A Layered Heterogeneous Cognitive Robotics Architecture

Alistair E. R. Campbell
Department of Computer Science
Hamilton College
198 College Hill Road
Clinton, NY 13323
acampbel@hamilton.edu

Debra T. Burhans
Department of Computer Science
Canisius College
2001 Main Street
Buffalo, NY 14208
burhansd@canisius.edu

Abstract

The snarpy cognitive robotics architecture is a new member of the family of GLAIR architectures that performs grounded reasoning and action on real and simulated robots. We employ the SNePS knowledge representation, reasoning, and acting system and the Pyro robotics toolkit. We describe the architecture together with the methodologies used to realize it. With the snarpy architecture, robotic agents can be very easily and quickly constructed having both high-level reasoning skills *and* low-level acting and sensing abilities on real robotics hardware. Two prototype snarpy agents are presented.

Introduction

This paper presents a layered, heterogeneous architecture for cognitive robotics that is grounded in Pyro (*Py*thon *ro*botics). Our work has focused on connecting a SNePS (*S*emantic *Ne*twork *P*rocessing *S*ystem) agent to a Pyro robot. However, the framework we have developed, snarpy (an *ar*chitecture linking *SN*ePS with *Py*ro), is extremely flexible and allows for the integration of *any* formal knowlewdge representation, reasoning, and acting system with Pyro.

Cognitive Robotics is focused on higher-level actions and perceptions rather than lower-level details of robot interaction with a world. A programming environment for cognitive robotics provides a modeler with a set of primitives that can be incorporated into agent "brains" that control an associated robot. One such framework is Tekkotsu (Touretzky & Tira-Thompson 2005), which is used with AIBO dogs. GOLOG (Levesque *et al.* 1997) is a comprehensive logic-based system for robot control in dynamic environments, incorporating temporal and spatial logical primitives and reasoning. Our work provides a new framework within which cognitive modelers can experiment using SNePS, a high-level knowledge representation and reasoning system, to control any of the robots in the Pyro framework.

The architecture we have implemented, GLAIR (Grounded Layer Architecture with Integrated Reasoning) (Hexmoor, Lammens, & Shapiro 1993; Shapiro & Ismail 2003), is one of a number of multilayer models that have been proposed for robot control. An overview of robotic architectures may be found in (Arkin 1998). A number of projects have used SNePS and GLAIR for robotic control, see, for example, (Shapiro 1998).

Copyright © 2006, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

We present a detailed description of the architecture in the next section. Following this we describe two experiments that demonstrate the efficacy of our approach. We conclude with a brief discussion and directions for future work.

Architecture

In this section we discuss the principal components of a layered heterogeneous cognitive robotics architecture. We first describe the SNePS system which defines the top layer. Then we focus on GLAIR: a family of layered agent architectures of which snarpy is a new member. We also discuss how a single heterogeneous cognitive agent is created using multiple languages and systems, and how communication is achieved between the parts of this cognitive agent.

SNePS

SNePS is a knowledge representation, reasoning, and acting system (Shapiro 1979; Shapiro & Rapaport 1992) whose principal knowledge structure is a semantic network. A distinguishing characteristic of this network is that mental entities are represented by nodes. Nodes may be defined using a logic interface that builds them. In a SNePS network, nodes are partitioned into four kinds (Shapiro *et al.* forthcoming).

- **Proposition** is the kind that can be believed; for example, `shouldgo(left)` represents the proposition that a robotic agent should turn left.
- **Act** is the kind that can be performed; for example, the node `left` represents the action of turning left.
- **Policy** is the kind that links acts and propositions; for example, `ifdo(shouldgo(left),decidedirection)` represents the policy that if an agent needs to know whether he should go left, the agent should perform the act `decidedirection`.
- **Thing** is any other kind (including individuals, classes, relations, etc.)

Figure 1 shows a network for this example.

SNePS provides facilities for deducing the assertion status of a proposition—whether the agent believes it—and

for performing arbitrarily complex mental and physical acts. SNePS operates in an open world: a proposition may be asserted, its negation may be asserted, neither, or even both, in which case SNePS can signal a contradictory belief space, giving options including the choice to continue deduction in the presence of the contradiction without the risk that many traditional systems pose: that of inferring any proposition whatsoever. This gives a SNePS-using cognitive robot robust abilities to perform hypothetical reasoning in multiple belief spaces (Shapiro 1993; Chalupsky & Shapiro 1994; Johnson & Shapiro 2005a; 2005b).

In addition, SNePS has an integrated facility for acting and inference (Kumar 1996; Kumar & Shapiro 1994a; 1994b). For example, an agent implemented in the SNePS system may perform the act `believe(x)`, which is a mental action that causes a change in `x`'s assertion status for the agent. Preconditions and effects of actions are computed in the course of acting. For example, one effect of `believe(x)` is that the assertion of `x`'s negation is removed, and `x` is asserted.

Two basic sorts of **act**s are primitive acts and composite acts. Primitive acts are usually defined by the agent designer, but some, such as `believe`, are pre-defined. SNePS provides built-in facilities for constructing new composite acts from other acts. Three of the constructs that are of particular use in our current `snarpy` systems are

- `snsequence(a1,a2)` is the composite act whose performance consists of performing `a1` and then performing `a2`.
- `withall(?x,p(?x),a(?x),da)` is the composite act whose performance consists of deducing all entities `?x` such that proposition `p(?x)` holds, and performing act `a(?x)` on each such `?x` in a non-deterministic order. If no such `?x` can be deduced, the default act `da` is performed.
- `withsome(?x,p(?x),a(?x),da)` is the composite act whose performance consists of deducing all entities `?x` such that `p(?x)` holds, non-deterministically choosing one of them, and performing act `a(?x)` on it. If there are no entities satisfying `p(?x)`, the default act `da` is performed.

There is also a facility for naming composite acts:

- `ActPlan(a,p)` Is the proposition that a plan for performing the named act `a` is to perform the (usually composite) act `p`.

GLAIR

In the Grounded Layered Architecture with Integrated Reasoning (GLAIR) family of cognitive robotics architectures (Hexmoor, Lammens, & Shapiro 1993; Shapiro & Ismail 2003; Shapiro & Kandefer 2005), a robotic agent is realized in 5 distinct layers of processing. The top layer is where conscious reasoning and acting decisions occur. The lowest layer is where the agent physically operates in its environment. The layers in between serve to mediate and translate high-level decisions to actuators and sensors. A GLAIR system may encompass more than one concurrent computational process.

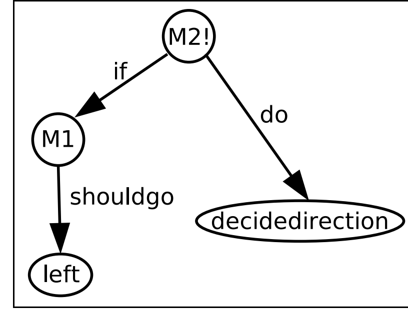

Figure 1: M1: `shouldgo(left)`: The proposition that the robot should go left; M2: `ifdo(shouldgo(left),decidedirection)`: the policy that in order to determine whether the robot should go left, the robot should perform a `decidedirection` action

1. **KL:** the *knowledge layer* at which high-level conscious decisions are made, and at which complex acts are formulated and scheduled for execution. For example, `because there is an obstacle ahead and to the right, turn left until the way is clear to move forward` is a KL reasoning process and behavior. The KL is distinguished from all other layers by being described entirely in terms of well-formed-formulae in some logic, rather than in terms of functions implemented in a computer programming language. In GLAIR agents, the KL is usually implemented in SNePS.
2. **PMLa:** the primary *perceptual-motor-layer* at which are implemented any unconscious actions requiring direct access to the KL. There are two basic kinds of actions here.
   (a) *control-processing algorithms* (such as sequence, selection, iteration) organizing the lower-level agent behaviors in service of the KL decision. For example, if an act has more than one applicable plan, this is the layer at which a plan is selected. Or, if an action is to be performed repeatedly, this is the layer at which the repetition occurs until a condition is met.
   (b) *basic primitive actions* which are executed by the agent. For example, `select a motion direction among` *`straight`*`,` *`hard-left`*`,` *`left`*`, and` *`right`* is a PMLa primitive action because it requires asserting the choice into the KL.
3. **PMLb:** the secondary *perceptual-motor-layer* at which are implemented other algorithms which are needed in service of higher-level PMLa actions. These algorithms run in the same computational process as the KL but do not require direct access to the KL structures. For example, `start a hard left turn, wait 0.25 seconds, and stop turning` is a PMLb primitive action which may be performed repeatedly in service

of the KL action example above. We also consider any facility that enables communication between a process implementing the higher layers of GLAIR and another process implementing the lower layers of GLAIR. Such a facility (for example, *robotipc*, described in detail below) would be considered a PMLb construct.

4. **PMLc:** the tertiary *perceptual-motor-layer* at which behaviors at the layer above are realized on a particular architecture. PMLc usually exists in an entirely different computational process, and may be implemented in a different programming language, or even on a different computer. For example, A third-party robot control program (such as Pyro) is a PMLc process in a GLAIR system. This layer is responsible for executing an action such as `set the rotational velocity to 120 degrees per second`, implementing part of the PMLb turning action example above. Another example might be `determine the closest shade (red, orange, yellow, etc)` of the object the robot is looking at, based on sensor data from the layer below.
5. **SAL:** the *sensory-actuator layer,* at which robot hardware motors are made to run, and at which sensors record and return raw data to the layer above. The robot may be real or simulated.

There is some flexibility in creating a robot from these GLAIR layers. Whether a computation is done at PMLb or PMLc is largely a matter of creative choice. For example, the problem of determining object shade could be solved entirely at PMLc, entirely at PMLb (because a PMLc process merely sends the raw data from SAL directly back to PMLb), or, usually, partially at PMLc, and partially at PMLb. Then, once PMLb determines the shade, the answer is returned to PMLa for assertion at KL.

When considering whether a GLAIR agent is properly called a "cognitive agent", the degree to which the KL and PMLa interact is critical.

Pyro

The Pyro toolkit is a Python-based, open source programming environment for robotics (Blank *et al.* 2006; Blank, Meeden, & Kumar 2003; http://www.pyrorobotics.org). Pyro allows for the development of a variety of agents, from simple reactive agents to neural network based learning agents. The Pyro framework provides underlying support for a variety of actual and simulated robots by providing a large set of high-level primitives for interacting with sensors and effectors. The Pyro user is thus free to focus on behavioral and cognitive issues in the design of agents. Supported robots include the Khepera, Pioneer, and AIBO, among others.

A Pyro "brain" comprises a Python program that interacts with a robot. This program accesses robot sensor data, including vision if a camera is available for the robot, and can issue commands that are associated with robot behaviors such as move, translate, and rotate. The default unit of measure in a "world" is one robot unit, which corresponds approximately to the diameter of the robot to which it refers.

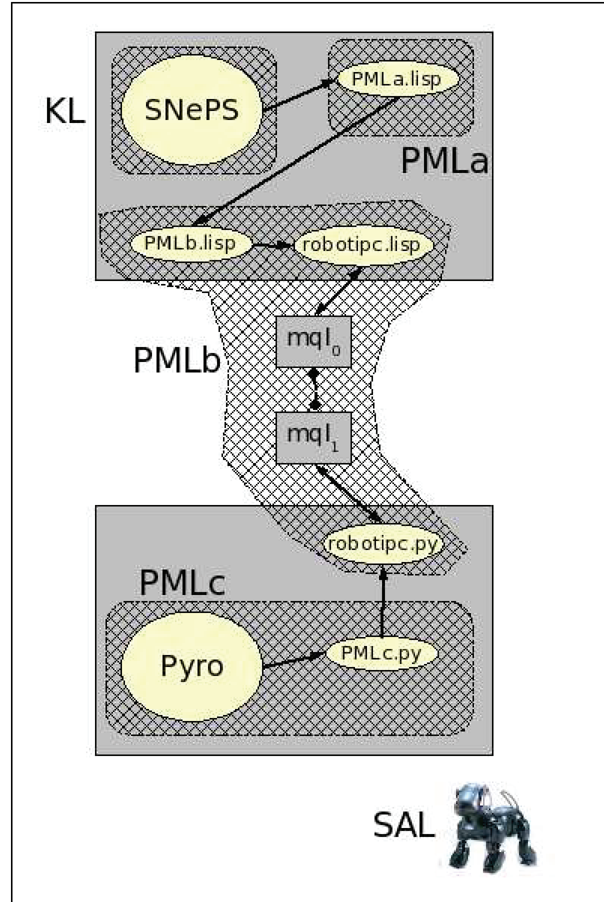

Figure 2: The `snarpy` system: a GLAIR agent having concurrent processes controlling an AIBO™ robot dog

In theory, the same brain can be associated with a number of different robots with few or no changes. This is the most powerful feature of the Pyro framework. In practice this is true as long as the robots share the same types of sensors (e.g. range sensors) and the brain is written with enough generality. However, even across different robot platforms much of the code in a brain program will work for any supported robot.

While there has been significant focus on the use of Pyro as an innovative and effective tool for teaching AI, the toolkit also provides an ideal grounding for cognitive robotics research. It is the linkage of sophisticated KR to the Pyro system that comprises the primary contribution of our work.

`snarpy` robots

Our cognitive robot is realized on a single computer in an architecture involving two concurrent processes. The *higher brain* is a Lisp process running SNePS; the *lower brain* is a Python process running Pyro. The higher brain encompasses the first three GLAIR layers: KL, PMLa, and the Lisp side

of PMLb. The lower brain encompasses the Python side of PMLb, PMLc and SAL, including the robot machinery itself. Since Pyro conveniently abstracts away details of the robot, it doesn't matter to us whether the robot is real or simulated.

Figure 2 shows the major components of `snarpy`. The large gray rectangle on top represents the Lisp process; the one on the bottom represents Python. The GLAIR software layers are indicated with hashed regions and labeled. The robot dog represents the SAL. While the figure includes an AIBO robot it could be replaced by any of the robots supported by Pyro. Control is transferred from the SNePS system through the GLAIR layers to Pyro, which signals the hardware, as follows:

1. The SNePS agent performs a high-level act.
2. The acting component of SNePS finds a detailed plan for doing the act and organizes its performance through PMLa.
3. Primitive unconscious actions are executed in PMLa and PMLb, which result in requests sent through PMLb's *robotipc* interface (discussed below).
4. The PMLc.py brain for Pyro is simple. It repeatedly queries its robotipc interface for messages from the higher brain, and acts on them accordingly. Examples of two kinds of messages PMLc can handle are shown below:

   move(*translate*, *rotate*) for moving forward or backwards at speed *translate* (with 1 being full speed forward and -1 full speed backward) and *rotate* specifying the speed and rotation direction. Pyro sets these velocities but makes no response to the higher brain.

   distanceq(*direction*) for asking the distance to an obstacle in the specified *direction* $\in$ {front,left-front,right-front} based on the most recent range sensor values. Pyro responds with the answer.
5. Any requirement for a particular robot acting or sensing operation is determined by Pyro and sent from the host computer to the actual hardware (SAL) by radio or other signal as provided by the robot. Responses from the SAL sensors are similarly received by Pyro and made available to PMLc.

In our architecture, the robot acts according to cognitive decisions made in the higher brain. An alternative architecture utilizing the same components could have a Python brain in charge of the decisions (cognitive or otherwise) but also use an external system for performing reasoning if needed–a SNePS-based database or another agent perhaps. In that case, control is transferred through the robotipc subsystem to query the external system. Pyro simply waits for a response, and continues processing. Since a number of Pyro brains have already been constructed it would be easier to add a reasoner as an extra component to an existing brain rather than mapping the brain functions onto an architecture that requires the KL to be the driving process.

Brain communication via *Robotipc*

Communication between the higher and lower brain occurs through a C program (*mqI*) that runs as a subprocess of each brain component. The mqI program facilitates message-passing via POSIX message queues. A simple protocol allows two processes to communicate by sending and receiving character string data. Once initialized, the C program is responsible for allocating the queues and reading requests from its parent process. The brain component interfaces with mqI via the latter's standard input and output, accessible to parent programming languages as file descriptors. The protocol supports basic requests such as *send* and *receive*. For example, the *send* request is accomplished by writing the word "SEND" followed by a string to be sent to the other process. At startup, the mqI program reads three important numbers—an *agent number* $\in \{0,1\}$ representing which brain component this instance of mqI is serving (e.g, 0=Higher, 1=Lower); and two positive integers: system level identifiers of the message queues to be allocated and shared between the processes (e.g. 500 and 501).

To access mqI, each brain component utilizes a package or module called *robotipc* whose function are abstractions of the basic requests. This way, the brain doesn't have to be concerned with actual message queue system calls, or with the mqI protocol itself. Robotipc supports the following operations:

1. **INITIALIZE**: creates the mqI subprogram, gives it agent and message queue numbers.
2. **RECEIVE**: blocks until a message has been sent by the other process; returns the message.
3. **SEND(message)**: sends *message* to the other process. If the message queue is full, this operation will block until the other process executes a receive operation.
4. **QUERY**: Returns *true* if a message has been sent by the other process; *false* otherwise, i.e., if no message is waiting to be received.
5. **SHUTDOWN**: a request to shut down the mqI process, severing communication with the other side of the brain.

While it is possible to implement interprocess communication routines directly in the Lisp and Python brain components, the use of auxillary processes facilitates more easily the connection of heterogeneous systems in general. In many languages, directly accessing message queues requires the using a foreign-function interface, which is generally more complicated than creating a subprocess and performing I/O with its associated descriptors.

More importantly, with separate communication processes, the architecture does not depend on the use of SNePS for the higher brain, or on Python for the lower brain. We recognize that these components, whatever they are, may be heterogeneous. Keeping the communication aspects separate from the brain reduces the burden on the brain.

Experiments

To test the feasibility of the `snarpy` architecture, we have developed two cognitive robot control systems. The first demonstrates the way in which an existing Pyro "brain" — `Avoid.py`, included with Pyro — can be mapped onto the proposed architecture and involves simple obstacle avoidance with a simulated Pioneer robot in a two-dimensional

world. The second involves using a single brain to evoke the same behavior in two different robots: (a) the simulated Pioneer and (b) a real AIBO dog.

Obstacle-avoid

The *obstacle-avoid* system ultimately controls a simulated Pioneer robot. The agent's basic cognitive process is simply to repeat forever three sequential steps:

1. Decide on a direction to move
2. Move in that direction for a short period of time
3. Forget which direction you are moving

The knowledge level consists of the following logical assertions:

- Facts about what motion directions are possible:

```
possibledirection(hardturn).
possibledirection(right).
possibledirection(left).
possibledirection(straight).
```

- Two rules associated with moving: (1) a plan for performing a `move` action is to find some `?x` such that `shouldgo(?x)` is currently believed, and perform the `?x` action. If no such `?x` can be found, signal an error. (2) to `forgetdirection` means to disbelieve every possible proposition of the form `shouldgo(?x)`.

```
ActPlan(
  move,
  withsome(?x,shouldgo(?x),
           action(?x),error)).
ActPlan(
  forgetdirection,
  withall(?x,possibledirection(?x),
          disbelieve(shouldgo(?x)),
          error))
```

- Higher-level behavior structures: (1) The act of taking a single `avoid step` is to do, in sequence, the three defined actions in the process. (2) The entire (infinite) `avoid` act is to perform an `avoidstep` and repeat the `avoid`.

```
ActPlan(
 avoidstep,
 snsequence(decidedirection,
  snsequence(move,forgetdirection))).
ActPlan(avoid,
        snsequence(avoidstep,avoid)).
```

The PMLa contains the following code. In the `decidedirection` primitive action, there are four possible motion decisions. We ask PMLb for the distances to the obstacles in front of the robot. Then, based on the answers, we add one of four possible motion decisions to the KL.[1]

```
(define-primaction decidedirection ()
 (let
  ((f (ask-distance "front"))
   (fl (ask-distance "left-front"))
   (fr (ask-distance "right-front")))
  (cond ((< f 0.5)
         #!((add shouldgo hardturn)))
        ((< fl 0.8)
         #!((add shouldgo right)))
        ((< fr 0.8)
         #!((add shouldgo left)))
        (t
         #!((add shouldgo
                 straight))))))
```

Each of the primitive actions is defined simply in terms of a single PMLb-level operation: `basic-move-step`, which takes translation and rotation velocities (in robot-units). The remaining PMLa primitive actions are shown below:

```
(define-primaction hardturn ()
  (basic-move-step 0.0 0.3))

(define-primaction right ()
  (basic-move-step 0.1 -0.3))

(define-primaction left ()
  (basic-move-step 0.1 0.3))

(define-primaction straight ()
  (basic-move-step 0.5 0.0))
```

The PMLb, shown below, contains the code for the `basic-move-step` function along with the code for communicating with the PMLc (Pyro program).

```
(defun basic-move-step (trans rot)
  (tell-move trans rot)
  (sleep 0.25)

(defun ask-distance (direction)
  (robotipc:send "DISTANCEQ")
  (robotipc:send direction)
  (read-from-string
    (robotipc:receive)))

(defun tell-move (trans rot)
  (robotipc:send "MOVE")
  (robotipc:send
    (format nil "~A" trans))
  (robotipc:send
    (format nil "~A" rot)))
```

The PMLc level is implemented as a Pyro brain. The `step` function is automatically called by the Pyro system 10 times per second. The `processDirective` function receives and acts upon messages from from PMLb: The `DISTANCEQ` message computes and answers the smallest distance from the robot to an obstacle along the requested vector (`front, left-front`, or `right-front`). The

[1] The `#!((...))` lines are written in traditional SNePS User Language rather than in the SNePSLOG language employed elsewhere in this paper.

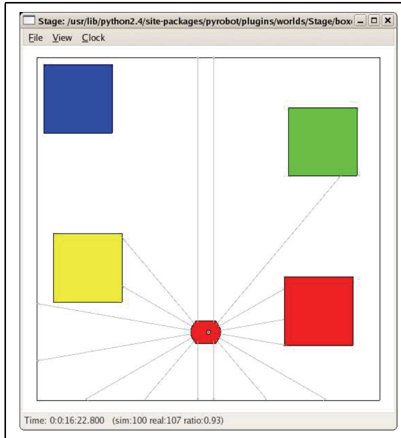

Figure 3: Simple world created with the Stage simulator and the Pioneer robot

MOVE message sets the physical robot in motion according to received translation and rotation velocities.

```
def processDirective(self):
  preface = robotipc.receive()
  if preface == "DISTANCEQ":
    vector = robotipc.receive()
    distance =
     min([s.distance()
         for s in
         self.robot.range
          [vector]])
    robotipc.send(str(distance))
  elif preface == "MOVE":
    trans = float(robotipc.receive())
    rot = float(robotipc.receive())
    self.robot.move(trans,rot)

def step(self):
  if robotipc.query():
    self.processDirective()
```

With all the layers in place, the simulated robot is set up in an environment with walls and other obstacles. After the user issues a `perform avoid` request to the KL, the robot runs around forever in the room, turning as needed to the left or right to avoid the obstacles. Figure 3 illustrates a simple world in which the Pioneer avoid brain can be effectively run.

AIBO-cha-cha

The *cha-cha* system can be used, without changes, to control either a simulated Pioneer or a real AIBO robotic dog. This is due to the fact that the system only involves forward and backward movement and does not entail processing any sensor data. Note that the sensory capabilities of the AIBO and Pioneer are very different and would have to be taken into consideration in terms of the types of interactions the robot might have with its environment. The purpose of this experiment was simply to test the effects of running the same brain on two very different robot platforms.

The agent's basic cognitive process is simply to do the cha-cha five times, where a cha-cha consists of a *cha1* (move forward one quarter robot length and pause) followed by a *cha2* (move backward one quarter robot length and pause).

In the simulated world the Pioneer can be observed moving foward and backward five times. Similarly, when the AIBO dog controlled with the same brain (at all levels) the dog moves foward and backward five times.

This experiment demonstrates the utility of Pyro as part of the `snarpy` architecture.

Conclusion

We have introduced a new member of the GLAIR family of cognitive robotics architectures. This work outlines the principles of `snarpy` agents.

The architecture presented herein is not novel, rather, it is the use of this architecture to connect a sophisticated KR system to Pyro which is the primary contribution. In terms of education, Pyro is being used by an increasing number of colleges and universities in introductory AI courses to teach a variety of concepts, from vision and neural networks to basic robotics. Such courses often include instruction on knowledge representation. The framework we have presented can easily be adapted to allow for the integration of any KR system with Pyro robotics, thus facilitating the integration of disparate instructional components of such courses and letting students experience the importance of KR in "real-world" processing via robots. Traditional AI courses may not include robotics or vision, but could quite easily do so by using the framework we describe in this paper. Robotics has the potential to energize and excite students in terms of future study of computer science in general and AI in particular: the system we present is easily accessible to students at many levels and to faculty without previous robotics experience. The simulated robots and worlds in Pyro allow for the use of the framework in the complete absence of real-world robots.

Pyro has received most attention in terms of teaching, but it is also useful for research robotics, particularly cognitive robotics, as it provides a cognitively motivated set of primitives that allow researchers whose focus is not on robot hardware to easily interface to actual robots. We have attempted to show some of the features of Pyro by demonstrating the use of the same Pyro brain in both a simulated Pioneer and an Aibo dog. The examples we have presented are simple, but they help to convey the potential power of the approach.

To better demonstrate the capabilities of this particular architecture (namely, the linkage of SNePS with Pyro), future agents we develop must have increased capabilities at the knowledge level to make conscious decisions about actions and to forget about—more than merely disbelieve—previously asserted propositions. Our next agents will connect to more interesting sensor and actuator interfaces in the Pyro framework such as vision. By using SNePS at the top of a layered architecture, Pyro cognitive robotic agents will

be able to reason, act, and respond to human queries about what they know, what they have seen and what they have done.

References

Arkin, R. C. 1998. *Behavior-Based Robotics*. MIT Press.

Blank, D.; Kumar, D.; Meeden, L.; and Yanco, H. 2006. The Pyro toolkit for AI and robotics. *AI Magazine* 27(1). special issue on robots and robotics in education.

Blank, D. S.; Meeden, L.; and Kumar, D. 2003. Python robotics: An environment for exploring robotics beyond legos. In *ACM Special Interest Group: Computer Science Education Conference*.

Chalupsky, H., and Shapiro, S. C. 1994. Sl: A subjective, intensional logic of belief. In *Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society*, 165–170. Hillsdale, NJ: Lawrence Erlbaum.

Hexmoor, H.; Lammens, J.; and Shapiro, S. C. 1993. Embodiment in GLAIR: a grounded layered architecture with integrated reasoning for autonomous agents. In Dankel II, D. D., and Stewman, J., eds., *Proceedings of The Sixth Florida AI Research Symposium (FLAIRS 93)*, 383–404. The Florida AI Research Society.

http://www.pyrorobotics.org. Pyro Website.

Johnson, F., and Shapiro, S. C. 2005a. Dependency-directed reconsideration: Belief base optimization for truth maintenance systems. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*. Menlo Park, CA: AAAI Press.

Johnson, F., and Shapiro, S. C. 2005b. Improving recovery for belief bases. In Morgenstern, L., and Pagnucco, M., eds., *Proceedings of the Sixth Workshop on Nonmonotonic Reasoning, Action, and Change*. Edinburgh, Scotland: IJCAI.

Kumar, D., and Shapiro, S. C. 1994a. Acting in service of inference (and vice versa). In Dankel II, D. D., ed., *Proceedings of The Seventh Florida AI Research Symposium (FLAIRS 94)*, 207–211. The Florida AI Research Society.

Kumar, D., and Shapiro, S. C. 1994b. The OK BDI architecture. *International Journal on Artificial Intelligence Tools* 3(3):349–366.

Kumar, D. 1996. The SNePS BDI architecture. *Decision Support Systems* 16(1):3–19.

Levesque, H.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. 1997. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming* 31:59–84.

Shapiro, S. C., and Ismail, H. O. 2003. Anchoring in a grounded layered architecture with integrated reasoning. *Robotics and Autonomous Systems* 43(2–3):97–108.

Shapiro, S. C., and Kandefer, M. 2005. A SNePS approach to the wumpus world agent or cassie meets the wumpus. In Morgenstern, L., and Pagnucco, M., eds., *IJCAI-05 Workshop on Nonmonotonic Reasoning, Action, and Change (NRAC'05): Working Notes*, 96–103. Edinburgh, Scotland: IJCAI.

Shapiro, S. C., and Rapaport, W. J. 1992. The SNePS family. *Computers & Mathematics with Applications* 23(2–5):243–275.

Shapiro, S. C.; Rapaport, W. J.; Kandefer, M.; Johnson, F. L.; and Goldfain, A. forthcoming. Metacognition in SNePS. *AI Magazine*.

Shapiro, S. C. 1979. The SNePS semantic network processing system. In Findler, N., ed., *Associative Networks: The Representation and Use of Knowledge by Computers*. New York: Academic Press. 179–203.

Shapiro, S. C. 1993. Belief spaces as sets of propositions. *Journal of Experimental and Theoretical Artificial Intelligence (JETAI)* 5(2–3):225–235.

Shapiro, S. C. 1998. Embodied cassie. In *Cognitive Robotics: Papers from the 1998 AAAI Fall Symposium, Technical Report FS-98-02*. Menlo Park, California: AAAI Press. 136–143.

Touretzky, D. S., and Tira-Thompson, E. J. 2005. Tekkotsu: A framework for aibo cognitive robotics. In *AAAI*, 1741–1742.