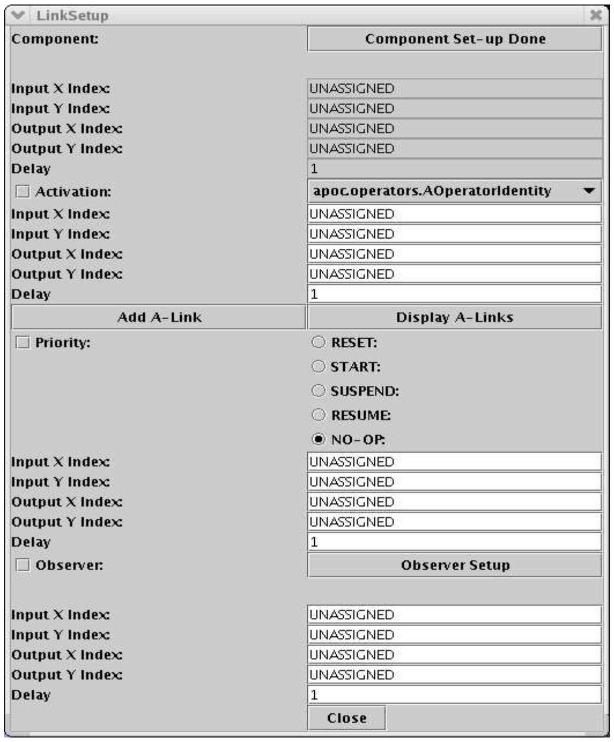
- *A time delay.* The delay slot is always available for editing and it is used to specify the number of update cycles that are performed on a link between the time a piece of information enters the link to the time it is available at the other end. The default duration of an update cycle is 100ms and can be modified by the user. All links created simultaneously have the same delay.
- *Input and output ports.* Port specifications are provided through the X and Y input and output slots at the bottom of the panel. The X value specifies the set of inputs (outputs) that the link connects to. An $UNASSIGNED$ Y value indicates that the link can connect to any port within the respective set, while other values uniquely identify the port to which the link is to be connected.

Each link type also requires a specific setup, which is activated once the particular link type is selected.

The A-link definition requires the selection (and, perhaps, set-up) of an operator, which will act on the information passed through the link as specified in ADE. The operator can be specified by choosing from a list of available operators, both standard and user-defined. By default, the "identity operation" is selected.

P-link definition consists of choosing a default signal. The default signal is transmitted along the link upon link activation (if the controlling component does not specify its own signal). Typically, this is set to "no operation".

The O-link allows users to specify what elements of a component should be sent across an O-link observing that component. The elements which are eligible



LinkSetup

Component:	Component Set-up Done
Input X Index:	UNASSIGNED
Input Y Index:	UNASSIGNED
Output X Index:	UNASSIGNED
Output Y Index:	UNASSIGNED
Delay	1
☐ Activation:	apoc.operators.AOperatorIdentity
Input X Index:	UNASSIGNED
Input Y Index:	UNASSIGNED
Output X Index:	UNASSIGNED
Output Y Index:	UNASSIGNED
Delay	1
Add A-Link	Display A-Links
☐ Priority:	RESET: START: SUSPEND: RESUME: [✓ NO-OP:]
Input X Index:	UNASSIGNED
Input Y Index:	UNASSIGNED
Output X Index:	UNASSIGNED
Output Y Index:	UNASSIGNED
Delay	1
☐ Observer:	Observer Setup
Input X Index:	UNASSIGNED
Input Y Index:	UNASSIGNED
Output X Index:	UNASSIGNED
Output Y Index:	UNASSIGNED
Delay	1
	Close

Fig. 3. ADE Link Specification Prompt

for observation in ADE have to be declared as being instances of a particular ADE-defined observable type by extending the *APOCObservable* interface. A menu presents the user with the available options. By declaring a variable as being of the observable type, it will automatically be included in the menu.

The C-link, in addition to delay and input and output port specifications, contains the following data:

- an identifier for the type of component which can be instantiated through the link
- one or more definitions for the types of links which can be instantiated through the C-link

4.2. *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. The arrows are color coded to differentiate among the four link types: A-links are represented by a blue arrow, P-links by a cyan one, O-links by red, and C-links by yellow.

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.

In both cases, the idea is to permit users to modify the instantiated architecture in the running virtual machine only in line with the architecture layout to prevent inconsistencies. If new links need to be added that are not part of the architecture specification, then the architecture needs to be modified in the architecture layout space first, before it can be reinstantiated.

4.3. *Other Functionality*

In addition to the architecture construction facilities described above, ADE also provides tools which aid in visualizing and understanding the relationship between various parts of the architecture. Thus, an "architecture analysis mechanism" which aids in understanding structures that form as a result of interaction between the agent and its environment can be accessed by clicking on the "Abstraction" button in Figure 1 (for details about this function see[33]). Another tool provided through the "Graph" button allows for the visualization of variations of observable variables within a component over time (e.g., activation level and priority, see Section 7).

A "grouping " mechanism is also supported by ADE. Components can be selected and grouped together. A group can then be collapsed and represented as a single component in the architecture. Links drawn to and from the new component thereafter connect to all components in the group.

4.4. *Operating Modes*

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

4.4.1. *Edit 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.

4.4.2. *Synchronous Mode*

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. Specifically, the updates are performed asynchronously and upon completion a wait operation is performed on an external signal, which can be provided by the user or by the system, before the next update cycle is performed.

4.4.3. *Asynchronous Mode*

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[14, 15]) 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 operating system.[†]

5. ADE Building Blocks III: 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).

5.1. ***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.

In ADE, these transactions can be of the following types:

(1) an APOC servers requires another APOC server to instantiate a component of a given type (which resides on that server)[‡]

[†]On realtime operating systems, this update frequency can be guaranteed.
[‡]Components are typically instantiated in the APOC virtual machine that keeps their JAVA class

(2) a graphical user interface requires a server whose information it needs to display
(3) a server requires a graphical user interface on which to display its information
(4) an instance of an APOC component which directly controls agent sensors and/or effectors requires an agent server on which to apply its function
(5) an instance of an APOC component which represents/uses the functionality of a utility server requires a utility server from which to fetch its data

In each of the above cases, 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).

5.2. *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 (hence, each ADE server has a *main* method that sets up the service). 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).

In the following sections we describe each of four server types–APOC server, GUI server, agent server, and utility server–and their functions within ADE.

5.2.1. APOC *and GUI Servers*

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 order to be able to notify the GUI servers whenever a new component is instantiated or an old one is deleted. Conversely, the GUI servers need to pass on user actions (such as adding a node) to APOC servers. Hence, each GUI server also maintain connections to all available APOC servers.

Upon start-up, an APOC server contacts the registry and requests connections to all GUI servers currently registered. Upon successful completion of the connection requests, direct two-way communication channels are established between an APOC server and each GUI server. This process is mirrored in the startup process of a GUI server, thus allowing new GUI or APOC servers to be added dynamically to an ADE system at runtime.

description. Multiple such descriptions in different virtual machines are possible, however, and allow for the implementation of load balancing mechanisms at the agent architecture level.

5.2.2. *Agent Servers*

Agent servers provide access to the body of a virtual or robotic agent by establishing connections to its sensors and effectors. They are independent computational resources and can, therefore, be started independently. After startup, they automatically connect to the registry announcing their service and then wait for clients to connect. Users can define their own agent servers by extending the *ADEServerImpl* class provided with the ADE framework:

```
public class UserServerImpl extends ADEServerImpl implements
     Serializable, ClientInterface {

     public UserServerImpl (String registryHost, int registryPort,
         String registryName, String myName) throws RemoteException {
          super(registryIP,registryPort,registryName,myName);
          ...
     }
     ...
}
```

It is worth noting that any server could be integrated into the ADE system as long as it contacts the registry initially and provides a remote service, called *requestConnection*, through which it can be contacted. By extending the *ADEServerImpl* class, however, the registry connection is handled by the ADE system automatically.

The default constructor of the *ADEServerImpl* class takes four parameters: the name of the computer on which the registry is running, the port on which the registry can be contacted (by default 1099, the JAVA RMI port), the name under which the ADE registry is known to the JAVA Naming service, and the name under which the agent server will register with the ADE registry.

5.2.3. *Utility Servers*

Utility servers provide a service which may be needed by one or more APOC components. Generally speaking these servers implement computationally expensive operations and are implemented as separate components of ADE as an additional aid to the distributed nature of the system. Utility servers follow the same startup process as agent servers, but may require an additional connections to agent or utility servers (established through the registry) in order to obtain the data they are supposed to process.

5.3. *Example: Generic* ADE *Set-up for a Robotic Agent*

To illustrate the different relationships among the four server types and the registry, we briefly sketch the generic setup for the control of a robotic agent in ADE.

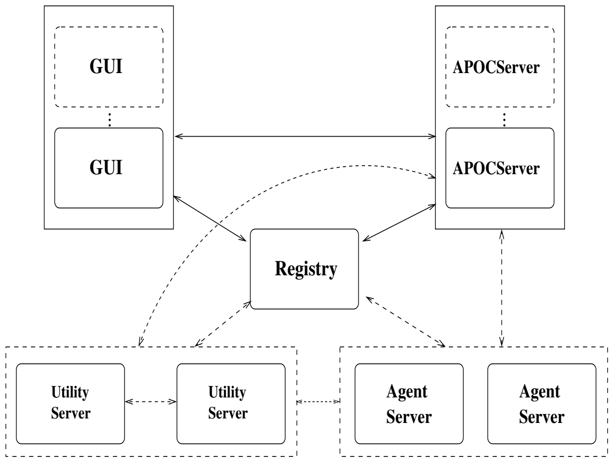


Fig. 4. The relationship among ADE components in a generic set-up for a robotic agent

Figure 4 shows the overall system configuration, where continuous lines represent components which are always present in an ADE environment, while dashed lines are components used only in particular instances (such as the specific robot environment). Thus, a minimal system consists of the registry, one APOC server and one GUI server. Note that the GUI server is not strictly necessary. It is typically used to load and start the agent architecture and it is possible to configure ADE in such a way that no GUI servers are needed for its operation.

Utility servers may be added if computationally intensive functionality (e.g., image processing) is required by the agent. Some of the utility servers may require other utility servers for their operation (e.g., a utility server performing 3D object recognition might require a utility server that provides basic image analysis services).

Additional agent servers may also be added to the system (before startup or at runtime), e.g., to provide a simulation environment for the robot or to provide a "simulated body model" for the robot if it is also part of a virtual environment (e.g., a combined real-virtual environment[9]).

Note this example is only one of several possible configurations for robotic setups. Different agent servers could be used, for example, for different sensors and effectors of the robot, or for different robots in a multi-robot setting. Depending on the task requirements, ADE can be dynamically adapted to work with any configuration of servers.

5.4. *Client-Server Communication*

Once a client-server connection has been established (e.g., between an APOC virtual machine and an agent server after mediation by the registry), the client is responsible for the maintenance of that connection. Maintenance for client-server connections consists of the client calling a server function (*updateConnection*) at regular time intervals to ensure that the connection is alive. The length of the interval is set up at connection time, but can be modified at any point by the client. This mechanism effectively ensures the integrity of the ADE system and allows it to react to possible failures. It also supports interactions in real-time domains (such as in the robot setup mentioned above).

5.5. ADE *Configuration File*

In order to simplify the startup of an ADE system, which may be quite complex and involve numerous different hosts, on which services need to be started, ADE uses a global configuration file. In a typical setting, this configuration file is run in one GUI server (which needs to be started manually), which then bootstraps the rest of the ADE system from a single host.[§] In the bootstrapping process, the GUI server reads the configuration file and uses a secure shell to open connections to each host in the "hosts list". Once a connection is established to a remote host, the "SSH_COMMAND" is executed on the remote host to start an APOC server.

The following parameters can be set in the configuration file:

(1) REGISTRY - a tuple containing the name of the machine on which the ADE registry is running and the name under which it is registered with the JAVA Naming service.
(2) SSH - the secure shell program used to start remote servers.
(3) SSH_COMMAND - the command to be executed on the remote host in order to start a remote server.
(4) HOSTS - the computers that are available to host services of the ADE system.
(5) COMPONENTDIRS - the directories containing APOC component definitions. In addition to the default components included in ADE, users can add their own component definitions, which can then be used within the whole ADE system.
(6) OPERATORDIRS - the directories containing operator definitions for A-links. In addition to the default operators included in ADE, users can add their own operators, which can then be used within the whole ADE system.
(7) INPUTFILE - an agent definition file to be loaded at start-time.

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

6. ADE as a Tool for Designing, Testing, and Running Agent Architectures

ADE serves at three-fold role in the development process of agent architectures: (1) as a design tool for developing agent architecture layouts and their implementa-

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

tions, (2) as a platform for run-time control of agents, and (3) as a tool for testing agent architectures. Although these three roles typically overlap in the practice of developing agent systems, we address each role separately to emphasize different characteristics of ADE.

6.1. ADE*: a Design Tool*

ADE is built on the premise that a graphical representation of an agent architecture that can be viewed at different levels of detail is crucial in the design process of agent architectures. The graphical interface allows users to add components and links, specify and modify their properties, and arrange them in their preferred layout. Since architecture descriptions are saved in XML format to a file, they can be also viewed and edited using standard XML authoring tools. Furthermore, they can be inserted into existing architectures, thus allowing for efficient reuse of common subarchitectures.

The tight integration of ADE with Sun's JAVA source development kit allows for the definition and compilation of new components within ADE, which are then immediately available for use in the architecture. Furthermore, by using JAVA as the implementing language, ADE is platform independent, i.e., it runs on any operating system for which JAVA virtual machines exist and thus, provides essentially the same environment on different machines. More importantly, once components are defined and compiled on a particular system, they are available on all systems with JAVA support. This is particularly useful for distributed design setups that involve multiple computers with different operating systems, all of which can seamlessly interact in ADE.

As mentioned in the Introduction, one of ADE's advantages over other agent development environments is that it is not based on a particular architecture paradigm, but rather that it is capable of implementing any agent architecture (e.g., cognitive architectures such as SOAR,[25, 30] ACT-R,[6] and others, as well as behavior-based architectures such as subsumption,[14] motor schemas,[8] situated automata,[21] etc.) in a unified way. Therefore, it can be used for the design and comparison of agent architectures. For example, the action selection mechanism in Maes' ANA architecture requires global control despite some claims that it uses only local mechanisms.[26] It is also possible to compare the tradeoffs of different architectures with respect to some particular task. For example, two different architectures implementing a "target-finding task" for a robot can be compared with respect to the number of components used or the total time required for achieving a goal.

ADE also allows for the display and design of agent architectures at different levels of abstraction, depending on the complexity of the update functions in employed APOC components and the topology of the connections among them. It is, for example, possible to use Boolean update functions in components to implement logic circuits (such as gates, inverters, flip-flops, etc.). A network of such components could then, for example, model the low-level architecture of an embedded processor or robot controller. By the same token, it is possible to use APOC components that implement very high-level functionality such as condition-action rule-interpreter, a planner, a reasoning engine, a search algorithm, etc. In that case, ADE will only be able to display a high-level view of the architecture, while the details of how the rule-interpreter, planner, etc. work are hidden in the associated process of the component implementing them.

It is also possible to use the associated process of an APOC component to run algorithms that are not defined and implemented within the ADE framework. Input to and output from these processes is effected through A-links within ADE via their input and output streams. Hence, other components can communicate with such "external processes" in a transparent way (i.e., without having to know that they are *external* to ADE). Again, the details of the implementation of such processes is necessarily hidden and cannot be visualized in ADE.

6.2. ADE *as Run-Time Tool*

A "running architecture" (i.e., the architecture instantiated in the APOC virtual machine) can be inspected and modified at any given time through the graphical user interface. Double-clicking on a component, for example, reveals information about the component (in particular, the information provided by the *extendComponent* function, which is user-definable in classes derived from *APOCNode*).

The example in Figure 5 illustrates the information which can be viewed for a component representing a robot body description. Clicking on any of the buttons brings up additional windows displaying internal information of the robot (e.g., clicking the "Camera" button will bring up a panel displaying the current image of the camera and the processed image, as seen in Figure 6).

Similarly, general information about links can be obtained by double-clicking on the link's arrow. Figure 7(a), for example, shows the information displayed by O-links. Clicking on the "Link Info" button, results in a display of more detailed information about the actual data passed through the links. In Figure 7(b), the fields being observed are displayed in the first line (*act* and *pri*), with the data currently in the link displayed below. Specifically, an O-link of delay 2 is displayed after two update cycles. Thus, there is information in the first two slots of the link, but no information is available yet for retrieval from the link (the top slot is empty). It can be seen that the activation value of the observed node changed from 0.27688 to 0.28092 in between updates of the O-link, while the priority remained constant.

Users have the option of altering values displayed in text fields, and can thus directly influence the behavior of the system to test various aspects of an architecture. It is also possible to instantiate components and links or delete them in the running architecture (as long as the operation does not conflict with the resource limits in the architecture layout).

6.3. ADE *as Test Tool*

The synchronized updating mode of components and links in ADE is particularly useful for testing purposes, as an architecture can be "frozen" at any point in the life-time of an agent and inspected. Users control when the next update occurs, and can inspect and modify architectural parameters between updates. It is even possible to change the architectural layout and continue the update process with a modified architecture.

ADE provides several tools for tracking, displaying, and analyzing information and information flow in the architecture. The "Graph" tool, for example, can be used to track the values of any variable of type *APOCObservable* in a component over time. Graphs of the temporal evolution of these variables can then be saved or printed. Figure 8 shows an example of the graph tool, tracking the activation value of an APOC component.

Other tools, such as the "Abstraction" tool, for example, allow users to automatically group components according their level of abstraction as determined by their C-link structure. This is intended to help users in isolating structures that might have formed in a self-modifying architecture over time for later reuse (for details see[33]).

Most importantly, it is possible in ADE to insert "inspection components" into an architecture without influencing the processing of existing components. These inspection components can observe any part of the architecture via O-links and report the data to the user, either through the graphical interface, or by saving the data to a file.

Testing of architectures in ADE can also make use of the fact that communication among components is done exclusively through links. As already mentioned, the same architecture can be run in both a robotic agent and a simulated virtual agent as long as agent servers exist that allow the architecture to connect to the respective "body representations" of the agents (i.e., to the physical robot or the simulation environment). This has the advantage that an architecture can be tested in a simulated environment before it is run on a physical robot. The concept can be taken further by connecting a physical robot to the simulated environment and observing any divergent actions between the robot and the simulated agent previously controlled by the same architecture (e.g., which may be due to time delays in command execution, since effectors will not move instantaneously after a command is issued).

Having looked at the threefold role of ADE in the agent architecture development process, we now present two sample architectures that demonstrate ADE's versatility and effectiveness as a tool. The first is an example of a virtual multi-agent setting, in which each agent is implemented as an APOC component. All agents are distributed over multiple hosts and communicate via A-links. The second is an example of a single robotic agent, in which various APOC components of the robot architecture use distributed ADE servers (both agent servers and utility servers).

7. Virtual Multi-Agent System

ADE can be used to implement multi-agent systems in various ways. Each agent can be implemented by one or more APOC components, which themselves may reside on one or more hosts. Agents can interact with other agents through APOC links and, in the latter case, dynamically create, modify, and destroy parts of their and other agents' architectures. Which design is to be preferred will depend on various factors, such as the task to be accomplished by the multi-agent system, the available computational resources, etc.

In the following example, we first specify the task and lay out the architectures of the multi-agents system. Then we show how the system can be implemented in ADE.

7.1. *Agent Task*

The task for the multi-agent system is an information retrieval task, in which various domains on the internet (e.g., ".edu", or ".ac.at") are to be searched for web pages containing given keywords (e.g., "agent toolkits"). The results should be ranked according to additional criteria (e.g., rate of occurrence of key expressions such as "distributed architecture").

7.2. *Agent Architecture*

The architecture layout of the multi-agent system and its initial state are shown in the left and right halves of Figure 9(a), respectively. Three types of agent can be present in the running virtual machine. The *Start-up agent* takes a string to be searched for (e.g., "agent toolkit"), a set of relevant expressions for that search (e.g., "virtual and robotic agents", "distributed agent architecture"), and a set of domains over which the search is to be performed (e.g., ".edu", ".ac.it"). Once the *Start-up agent* has received its data, it creates (within the resource limits specified in the architecture layout) one *Search agent* for each of the given domains (if the number of domains exceeds the instantiation limit of the search agents, then the search is performed in part sequentially).

The *Start-up agent* passes the relevant data (i.e., the search string, the domain name, and the relevant keywords) to the *Search agent.* Then the *Search agent* uses one or more of a set of web-search engines (e.g., www.yahoo.com) to search for the information given by the search string and collects the results. Once the search is completed, the *Search agent* creates *Evaluation agents*, each of which receives a document and the set of relevant keywords. The state of the system after all *Evaluation agents* have been created is shown in Figure 9(b). Each *Evaluation agent* then parses its document, rates it based on the occurrence of the relevant keywords, and returns the rating to the *Search agent.* The *Search agent*, in turn, selects all documents with a rating higher than a given threshold and returns them to the *Start-up* agent. When all *Search agents* have returned their information, the multi-agent system has completed its task.

Note that the details of how the various agents achieve their tasks are not displayed in ADE because they have been implemented in the update function that represents each agent (for the sake of simplicity). It is, however, possible to distribute the computations perform by each agent over multiple APOC components, in which case the details of their implementation can be visualized and manipulated graphically in ADE.

7.3. *Setup in* ADE

The set-up steps for the experiment are as follows:

(1) Start the *Registry*, in this case on airolab2.cse.nd.edu

```
java com/ADERegistry
```

(2) Start the *GUI*

```
java APOCstart
```

The GUI server then automatically starts the *APOCServer*(s) by connecting to the remote host(s) specified in the configuration file (in this case airolab4.cse.nd.edu and airolab5.cse.nd.edu) and running the following command on each:

```
java com/apoc/APOCServer <registry hostname>
```

The information display features of both components and links in ADE can be used to supervise the progress of the agents in this task. Possible uses include

- inspection of the links between *Search agents* and *Evaluation agents* for the actual web address being evaluated
- inspection of the *Evaluation agents* for the current rating of the address being evaluated
- inspection of the *Search agents* for the highest current evaluation of an address

If, as a result of an inspection, the user decides that a particular address is not worth exploring, the corresponding *Evaluation agent* could be deleted, freeing up resources for the system to evaluate another address.

8. Robotic Agent

Robotic agents pose a different challenge for agent designers, as they operate in realtime and timely updating of components is crucial for the proper operation of the architecture. As with virtual agents, there several ways of structuring robotic architectures depending on how many utility and agent servers are involved. In the simplest case, there is only one agent server that represents the robots sensors and effectors and also runs the APOC virtual machine implementing the robotic architecture.

In the following example, we again first specify the task and lay out the architectures of the robot system, and then show how the system can be implemented in ADE.

8.1. *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. A typical situation encountered by the robot during this task is shown in Figures 10(a) and 10(b).

In the schematic of Figure 10(a), the opening between the two obstacles is 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. The architecture for such an agent is described in the following section.

8.2. *Agent Architecture*

The architecture development for a robotic agent takes place in two stages: designing the underlying structure of servers and designing the actual agent architecture. The server structure for the experiment is presented in Figure 11. 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.

The robot architecture used for the experiment is presented in Figure 12(a) (the run-time virtual machine side of the toolkit is shown). Information about the contents of the architecture can be added to the GUI. Thus, Figure 12(b) shows the same architecture, with labels placed next to each component. It should also be noted that in this image the links have been hidden so that the labels can be properly read. ADE offers facilities for viewing/hiding components, links, and labels through the "View" menu.

The *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. In this experiment, the determination of progress was made using the perceived size of the ball: progress is not made if the maximum perceived size of the ball does not increase over a preset period of time. 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.[32]

8.3. *Setup in* ADE

The setup steps for the experiment are as follows (first two steps are identical to those in the virtual agent):

(1) Start the *Registry*,
(2) Start the *GUI*
(3) Start the *RobotServer* and the *ImageAcquisitionServer*. This step and the previous one are interchangeable.

```
java com/pioneer/PioneerServerImpl <registry hostname>

java com/framegrabber/FramegrabberServerImpl <registry hostname>
```

(4) Start the *ImageProcessingServer*.

```
java com/camview/CamviewServerImpl <registry hostname>
```

(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.

As with the virtual agents, ADE architecture inspection mechanisms can be useful in several ways:

- Inspecting the *VisionNode* allows the user to see what the agent "sees" (as shown in Figure 6).
- 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.
- 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 ability to stop and restart the robot can in this instance be coupled with inspection mechanisms. The user can thus 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.[5]

9. Discussion

We introduced the ADE environment as a first attempt to bridge the gap between single and multi-agent frameworks and provide an architecture-neutral, multi-user tool for the development of virtual and robotic agent architectures in single and multi-agent settings. ADE is intended as very general, versatile tool for the implementation, development, testing, and deployment of agent architectures, providing functionality for:

(1) the description and implementation of agent architectures
(2) graphical insertion and deletion of parts of the architecture (i.e., components and links)
(3) the distribution of computations associated with an agent architecture over several computers
(4) the development of single and multi-agent systems in a homogeneous environment
(5) the modification of agent architectures throughout the life-time of the agent
(6) analysis and visualization tools for architectures and their components
(7) the use of "off-the-shelf" programs in agent architectures
(8) the use of the same control code in simulated and robotic environments

ADE allows the agent developer to implement, at different levels of abstraction, several *prima facie* unrelated formalisms (such as neural networks, cellular automata, and planning algorithms). Not only is it possible to compare the functionality of these different formalisms in a unified framework, but it is is also possible to combine several such formalisms in the same architecture (e.g. using neural networks for image processing in an architecture that also uses a planner, and sending the information obtained in processing the image to the planner).

We demonstrated ADE's utility as design, implementation, test, and run-time tool with two examples, a group of virtual agent and a robotic agent, respectively. These examples were primarily intended to cover as much of a range of different architectural designs and setups as possible within the given space restrictions to show ADE versatility. Consequently, the tasks for which they were design were 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.[7, 33, 34]

Although ADE already provides a large number of features, the development and improvement of such a toolkit is an ongoing process.[¶] Future work in ADE will improve the rudimentary algorithm that extracts information about substructures (such as modules) and their functional organization from architecture descriptions. We will also include interfaces to allow ADE to work together with other agent toolkits and agent simulation environments. Statistical tools will also be included in the toolkit to provide direct access to measures such as the average and standard deviation of the amount of information transferred through links per second. We believe that ADE fills an important niche in the existing agent development tool landscape and hope that it will become a valuable addition which, because of its versatility, will appeal to a diverse group of agent architecture designers.

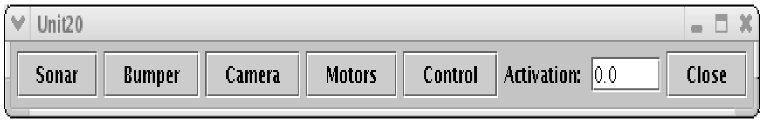
References

1. IBM agent building environment. http://rtdcs.hufs.ac.kr/docs/intelligent_java/abe/.
2. Adventnet. http://www.adventnet.com/products/javaagent/.
3. Agentbase. http://www.sics.se/~market/toolkit/.
4. IBM aglets. http://www.trl.ibm.co.jp/aglets/.
5. 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.
6. J. R. Anderson, D. Bothell, Byrne M. D., and C. Lebiere. An integrated theory of the mind. To appear in Psychological Review.
7. Virgil Andronache and Matthias Scheutz. Contention scheduling: A viable action-selection mechanism for robotics? In Sumali Conlon, editor, *Proceedings of the Thirteenth Midwest Artificial Intelligence and Cognitive Science Conference, MAICS 2002*, pages 122–129, Chicago, Illinois, April 2002. AAAI Press.
8. R. C. Arkin. Motor schema-based mobile robot navigation. *International Journal of Robotic Research*, 8(4):92–112, 1989.
9. S. Balakirsky, E. Messina E., and J. Albus. Architecting a simulation and development environment for multi-robot teams. In *Proceedings of the International Workshop on Multi Robot Systems*, Washington, DC, 2002.

[¶]The current beta version of the toolkit is freely available and can be downloaded from http://www.cse.nd.edu/~airolab/apoc

10. 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.
11. 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.
12. 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.
13. Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. 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*, pages 97–108, London, April 1999.
14. Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.
15. Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.
16. 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.
17. 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.
18. 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.
19. Brian Gerkey et al. Most valuable player: A robot device server for distributed control. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1226–1231, Wailea, Hawaii, October 2001.
20. Brian Gerkey et al. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th International Conference on Advanced Robotics*, pages 317–323, Coimbra, Portugal, June 2003.
21. L. P. Kaelbling and S. J. Rosenschein. Action and planning in embedded agents. In Pattie Maes, editor, *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, pages 35–48. MIT Press/Elsevier, 1991.
22. 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.
23. K. Konolige, K. L. Myers, E. H. Ruspini, and A. Saffiotti. The Saphira architecture: A design for autonomy. *Journal of experimental & theoretical artificial intelligence: JETAI*, 9(1):215–235, 1997.
24. Kurt Konolige. Saphira robot control architecture. Technical report, SRI International, Menlo Park, CA, April 2002.
25. J. E. Laird, A. Newell, and P. S. Rosenbloom. SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.
26. P. Maes. How to do the right thing. *Connection Science Journal*, 1:291–323, 1989.
27. Nelson Minar, Roger Burkhart, Chris Langton, and Manor Askenazi. The Swarm simulation system, a toolkit for building multi-agent simulations. http://www.santafe.edu-

/projects/swarm/overview/overview.html., 1996.
28. M. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry.* MIT Press, Cambridge, MA, 1969.
29. 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.
30. P.S. Rosenbloom, J.E. Laird, and A. Newell. *The Soar Papers: Readings on Integrated Intelligence.* MIT Press, Cambridge, MA, 1993.
31. Matthias Scheutz. Ethology and functionalism: Behavioral descriptions as the link between physical and functional descriptions. *Evolution and Cognition*, 7(2):164–171, 2001.
32. Matthias Scheutz and Virgil Andronache. Architectural mechanisms for the dynamic selection of behaviors in behavior-based systems. under review.
33. Matthias Scheutz and Virgil Andronache. APOC - a framework for complex agents. In *Proceedings of the AAAI Spring Symposium.* AAAI Press, 2003.
34. 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.
35. A. Sloman. Damasio, Descartes, alarms and meta-management. In *Proceedings International Conference on Systems, Man, and Cybernetics (SMC98), San Diego*, pages 2652–7. IEEE, 1998.
36. A. Sloman. Sim_agent help file, 1999.
37. Katia Sycara et al. The retsina mas infrastructure. *Autonomous Agents and Multi-Agent Systems*, 7(1):29–48, 2003.



Activation: 0.0

Fig. 5. Node information display

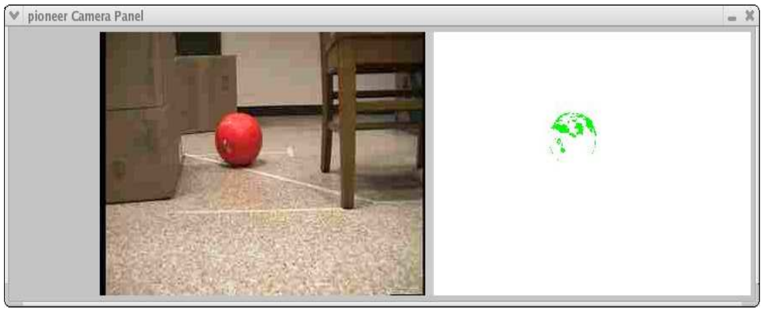
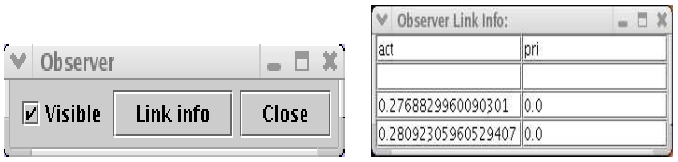


Fig. 6. Camera panel displaying the original picture (left) and the post-processed image, the contour of the ball (right)



act	pri
0.2768829960090301	0.0
0.28092305960529407	0.0

Fig. 7. O-link information GUI (left) and data display (right)

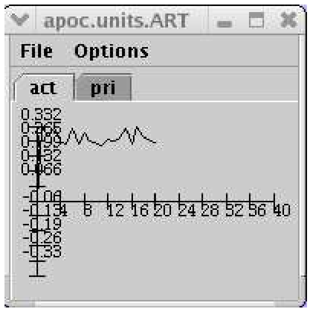
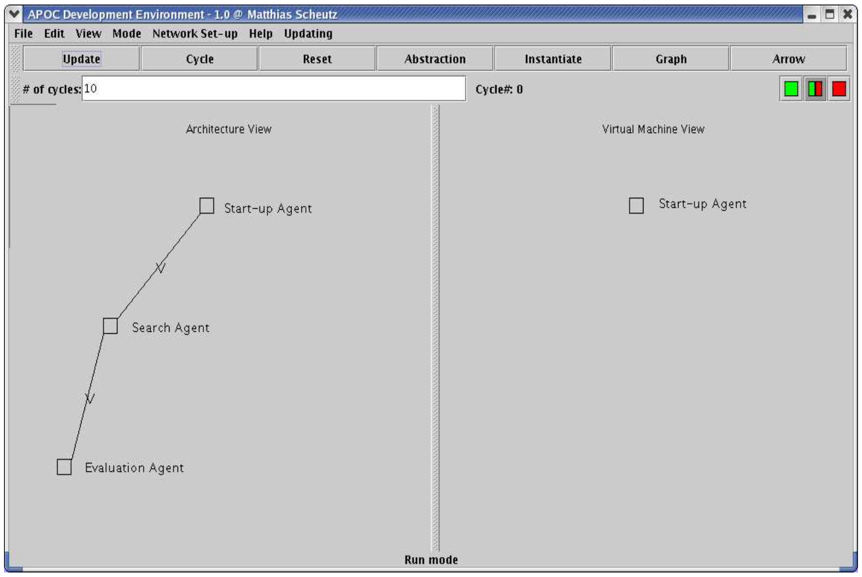


Fig. 8. Sample graph data in ADE



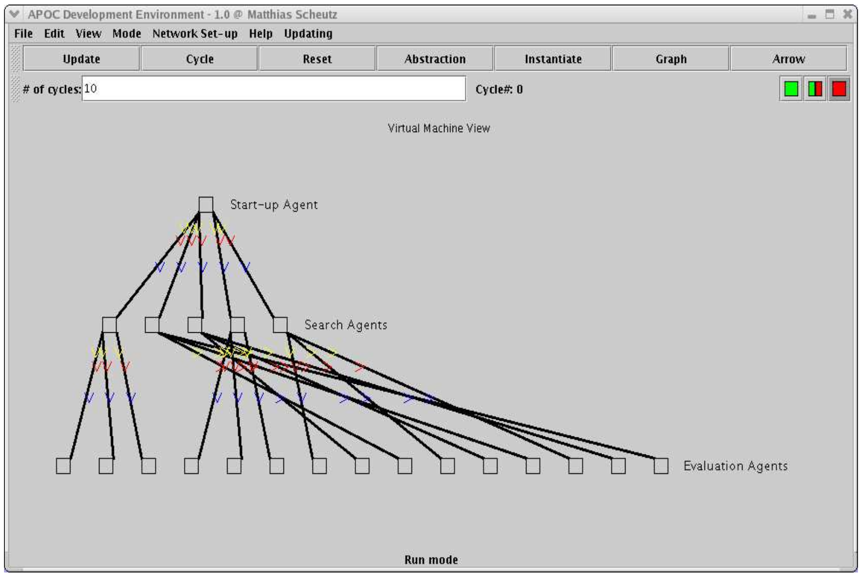


Fig. 9. ADE Initial architecture for web search agents (top) and the final architecture of the system (bottom)

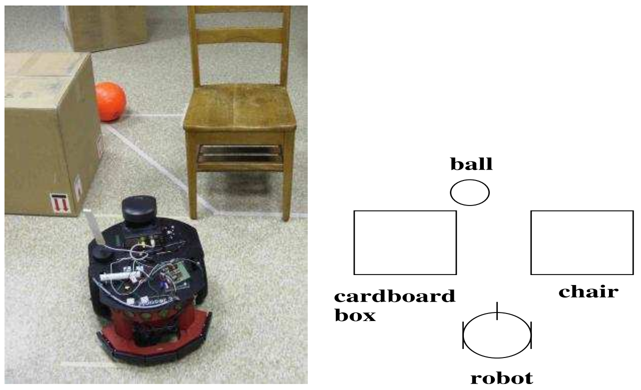


Fig. 10. Typical situation for the robotic agent: the path towards its target is obstructed, although the target is still in sight. The actual situation is shown on the left and a schematic on the right

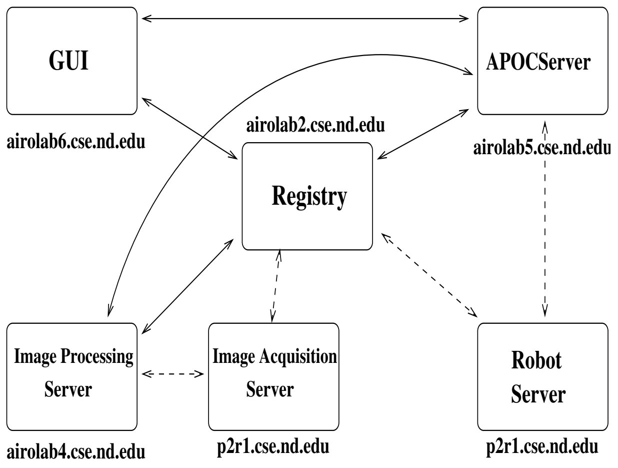
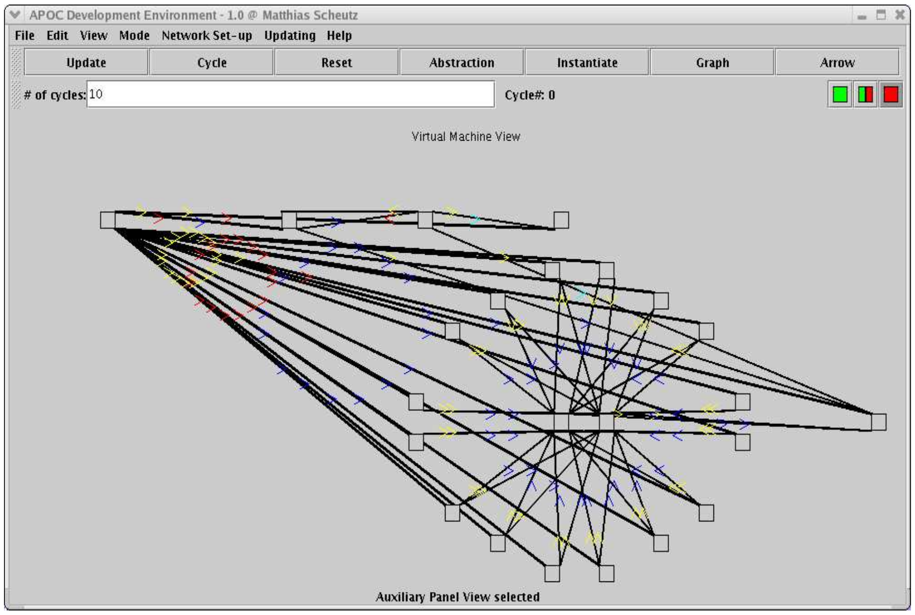


Fig. 11. ADE Set-up for ball retrieval in the robot experiment


APOC Development Environment - 1.0 @ Matthias Scheutz
File Edit View Mode Network Set-up Updating Help
Update
Cycle
Reset
Abstraction
Instantiate
Graph
Arrow
# of cycles: 10
Cycle#: 0
Virtual Machine View
Auxiliary Panel View selected

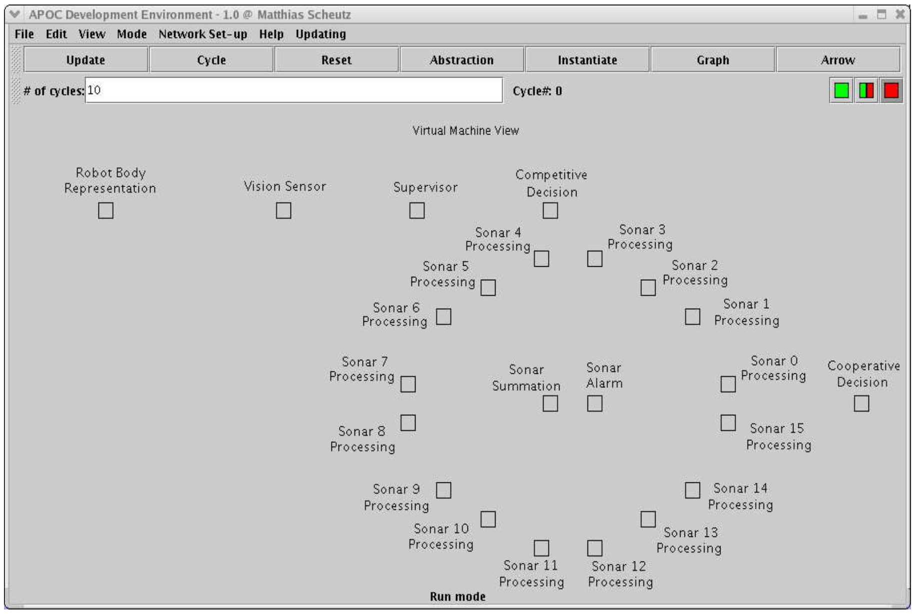


Fig. 12. ADE architecture for robotic experiment: virtual machine view with components and links (top), view with hidden links and added labels (bottom)