Integrating Theory and Practice: The Agent Architecture Framework APOC and its Development Environment ADE

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

Abstract

In this paper we present the results of the combined development of APOC *and* ADE*, an architecture framework for the analysis, comparison, and design of agent architectures and a distributed agent development environment which implements* APOC *principles, respectively. We show how the* APOC *framework relates to other architectures and design methodologies and demonstrate how other architectures and architectural components can be combined in a single architecture through* APOC*. Next we describe the* ADE *development environment and highlight the implementation of* APOC *principles. We show how* ADE *gives users the freedom to seamlessly switch between single and multiple computer environments, serial and parallel execution, and robotic and virtual agents. Finally, we illustrate the features of* ADE *through a robotic agent example.*

1. Introduction

The landscape of agent systems contains a variety of toolkits and systems for the design of both distributed *multi-agent* (e.g., JADE [5], RETSINA [20], AGENTBASE [1], ZEUS [15], Mozart [21]) and *single-agent* systems (e.g, SimAgent [19], ARIA/Saphira/Colbert [10–12], Player/Stage [7, 8]).

Multi-agent systems typically provide features such as a distributed environment and communication protocols agent designers. For example, JADE [5] provides a communication language, a graphical user interface for controlling and monitoring agents, and a directory facilitator which provides services needed to allow agents to contact one another and communicate regardless of their locations in the system.

Single-agent systems provide tools for architectural design of agents. For example, ARIA and Saphira [11], for instance, provide system architecture capacities (a description of the robot, including sensors and effectors) and a robot control architecture (including predefined routines for such tasks are gradient-based navigation and localization). SimAgent [19], on the other hand, provides a set of libraries with functionality for the design of "agents with complex internal mechanisms."

However, existing systems rarely interweave characteristics of both single and multi-agent systems. Moreover, they do not have as a foundation a theoretical framework which will allow for the combination of components from different architectures into a single system. As a result, potential symbiotic effects which emerge from the interaction of components designed for different systems cannot be systematically explored.

In this context, we propose the ADE agent architecture development environment. ADE provides a user-friendly environment for the development of architectures for virtual and robotic agents in single and multi-agent settings. It is built according to the specifications of the APOC architecture framework [16, 17] a general, universal agent framework in which any agent architecture can be expressed and defined. Like many multi-agent systems, ADE is a distributed environment. Founded on a general architecture framework, the ADE development environment can run any agent architecture, either by using an APOC translation of that architecture or by embedding the entire architecture "as is" into a single architectural component. Together these two characteristics give ADE an automatic means of parallelizing the computation in any architecture. We will illustrate potential uses of ADE through an example in which components from two different types of architecture are combined in a straightforward manner to create a new, hybrid system. The system achieves a goal which could not be achieved independently by either single or multi-agent systems without the use of more complex components.

In the following, we first give an overview of the APOC framework and illustrate the characteristics which lend flexibility and robustness to ADE. Then we describe the user interface and the supporting environment, highlighting the implementation of APOC features in the development environment. Finally, an example of combining features from two architectures within the APOC framework is illustrated using a robotic application in ADE.

2. APOC

The idea of a unified theoretical framework for the implementation, design, and analysis of complex agents is not new. In this section we present the APOC framework and relate it to existing work in agent architectures.

2.1. APOC **Components and Links**

APOC is an acronym for "Activating-Processing-Observing-Components", which summarizes the functionality on which the APOC agent architecture framework is built: heterogeneous computational units called "components" which can be connected via four link types (activation, observation, process control, and component) to define an agent architecture. The components, links, and their interactions are described in detail elsewhere [16, 17]. Below we provide a brief description of their functionality.

In designing an architecture framework it was important not to place any unnecessary restrictions on the structure of the components which can be combined to form agent architectures. Thus, 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 process. To this end, the state of a generic APOC component was defined as consisting of: activation level, priority level, associated process, instantiation number, update function, inputs, and outputs.

Activation links (A-links) are the most general means by which components can exchange information. An A-link connects two APOC components and serves as a transducer. The information on the A-link can be transmitted without change, or an operation can be performed on the data as it passes through the link (e.g., numerical values could be "scales" by a particular factor analogous to the "weights" on connections in neural networks).

Priority links (P-link s) are an explicit means of allowing components to control other components' associated processes. Two numeric pieces of data are passed through a P-link: the priority level of the controlling component and a signal (START, INTERRUPT, RESUME or NOOP). The controlled component chooses from its incoming priority links the signal associated with the highest incoming priority and, if that priority is higher than its own, sends that signal to its associated process.

Priorities and P-links can be used to implement many types of control mechanisms, in particular, hierarchical preemptive process control. In embodied agents, such as robots, they could be used to implement emergency behaviors: the component with the associated emergency process would have the highest priority in the network and be connected to all the other components controlling the agents behavior, which it could suppress (thus implementing a "global alarm mechanism" as described by [18]).

Observer links (O-links) are intended to allow components to observe other components' inner states without affecting them. They extract the data from the observed component and send it to the observer. An example of O-link use is allowing a vision control component observe the visual processing component for the features found in the current image.

Component links (C-links) are used to instantiate and remove instances of APOC components at run-time (they are the only type of component that can instantiate or terminate an APOC component). They are also used to instantiate the other link types between APOC components and are themselves only instantiated by APOC components.

In the remainder of this section we show exemplify how work in behavior-based architectures, cognitive architectures, and general frameworks for architecture development relates to the APOC framework. We present two examples from each class and show how they relate to our framework. For two architectures, the behavior-based Subsumption architecture and the cognitive ACT-R architecture, we furthermore present direct translations into APOC.

2.2. Behavior-Based Architectures

2.2.1. Subsumption. The *subsumption architecture* [6] is a layered system, in which individual layers work on individual goals concurrently and asynchronously. Layers consist of nodes, each node being the representation of a behavior. Each behavior is implemented as an augmented finite state machine (AFSM).

Subsumption architectures can be translated into the APOC framework in a straightforward manner by defining their components, the augmented finite state machines(AFSM), as follows:

1. The state table is directly incorporated into the *update* function of an APOC node.
2. Environmental/data inputs map onto A-links
3. Inhibitor connections, which block inputs to nodes, use A-links and simple, specialized APOC nodes to decide whether to pass on information or whether to block it

4. Reset and suppressor connections, which substitute values for node outputs, can be implemented *via* P-links, coupled together with assigning nodes a priority proportional to the layer in which they are found. Thus, nodes in higher levels can control the execution of nodes lower in the layer hierarchy.
5. A-links are used for message passing.

2.2.2. GRL. [9]. GRL is a behavior-based architecture which treats arbitration mechanisms as "higher-level functions." Thus, GRL allows for the combination of some arbitration mechanisms in one architecture and moves closer to a complete integration of architecture and arbitration mechanism. APOC completes the process by running the arbitration mechanism in the form of specialized components in the architecture.

2.3. Cognitive Architectures

2.3.1. ACT-R. For a description of ACT-R we chose the recent description of version 5.0 by Anderson et.al. [3]. Its description includes the brain regions which map unto each functionality of the architecture. ACT-R uses interactions between data (chunks) and condition action rules (productions) to simulate cognition. ACT-R uses two types of memories - declarative and production, and a goal stack. On each update cycle, a single production is fired based solely on the goal that is currently pursued. The architectural layout of ACT-R is shown in Figure 1.

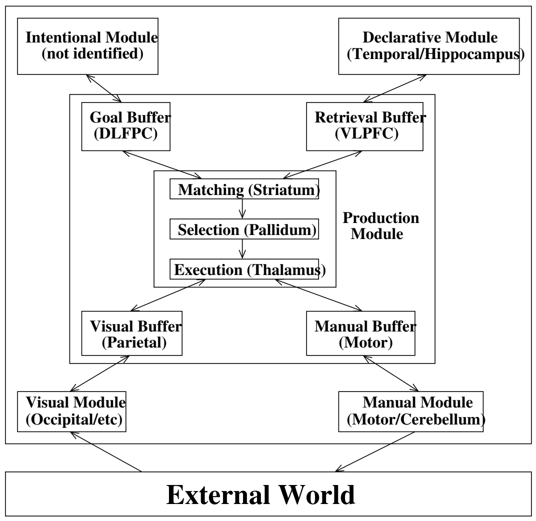

Figure 1. The ACT-R architectural structure.

The main components of ACT-R are described below.

- A *Perceptual − Motor System*, which interacts with the environment, receiving visual input and sending commands to the effectors. Perceptual information elements (chunks) are also created in this system.
- An *Intentional Module*, also known as a *Goal Module*, which holds representations of intentions and keeps track of them, so the behaviors serve the goal. Abstract and/or compound chunks are created here.
- A *Declarative* (*Memory*) *Module*, which holds and retrieves records of chunks.
- A *Procedural Module*, which performs two main functions: partial matching on the conditions to determine the rules which are eligible for execution, selection of one rule among those eligible for execution to fire in the current cycle.

In the simplest translation, each of the above elements can map to one APOC component. Chunks are also created as APOC components, which, in the ADE environment leads to the creation of a distributed memory structure among the participating computers on which the ACT-R architecture is distributed. A distributed matching system can then be implemented within an ACT-R architecture by creating a partial matcher on each machine. On each update cycle, the partial matcher reports which objects on its host match rule antecedents. The partial matches on each host are reported to the partial matcher from the host on which the rule resides, which then completes the matching process.

Communication among various elements of the architecture is made through a buffer associated with each element. Each buffer contains one piece of data (a chunk) which is accessible to the system at large. In APOC the buffer is simply a field in each component and its value is communicated to other architectural elements through the use of O-links.

The chunk retrieval process is activation based, with the activation of a chunk given by $A_i = B_i + \sum_j W_j S_{ji}$, where B_i is the base level activation of the chunk, W_j is the attentional weighting of the elements that are part of the current goal, and S_{ji} are the strengths of association from the elements j to chunk i. It should be noted that activation values map directly to the *act* element of APOC components . The base level is given by $B_i = \ln(\sum_{j=1}^{n} t_j - d)$, where n is the number of times element i in memory has been accessed (practiced), t_j is the time since the j^{th} practice of item i and d is a parameter estimated at 0.5. Retrieval time for chunks is then given by $Recognition\ time = I + Fe^{-A_i}$, with I and F being experimentally determined time constants. The retrieval time of a chunk can be adjusted in APOC by varying the delay on the link going to the component representing that chunk.

2.3.2. SOAR. Unlike ACT-R, Soar [13] uses a single memory to hold data and productions, with a working memory being the equivalent of the ACT-R goal stack. Soar allows multiple rules to fire within the same cycle. A translation of Soar into APOC is also available [4].

2.4. Frameworks

2.4.1. RS. RS [14] proposed a model of computation based on interaction among "concurrent computing agents". Agents in RS are instantiated schemas which are connected through universal message passing links. APOC builds upon RS by adding several capabilities, such as the option of performing the computation both synchronously and asynchronously and the possibility of specifying a time delay for each link.

2.4.2. RCS. RCS [2] is a design methodology which imposes no methodological constraints on the architecture developer. The universality of the RCS approach is a feature also present in APOC. However, RCS being only a design methodology, does not provide a structure within which agent architectures can be developed. APOC provides a definition of the components of an architecture (computational and communication elements), giving architecture developers the basic building blocks for developing architectures, possibly designed according to the RCS methodology.

The generality of APOC illustrated by the above examples is inherited by its implementation in the ADE development environment. ADE supports all the features mentioned above in behavior-based architectures, cognitive architectures, and system development frameworks. Many other features are present in ADE either as a direct result of its theoretical foundation, or by virtue of design/implementation decisions. We begin the description of our approach to integrating theory and practice with a description of the APOC architecture framework. We present the components and links of the framework and we show how APOC can be used to express architectures in a unified way.

The two translations illustrate a straight-forward mapping between the structure of components in other architectures and APOC components. Such translations are well-suited for the development of APOC templates representing generic components of other architectures. Translations of actual architectures into APOC are then reduced to a "fill-in-the-blank" process. Libraries of components built in various paradigms can thus be easily created by translating components of working architectures to their APOC counterparts. These libraries provide the opportunity of studying the interplay among various architecture design methodologies by allowing components from different paradigms to be used together. Several architectures have been implemented where co-operative and competitive elements were successfully merged using ADE, the development environment described in the next section.

3. ADE

In this section we will describe the functionality of the Agent Development Environment. Features which are directly related to the APOC framework (e.g., implementation of components) will be identified and general features (e.g., distributed computation) will be discussed. ADE will be presented in two parts: the user interface, showing the functionality available to the user, and the underlying system, describing the set-up which forms the practical foundation for the environment's flexibility.

3.1. The ADE User Interface

The user interface of ADE is shown in Figure 2. The interface is divided into two subspaces: an architecture view and a virtual machine view. The architecture view describes the relationship among components at the type level, i.e., what types (programs) can be present in the architecture, which types can be connected through links and what types of links can be part of those connections. The virtual machine view presents the instantiated version of the architecture: the instances of each type and the connections that have been created. Components and links at the virtual machine level implement (with minor variations) the components and links defined as part of the APOC framework. We present the interface in three sections: general functionality, the functionality of the architecture view space, and the functionality of the virtual machine space.

3.1.1. 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). Upon adding a type to an architecture, users are prompted to specify a number of instances which are created by default (possibly 0) and a maximum number of instances which can be part of the architecture. ADE uses a specialized class called APOCType to identify the types of components which can be present in the running virtual machine. A copy of this class is instantiated for each component type. After instantiation, each copy keeps track of the type it represents, the number of instances of that type which exist in the running virtual machine and the maximum number of instances which can simultaneously be present there. The Architecture space descriptions of virtual machine components act as resource managers in the system: they enforce the user-specified limits on the number of instances simultaneously present in the virtual machine and decide on what computer a new instance of that type should be created (if ADE is run as a distributed platform).

Links in ADE are created and configured through a link creation tool. In architecture space, links between two types indicate the potential of creating a link between instances of those types. 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.

3.1.2. Virtual Machine View In the running virtual machine, boxes indicate actual computational components present in the instantiated architecture. All components extend an APOCNode class, which provides them with: variables for activation level, priority level, and instantiation number, an update function (modifiable by users), and vectors for inputs and outputs. Since nodes in the APOC framework do not necessarily have an associated process, the APOCNode class does not provide a default associated process.

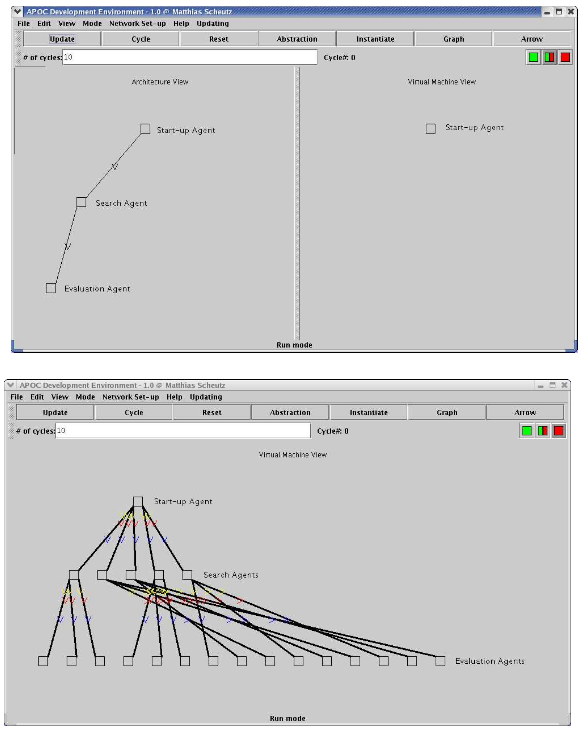

Figure 2. The ADE interface: (a) top - the type-level description and initial state of the run-time virtual machine, (b) bottom - the final state of the run-time virtual machine.

Edges represent instantiated APOC links. Multiple arrows can be present along each edge, indicating each type of instantiated link.ADE A-links are defined by a start node, a destination node, a vector containing data being passed through the link, an operator which is applied to the data, and the time it takes for the data to advance one element through the vector, referred to as the cycle time of the link. The cycle time, together with the length of the vector define the link delay of the APOC A-link. Similar parallels exist between each of the other three link types in APOC and their ADE equivalents.

It can thus be seen that the ADE implementation of components and links closely parallels their theoretical, APOC counterparts.

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.

3.2. The ADE Supporting Environment

The supporting environment provides the infrastructure to distribute agent architectures and to operate virtual as well as robotic agents. There are five servers in an ADE system: registry, APOC, GUI, agent and utility. Each type or ADE server will be briefly discussed below.

The Registry is a centralized resource manager for the system. It keeps track of the available servers and the services they provide. Therefore, each server which is part of an ADE system contacts the registry in order to make its services available to other system components. The initial communication specifies several parameters, such as maximum number of client connections supported and names of clients which are allowed to connect to the server. A client, then, is an element of the system (a server or a separate program such as an ADE computational component) which requires the use of the facilities provided by a server (e.g., a robot motor control node requiring a connection to the agent server in order to access the motors). A client requiring the services of the server contacts the registry and requests a connection. If a connection is possible, the registry connects the client with the server and is no longer involved in their transactions, thus avoiding the creation of a system-wide bottleneck.

APOC servers are independent entities which serve as hosts for components and links. Each APOC server has capabilities for creation and deletion of 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.

GUI servers are the interface of the ADE environment (their functionality was described in Section 3.1). Each *GUI server* maintains connections to all APOC servers in the system in order to relay user commands (such as insertion of a component) to APOC servers and to display new components (created during the life-cycle of the architecture).

Agent servers provide access to "bodies" of virtual or robotic agents. In the case of robotic agents, this involves establishing connections to the physical sensors and effectors of the robot. For virtual agents, agent servers provide a description of the body of the agent they represent and implementations of its sensors and effectors.

Utility servers provide additional distributed services that are not part of the agent architecture. They are used as an additional tool to ease the incorporation of complex features into ADE systems. Typically these servers implement computationally expensive operations.

3.3. Dimensions of ADE Flexibility

In this section we exemplify the flexibility present in the ADE environment by considering ADE. We show how ADE can run as a single and multi-computer environment, how it can function as a medium for serial and parallel execution and how it can be used for the implementation of virtual and robotic agents.

3.3.1. Single and Multiple Computer Environments. The ADE system has facilities to run both as a single and multi-computer system. Switching between the two modes can be done easily through the GUI. In switching from a multi-computer system to a single computer system, the GUI shuts down APOC servers running on machines other than the one on which the GUI is running. Switching from a single-computer system to a distributed system simply requires the GUI to go to through a normal start-up sequence for the hosts users want added to the system.

3.3.2. Serial and Parallel Execution. ADE provides two modes in which architectures can be run: a ”Synchronous” mode and an ”Asynchronous” mode. In the "Synchronous" mode, each node of the architecture updates once, then waits for all other nodes to complete their computation before proceeding to the next computational cycle. In "Asynchronous" mode, each node updates independently with an execution thread associated with each node.

3.3.3. Robotic and Virtual Agents. Web-based agents can be implemented in ADE using the JAVA URL class, which allows the opening of web documents (see also Figure 2). "Embodied" virtual agents can also be developed and tested by using an available simulator which is run as a *utility server* in the system. Body descriptions in the simulator can be configured to provide the same access methods as those available on an actual robot.

Robotic agents can be developed by running *agent servers* which provide access to the sensors and effectors of the robot. The same control code can be executed on both the robotic and virtual agents, allowing a seamless transition between the two realms and providing a simulated testing environment for robot control code.

The robotic agent example provided in the next section gives additional details on facilities provided by ADE.

3.4. ADE Example

In this section we present an example of combining two architecture design mechanisms within a single agent while re-using computational components. Some of the facilities provided by ADE to agent developers in order to ease optimization of their designs are also illustrated.

The robot was given 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.). A typical situation encountered by the robot during this task is shown in Figure 3.

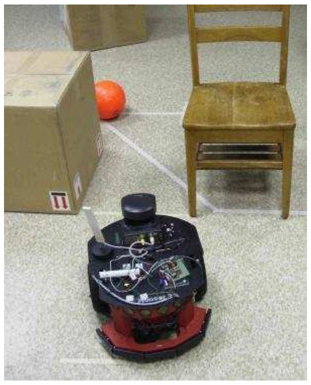

Figure 3. Typical situation for the robotic agent: the path towards its target is obstructed, although the target is still in sight.

In the situation of Figure 3, 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.

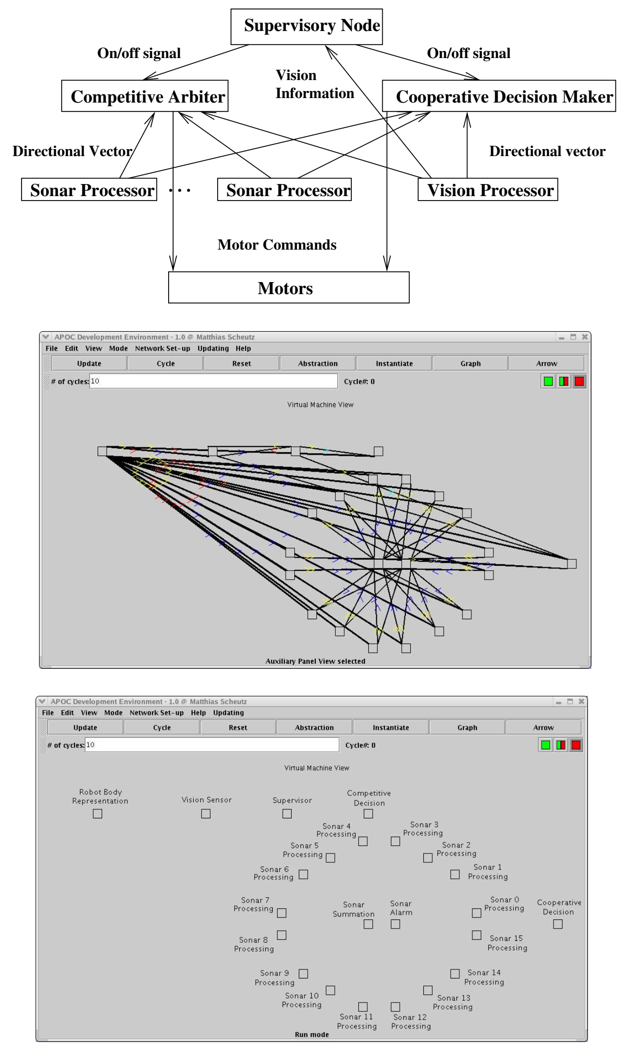

Figure 4. ADE **architecture for robotic experiment: (a) top - schematic of the** ADE **architecture for the robotic experiment, (b) center - virtual machine view with components and links, (c) bottom - view with hidden links and added labels.**

The robot architecture used for the experiment is presented in Figure 4(b) (the run-time virtual machine side of the toolkit is shown).[1]. A schematic of the architectural components and their interactions is shown in Figure 4(a).

1 Transparent to the user, components in the virtual machine execute on several hosts which were part of the ADE system. Thus, if ADE is run in multi-computer mode, users do not have to make explicit arrangements to distribute computation

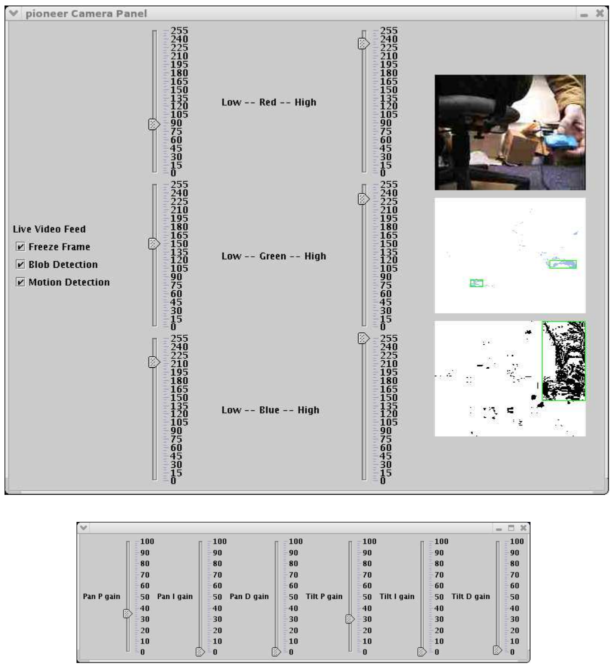

Figure 5. (a) Control panel for dynamic color range adjustment (top) and (b) control panel for dynamic adjustment of PID controller parameters (bottom)

Information about the contents of the architecture can be added to the GUI. Thus, Figure 4(c) 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.

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.

The process of identifying and fine-tuning the colors identified as potential ball matches was aided by the ease of adding visual interfaces to ADE components.

Figure 5(a) shows the panel which was added to the vision server and which was used to configure the parameters for blob detection. The panel displays three images: the unmodified camera picture (top), the results of performing blob detection on that image with the parameters set on the sliders (middle), and the results of performing motion detection on the original image.

Camera control was performed through the use of 2 PID controllers - one for horizontal movement and one for vertical movement. A similar calibration process was performed on the parameters of each controller, using another graphical tool (Figure 5(b)), which allowed us to change parameters as the architecture was running.

4. Conclusion

In this paper we have presented the universal architecture framework APOC and its development environment, ADE. We have illustrated the generality of the framework with regard to behavior-based architectures, cognitive architectures, and other frameworks and we introduced the possibility of combining components from different architectures to produce new and useful behaviors in agents.

Then we described the APOC development environment, ADE, and highlighted the relation between implementation and theoretical framework. Desirable features, both derived from APOC properties (e.g., universality) and design decisions (e.g., use as both a single-computer and multi-computer environment) were presented, and a practical example of ADE functionality was given. Work is currently in progress on integrating the ACT-R cognitive architecture with a behavior-based system.

References

[1] Agentbase. http://www.sics.se/˜market/toolkit/.

[2] 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.

[3] J. R. Anderson, D. Bothell, B. M. D., and C. Lebiere. An integrated theory of the mind. To appear in Psychological Review.

[4] V. Andronache and M. Scheutz. Ade - an architecture development environment for virtual and robotic agents. To appear in the International Journal of Artificial Intelligence Tools.

[5] 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.

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

[7] B. G. 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.

[8] B. G. 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.

[9] I. Horswill. Functional programming of behavior-based systems. *Autonomous Robots*, (9):83–93, 2000.

[10] K. Konolige. Colbert: A language for reactive control in saphira. In *Proceedings of the German Conference on Artificial Intelligence*, pages 31–52, Freiburg, Germany, 1997.

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

[12] 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.

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

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

[15] 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.

[16] M. Scheutz and V. Andronache. APOC - a framework for complex agents. In *Proceedings of the AAAI Spring Symposium*. AAAI Press, 2003.

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

[18] 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.

[19] A. Sloman and R. Poli. Sim_agent: A toolkit for exploring agent designs. In M. Wooldridge, J. Mueller, and M. Tambe, editors, *Intelligent Agents Vol II (ATAL-95)*, pages 392–407. Springer-Verlag, 1996.

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

[21] P. Van Roy and S. Haridi. Mozart: A programming system for agent applications. In *International Workshop on Distributed and Internet Programming with Logic and Constraint Languages*, 1999.