An Intelligent Agent Architecture In Which to Pursue Robot Learning

R. Peter Bonasso and David Kortenkamp

The MITRE Corporation
1120 NASA Road 1
Houston , TX 77058
korten@aio.jsc.nasa.gov

Abstract

This paper describes a multi-layered, intelligent agent software architecture, developed for mobile and undersea robot applications in the defense sector, and to provide tele-autonomy to space-based manipulator robots. The architecture has a deliberative layer which uses a state-based planner, a middle layer for sequencing partially ordered plans using robot skills, and a lower layer repertoire of continuous robot skills. The system has been shown to provide a higher level of human supervision that preserves safety while allowing for task level direction, reaction to out-of-norm parameters, and human intervention at all levels of control. For this workshop, we hypothesize that the architecture is a useful framework in which to explore learning techniques. In particular, we outline techniques appropriate to learning within a given layer, techniques for migrating competences from higher to lower layers, and overall system adaptation from its interaction with the environment. Examples are reinforcement learning for tuning individual skills, case-based techniques to improve the re-planning capability of the deliberative layer, and chunking or explanation-based learning to migrate new strategies created by the planner into standard procedures for the sequencing level.

Background and Motivation

Since the late eighties we have investigated ways to combine deliberation and reactivity in robot control architectures [Sanborn et al 1989, Bonasso 91, & Bonasso et al 92], in order to program robots to carry out tasks robustly in field environments. Field environments are those in which events for which the robot has a response can occur unpredictably, and wherein the locations of objects and other agents is usually not known with certainty until the robot is carrying out the required task.

A robot control software architecture, developed at MITRE is an outgrowth of several lines of situated reasoning research in robot intelligence [Firby 89, Gat 91, Connell 91, Slack 92, Yu et al 94, Elsaesser & Slack 94], and has proven useful for enabling mobile robots to accomplish tasks in field environments. This architecture separates the general robot intelligence problem into three interacting pieces (see Figure 1, below, Figure 2 is discussed in the section on learning):

o A set of robot specific reactive skills. For example, grasping, object tracking, and local navigation. These are tightly bound to the specific hardware of the robot and must interact with the world in real-time.

o A sequencing capability which can differentially activate the reactive skills in order to direct changes in the state of the world and accomplish specific tasks. For example, exiting a room might be orchestrated through the use of reactive skills for door tracking, local navigation, grasping, and pulling.

o A deliberative planning capability to reason in depth about goals, resources and timing constraints.

These capabilities allow a robot, for example, to accept guidance from a human supervisor, plan a series of activities at various locations, move among the locations carrying out the activities, and simultaneously avoid danger and maintain required resource levels.

We have been successful in applying this architecture to mobile land [Bonasso et al 92] and undersea robots [Bonasso & Barrett 93] in the defense sector, and to mobile, two-armed manipulator systems in support of NASA. We believe the architecture is developed to the point where it can be used as a framework for integrating the work of other AI disciplines such as spoken language techniques and machine learning. This paper discusses the architecture from the standpoint of how various learning techniques might be accommodated.

The next section details the architecture in each of its three layers. The subsequent section discusses ideas for applying machine learning techniques to the architecture.

Architecture Discussion via Space Examples

We have recently been using the architecture as a framework for controlling a two-armed manipulator robot maintaining a space station from the ground. The idea is that an intelligent ground control station can enable a ground crew to supervise the routine maintenance activities of the robot, and thus allow the on-orbit personnel to concentrate on user missions. We use examples from this domain to illustrate the architecture.

The Planner

The planning system is envisioned to be a state-based, non-linear hierarchical planner a la SIPE. The planner we use, known as AP [Elsaesser & MacMillan 91], is a multi-agent planner which can reason about metric time for scheduling, monitor the execution of its plans, and replan accordingly. A typical AP plan operator for carrying out routine maintenance at various sites is:

```
(Operator conduct-inspections-or-repairs

   :purpose
      (sites-inspected-or-repaired ?planner ?list-of-sites)
   :agents :none
   :arguments (
              (?broken-sites
                 (list (get-sites-with-broken-items)))
              (?number-of-broken-sites
                 (length (quote ?broken-sites)))
              (expand   ?number-of-broken-sites
                   (?broken-site)
                   (?broken-site
                      (list
                         (nth *ap-subst-count*
                            (quote ?broken-sites)))))
              (?number-of-sites
                   (length (quote ?list-of-sites)))
              (expand ?number-of-sites (?site)
                      (?site
                         (list
                         (nth *ap-subst-count*
                            (quote ?list-of-sites)))))
              )

     :preconditions ( (at ?planner cl-user::dock) )

     :plot

  ;; The plot is to wake-up, get any tools and replacements,
  ;; inspect or repair each site, then return

     (sequential
        (ready-status ?planner ready)
        (expand ?number-of-broken-sites
              (?broken-site)
              (ready-for-site-repair ?planner
                 ?broken-site))
        (attached-to ?planner none) ;;detached
           (expand ?number-of-sites (?site)
              (inspected-or-repaired ?planner ?site))
        (status ?planner docked)
      )

   :effects ( (sites-inspected-or-repaired ?planner
              ?list-of-sites) )
   )
```

This operator matches a goal to inspect or repair a list of sites. The arguments invoke functions which query either the planner's world model (a frame system of CLOS objects) or the dynamic memory of the sequencing tier of the architecture. The planner expands this operator by searching for other operators whose purpose unifies with the various propositions in the plot. For example a number of operators have the (inspected-or-repaired ?planner ?site) purpose depending on the preconditions and arguments, e.g.,

```
(Operator    repair-site-with-broken-item-needing-
                   replacement-only

   :purpose (inspected-or-repaired ?planner ?site)
   :agents :none
   :arguments (
              (?site-item
                 (first    (cl-user::get-all-items-at-except
                               ?site 'cl-user::pdgf)))
              (truth (cl-user::memory-ask
                        `(cl-user::class ?site-item
                          fire-extinguisher cl-user::true)))
              (?replacement
                 (get-broken-item-replacement
                    ?site-item))
              )
  :preconditions ( (status ?site-item off-nominal) )
  :plot

     (sequential
           (item-in ?replacement oru-pouch)
           (at ?planner ?site)
           (attached-to ?planner ?site)
           (repaired ?planner ?site-item)
           (attached-to ?planner none))

   :effects ((inspected-or-repaired ?planner ?site) )
   )
```

which describes how to repair fire-extinguishers by using a suitable replacement.

The repairs themselves have several standard procedures which the planner need not keep track of in detail. But the planner needs knowledge of these to determine the resources needed and the nominal times required. For example, the hard-repair-at-site operator (next page) shows the need for a manipulator to stabilize the robot while repairing heavy items, and to use a wide-field of view vision agent.

Each primitive operator (e.g. fix-item-with-arm on the next page) has a user-supplied function that estimates the time to complete the operation which is used for propagating earliest and latest start and end times throughout the resulting plan tree.

```
(Operator hard-repair-at-site
        :purpose (repaired ?planner ?site-item)
        :arguments (
                        (?site (get-site-from-memory
                                ?site-item))
                        (?weight
                           (get-size-from-memory ?site-item))
                         (truth (>= ?weight 20)) )
        :constraints ( (and
                        (member ?arm-1 '(left-arm right-arm))
                        (member ?arm-2 '(left-arm right-arm)))
                        (check-arm-for-tool-and-strength
                           ?site-item ?weight ?arm-1)
                        (eq
                           (gsv ?vision-agent 'field-of-view)
                              'wide) )

       :preconditions ( (attached-to ?planner ?site) )

        :plot (sequential
                (grappled ?arm-2 ?site)
                (simultaneous
                   (examined ?vision-agent ?site-item)
                   (fixed ?arm-1 ?site-item) )

        :effects (
                     (repaired ?planner ?site-item)
                     (arms-status ?planner unfolded)
    )
    )


(Operator fix-item-with-arm
        :purpose (fixed ?arm ?site-item)
        :arguments (
                        (?site
                           (get-site-from-memory ?site-item)) )
        :preconditions ( (attached-to ?planner ?site) )
        :effects (   (fixed ?arm ?site-item)
                     (arms-status ?planner unfolded) )
        :task-time duration-of-fix-item-with-arm
        )
```

The Sequencer

We are using a new version of Firby's Reactive Action Packages (RAPs) [Firby 93] as the instance of our sequencing system to encode routine behavior as sequences of situated skills. The RAP interpreter uses a library of RAPs to decompose sequences of behaviors to accomplish a task. The system can quickly transform a planner-directed task (i.e., primitive AP operator) into a context specific sequence of skills (which may be run concurrently) by caching solutions to common tasks. The richness of the RAP system can be seen from the following examples.

A basic repair RAP used in the routine maintenance activity invoked by the fix-item-with-arm operator of the planner (the sequencer will use the suggested arm as a recommendation) is

```
(define-rap (repair-antenna ?item)
  (succeed    (and     (location ?item external)
                       (on-off ?item on)
                       (working-status ?item operational)))
  (preconditions (class ?item antenna true))
  (method m-1
        (context    (and     (arm-place ?arm ?someplace)
                             (not (= ?arm foot))
                             (not (arm-holding ?arm ?any))))
        (task-net
         (t0    (arm-pickup ?arm ?item)
                ((arm-holding ?arm ?item) for t1))
         (t1    (arm-toggle-p ?arm ?item) (for t2))
         (t2    (eye-examine-p ?arm ?item)
                ((on-off ?item on) for t3))
         (t3    (wait-p 15) (for t4))
         (t4    (arm-putdown-at ?arm ?item external) (for t5))
         (t5    (eye-examine-p external ?item) (for t6))
         (t6    (put-away-tool ?arm)))))
```

The robot must turn the antenna (the toggle operation is generic for turn or turn on/off) until it reaches its desired orientation (usually takes 15 seconds). This RAP will succeed when the antenna is on and operational (which will be its state when the antenna is properly oriented). The RAP only works with items of the antenna class. It consists of a single method which is invoked when there is a free arm to use, and involves seven sequential steps (each of which has its own RAP definition). The RAP interpreter insures the proper order and the critical preconditions of subsequent steps by use of the "for" clause after each subtask invocation. The RAP interpreter will instantiate the m-1 method for each arm that is available. If the method fails for that arm, it will try the other. Further, it will try each method twice (a user parameter), before giving up.

The memory query in the succeed clause will usually be made true by the firing of memory rules associated with the RAPs in the sub tasks. Memory rules are associated with each RAP as shown in the mort-turnto-angle example (next page) from one of our mobile robots named Mortimer.

The RAP first enables the robot's primitive turning skill, providing the desired angle, turning speed and accuracy factor. Then it enables the primitive event mort_at_angle which waits until the robot has turned to the desired angle and returns the actual angle. Concurrently, a primitive RAP, which is a Lisp function, informs the user of what is happening via a speech channel. The turning action and the speech act must occur before task t4 where the turning action is disabled; the second speech act can occur concurrently with t4. The lack of a succeed clause allows the RAP to execute once and then return as if (succeed t) were the case.

```
(define-rap  (mort-turnto-angle ?angle ?velocity
                    ?sensitivity)
   (method m1
            (task-net
               (t1   (mort_turnto ?angle ?velocity ?accuracy)
                     (wait-for
                        (mort_at_angle ?angle ?accuracy
                                 ?realangle)
                           :succeed (at-angle ?realangle))
                        (for t3)(concurrent-with t2)(for t4))
               (t2   (host-dospeek "mort turning" 3 t)(for t3))
               (t3   (host-dospeek "mort at angle" 3 t)
                        (concurrent-with t4))
               (t4   (mort_turnto_disable)))))
```

```
(define-memory-rule (mort-turnto-angle ?angle ?velocity
                              ?sensitivity) :event
  (match-result
      ((at-angle ?realangle)
         (rule (   (= ?realangle 90.0)
                  (mem-add (mort-direction EAST)) ) ) ) ) )
```

The memory rule fires when an event executing in the RAP returns a :succeed clause matching the result expected for that rule. Arbitrary Lisp functions can be invoked, but the special rule form allows the updating of the RAPs memory in a principled fashion. The following RAP used with a manipulator controlled from a ground station (i.e., with communications delays) shows the use of the memory rule to allow the RAP to complete:

```
(define-rap (arm-move ?arm ?place)
  (succeed (arm-place ?arm ?place))
  (preconditions (current-mode joint-immediate))
  (method pdgf-approach
       (context    (= ?place pdgf-approach))
       (task-net
        (t1    (move-current-arm 1)
               (wait-for (arm-moving ?speed) :succeed
                     (arm-moving ?speed))  (for t2))
        (t2    (no-op)
               (wait-for (arm-not-moving ?place) :succeed
                     (arm-not-moving ?place)) (for t3))
        (t3    (disable-move-current-arm ?place))))
  (method truss-approach
       (context    (= ?place truss-approach))
       (task-net
        (t1    (move-current-arm 2)
               (wait-for (arm-moving ?speed) :succeed
                  (arm-moving ?speed)) (for t2))
        (t2    (nop)
               (wait-for (arm-not-moving ?place) :succeed
                  (arm-not-moving ?place))
               (for t3))
        (t3    (disable-move-current-arm ?place)))))
```

```
(define-memory-rule (arm-move ?arm ?place) :event
  (match-result
    ((arm-not-moving ?place)
        (rule     (t
                      (mem-del (arm-place ?arm))
                      (mem-add (arm-place ?arm ?place)))))))
```

Here, an event is enabled to determine that the arm has started moving (the primitives have time-outs which can trigger other memory rules if needed). Afterwards, a NO-OP RAP (a null Lisp function) is enabled in order to enable an event which waits for the arm to cease moving, at which time the memory rule will fire asserting the place of the arm, which will allow the RAP succeed clause to become true.

The Skill Level

The sequencer's job is to coordinate the dynamic activation and deactivation of situated skills in order to configure the reactive layer for the task at hand. There are three types of skills: primitive actions (and their disabling counterparts), primitive events and primitive queries (queries were designed to appear as normal memory requests, when in fact they query the robot device itself). An example of each is shown below:

```
(define-primitive-action (move-current-arm ?place)
  (enable (:move_current_arm (:place . ?place))))

(define-primitive-action (disable-move-current-arm ?place)
  (disable :move_current_arm ?place))

(define-primitive-event
       (arm-init-mode ?mode1 ?mode2 ?which-mode)
  (event-definition
          (:arm_init_mode
             (:qstate1 . ?mode1)(:qstate2 . ?mode2)))
  (event-values :bound :bound :unbound))

(define-primitive-query (selected-mode ?mode)
  (query-definition (:selected_mode))
  (query-values :unbound))
```

Each skill is written in the language of the robot computer (both C and Rex languages have been used) and has a set of inputs, outputs, states and parameters. The inputs of actions, events and queries can only come from other actions, and the outputs of actions can only go to other events or actions. Event and query outputs consist of both a true/false output as well as any values to be returned. Thus to find out the status of the result of an action, an event or query skill must be constructed. Parameters are settings passed by the RAP upon invocation, such as accuracy of a turning angle.

The move-current-arm action above has a single parameter, the arm to be moved. The arm-init-mode event checks for the occurrence of one of two initialization modes and returns the mode detected. The select-mode query returns the current selected mode of the robot device.

In our experience with mobile and manipulator robots, approximately 20 - 25 skills make up a sufficient skill base from which to design RAPS to exercise the total robot capability. These include one or two D-skills -- skills to gather status information directly from the robot hardware, and the rest C-skills, or those which define the low-level competence of the robot.

Learning

Figure 2 depicts our instantiation of the complete architecture. The planner invokes partially ordered plans which are then specifically ordered and reordered by the sequencer's RAP interpreter, based on the actual environment. The interpreter in effect specifies for any phase of a task the skill set which when invoked will bring about the completion of that phase. The interpreter also maintains a dynamic memory of key states of the robot and the world used to invoke various RAPs. The planner, when notified by the interpreter of the completion of a planned activity, queries the same memory to determine the progress of the plan and invokes replanning when necessary.

We hypothesize three different ways of applying learning to our architecture. First, learning can be applied within each layer of the architecture to increase that layer's performance and thus the performance of the architecture as a whole. Second, learning can take place across layers, that is, activities that once required planning can, over time, be moved to the sequencer and finally to a skill. Third, learning can be used to alter the response of the architecture to the environment. For example, the sequencer could learn the correct timings for each skill in a reactive package. Each of these three areas are explored in the following subsections.

Learning within layers

Learning within layers can be addressed immediately using known techniques. There are several examples of learning being used to increase a planner's performance, for example, Soar [Laird et al 87] and [Knoblock et al 91]. We can envision a case-based learning system for our planner, especially in repairs to the plan; repairs to a plan can be cached using the current robot context available from the RAP memory.

The feedback that the RAPs system exploits from the low-level skills can be the basis of learning new methods in the sequencing layer, a process of learning by observation (e.g., [DeJong 86]. There is a simple but illustrative example from our manipulator work which was "learned" by the programmer. A RAP for selecting which arm to use is somewhat complex since if the selected arm is not the current arm, a set of memory queries needs to be invoked about the new arm. Additionally, the current arm needs to be placed in a standby mode prior to selecting the new arm. Once in the middle of the day, when an arm was brought on line after being down for repairs, the select-an-arm RAP timed out while waiting for the standby setting to be established for the new arm. This was because when the robot system first starts up (or one of the arms has gone down and must be reinitialized), there is a system state in which no arm has been selected, and further, no commands including mode settings can be accepted until an arm is selected. To solve the problem, the programmer added an additional method to check for that unique state and forego setting the current arm on standby.

There are also many examples of reinforcement learning being used to increase the performance of reactive robot skills, for example [Brooks 89]. In particular we are interested in having our skills learn various parameters that control their behavior. For example, our obstacle avoidance skill has several thresholds for determining whether a certain direction is open or closed. Adjusting these thresholds for each new environment is time consuming. As each layer of the architecture incorporates its own learning algorithms the performance of the architecture as a whole should improve.

Learning across layers

While each layer can incorporate its own learning algorithms fairly easily, interesting issues arise when we attempt to do learning within the architecture as a whole. In this case, we are interested in "migrating" competencies from the deliberative layer down to the skill layer in order to increase performance. It is easy to come up with examples of such learning in people, for example, piano playing takes incredible deliberative attention at first, but becomes automatic with practice. Such learning usually follows a power law where increased repetition leads to increased performance. Several systems, such as Soar and ACT* [Anderson 83] exhibit this kind of learning, but they are single-layer architectures. How can we achieve similar results within a multi-layer system?

There are two possibilities. First, when a plan is successful, the planner can create a new RAP that embodies the actions and variable bindings of that plan and add this RAP to the RAP memory. Now the planner can use that new RAP as an atomic action within a larger plan. This is the chunking approach taken by systems such as Soar, and is akin to explanation based approaches which use reduction on formal representations. The second possibility for learning across layers arises when the sequencer fails to achieve its goal. In this case, the planner is re-invoked and a new plan is developed. At this point, it would be possible for the planner to change the context, preconditions or succeed clauses of the RAP that failed. The planner could also add a new context and method to an existing RAP in order to prevent failure during future executions.

The previous paragraph discusses migrating competencies from planning to sequencing. In addition, competencies can migrate between the sequencing layer and the skills. At this point we have thought very little about how to take the results of sequencing and chunk them into skills. An intermediary approach might be to have skills learn to automatically sequence themselves, based on previous experiences. For example, if the "goto-person" skill is always activated immediately after the "find-person" skill, then the skill layer could begin to automatically activate the former as soon as it is done with the latter, without waiting for specific directions from the sequencer. In this case, the sequencer's roll is to start a skill chain and then interrupt that chain if the robot is supposed to do something out of the ordinary.

Learning about the environment

The sequencer layer in our architecture has the ability to control how often a skill is fired. For example, an obstacle avoidance skill is typically run at the maximum rate, while a skill to check the vision system for results is typically run only every second, since the vision system is slow. Currently, these timings are hard-coded by the programmer. An interesting area of research would be to develop an algorithm that learned some of these timings based on the robot's experience with the pace of the world around it. For example, it may learn that the skill for checking if a person is still behind it only needs to run every couple of seconds, based on its experiences with losing people. As the robot accumulates world knowledge it can better schedule its resources.

Conclusion

While we have not yet started to explicitly address learning in our architecture, we believe that our framework allows for easily incorporating learning at many levels. Here are some features of our architecture that make it suitable for learning research:

1) A layered architecture allows for many different learning mechanisms. Reinforcement learning may be appropriate at the skill level, but not at the deliberative level. Learning by discovery may be appropriate for the sequencer, but not for the planner, while case-based learning may be appropriate for the planner, but not for the sequencer. Thus, each level can make use of the learning algorithms most appropriate to it.

2) The sequencer (RAPs) uses a RAP library that is independent of the main execution cycle. Thus, new RAPs can be built and placed in the RAP library by the planner and they will automatically be invoked by RAPs in the appropriate context. This is possible in part because of the similarities in representations between RAPs and a state-based planner.

3) The sequencer has control of the firing rates of the skills. These rates are currently hard-coded, but could easily be adjusted autonomously.

4) The skill layer allows information to flow between skills without passing through another layer. Thus, skills can be "chained" together to produce larger skill sequences.

As we learn more about our architecture through implementation on more robots, we will certainly identify more areas where learning will help to increase robot performance. In each of these cases we will need to identify appropriate metrics to allow for comparisons between the architecture without learning and the architecture with learning. In this way, we can determine the most appropriate use of learning algorithms within a robot architecture.

References

[Anderson 83] John Anderson. The Architecture of Cognition . Harvard University Press, Cambridge, MA, 1983.

[Bonasso 91] R. Peter Bonasso. Integrating Reaction Plans and Layered Competences Through Synchronous Control. In Proceedings of the 12th International Joint Conference on Artificial Intelligence. Sydney, Australia. Morgan Kaufman.

[Bonasso et al 92] R. Peter Bonasso, H.J. Antonisse, M.G. Slack. A Reactive Robot System for Find and Fetch Tasks In An Outdoor Environment. In Proceedings of the Tenth National Conference on Artificial Intelligence. San Jose, CA. July 1992.

[Bonasso & Barrett 93] R. Peter Bonasso, R.P. & J. Barratt, J. A Reactive Robot System for Find and Visit Tasks in a Dynamic Ocean Environment. In Proceedings of the 8th International Symposium on Unmanned Untethered Submersible Technology, IEEE Oceanographic Society, Sep 1993.

[Brooks 89] Rodney A. Brooks. A Robot that Walks; Emergent Behaviors from a Carefully Evolved Network, Proceedings IEEE Conference on Robotics and Automation, 1989.

[Connell 91] J. H. Connell. A hybrid architecture applied to robot navigation. In Proceedings of the IEEE International Conference on Robotics and Automation, April 1992.

[DeJong 86] Gerald Dejong. An Approach to Learning from Observation. In Machine Learning, An Artificial Intelligence Approach,Vol II, R.S. Michalsky, J.G.

Carbonell, and T.M. Mitchell, eds. Morgan Kaufmann, 1986.

[Elsaesser & MacMillan 91] C. Elsaesser . Representation and Algorithms for Multiagent Adversarial Planning, MTR-91W000207, The MITRE Corporation, Dec 1991.

[Elsaesser & Slack 94] C. Elsaesser & M.G. Slack. Deliberative Planning in a Robot Architecture. In Proceedings of the AAIA/NASA Conference on Intelligent Robots in Field, Factory, Service and Space, March 1994.

[Firby 89] James R. Firby. Adaptive Execution in Complex Dynamic Worlds. PHD Diss. YALEU/CSD/RR #672, Yale University. 1989.

[Firby 93] James R. Firby. Interfacing the RAP System to Real-time Control (forthcoming), University of Chicago.

[Gat 91] Erann Gat. Reliable Goal-Directed Reactive Control of Autonomous Mobile Robots. PhD thesis, Virginia Polytechnic Institute Department of Computer Science, April 1991.

[Knoblock et al 91] Craig Knoblock, Steven Minton and Oren Etzioni. Integration of Abstraction and Explanation-Based Learning in PRODIGY, AAAI-91, 1991.

[Laird et al 87] John Laird , Allen Newell and Paul Rosenbloom. Soar: An Architecture for General Intelligence. Artificial Intelligence 33(1), 1987.

[Sanborn et al 1989] Jim Sanborn, B. Bloom, and D. McGrath. A Situated Reasoning Architecture for Space-based Repair and Replace Tasks, In !989 Goddard Conference on Space Applications of ARtificial Intelligence, Greenbelt, MD. NASA Pub # 3033.

[Schoppers 89] 6. Marcel J. Schoppers. In Defense of Reaction Plans As Caches. AI Magazine, 10(4): 51-60, 1989

[Slack 90] Marc G. Slack. Situationally Driven Local Navigation for Mobile Robots, JPL Pub. 90-17, 1990. NASA.

[Slack 92] M. G. Slack. Sequencing formally defined reactions for robotic activity: Integrating {RAPS} and {GAPPS}. In Proceedings of the SPIE Conference on Sensor Fusion, November 1992.

[Yu et al 94] S. Yu, M. G. Slack, and D. P. Miller. A streamlined software environment for situated skills. In Proceedings of the AAIA/NASA Conference on Intelligent Robots in Field, Factory, Service and Space, March 1994.