Distributed Programming and Coordination for Agent-Based Modular Automation

Alfred A. Rizzi, Jay Gowdy, and Ralph L. Hollis

The Robotics Institute
Carnegie Mellon University
{jayg,arizzi,rhollis}@ri.cmu.edu

Abstract

A promising approach to enabling the rapid deployment and reconfiguration of automated assembly systems is to make use of cooperating, modular, robust robotic agents. Within such an environment, each robotic agent will execute its own program, while coordinating with peers to produce globaly cooperative precision behavior. To simplify the problem of agent programming, the structure of those programs is carefully designed to enable the automatic encapsulation of information necessary for execution during distribution. Similarly, the programming model incorporates structures for the compact specification and robust execution of potentially complex and fragile cooperative behaviors. These behaviors utilize a run-time environment that includes tools to automatically sequence the activities of an agent. Taken together, these abstractions enable a programmer to compactly describe the high-level behavior of the agent while relying on a set of formally correct control strategies to properly execute and sequence the necessary continuous behaviors.

1. Introduction

Most robot programming approaches are based on standard computer languages, with the addition of special primitives, constructs, and libraries to support the physical control of a robot [1]. These languages are designed to enable the control of a single robot, and do not inherently support distributed systems of robots. More abstract programming models have recently appeared [2, 3], but typically these are either "task" or "process" based and are applied to the programming of work cells (which might contain multiple robots). This paper explores the distributed programming model we are developing for use with the Architecture for Agile Assembly [4, 5], an ongoing project in the Microdynamic Systems Laboratory at Carnegie Mellon University's Robotics Institute (for additional information see `http://www.cs.cmu.edu/~msl`).

The overall goal of the Architecture for Agile Assembly (AAA) is manufacturing agility – enabling both the rapid deployment and rapid reconfiguration of automation systems – facilitating the early delivery of a product to market and the ability to adapt to changing technologies and market needs. AAA strives to achieve this form of agility by utilizing modular robust robotic *agents* [6]. Agents are mechanically, computationally, and algorithmically modular mechanisms which operate in a deliberately limited domain, but possesses a high degree of capability within that domain. For the remainder of this paper we will focus on *minifactory*, o specific instantiation of AAA, designed to facilitate four-degree-of-freedom (4-DOF) assembly of high-value, high-precision electro-mechanical products (see Figure 1). Minifactories contain agents (called couriers) that are "experts" in product transport and local planar manipulation, and other agents (called manipulators) that are "experts" at vertical insertion and part rotation. Through cooperative group action these agents perform the 4-DOF operations required to produce a product.

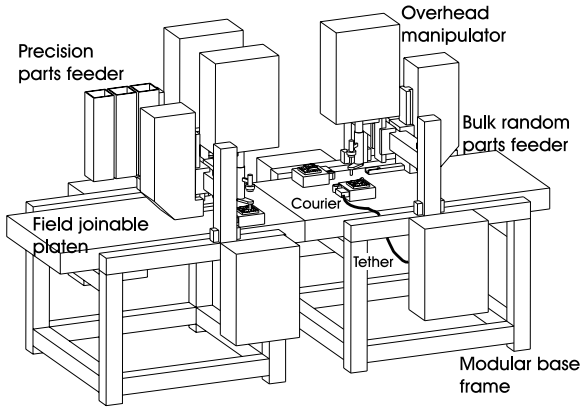

Figure 1: A minifactory segment

Modularity is a central philosophical concept in AAA; not only does it enable scaling of the factory system, it also offers the potential for improved system

robustness by eliminating single point failures. Unfortunately it also presents new and unique challenges for the system programmer. As there is no central factory "brain" and thus no single program for an entire AAA factory, each agent must execute its own program which must reliably interact with those of its peers in order to give rise to the desired overall system behavior. Whereas constructing generic distributed systems of this form is a difficult problem, it is our conjecture that by restricting our focus to a small but important class of robotic systems, sufficient constraints are placed on the problem to make it tractable.

Toward this end, we have developed a model for agent programs that seeks to simplify the distributed programming problem: *i)* agent programs are self contained "applets," that utilize standard protocols to synchronize and communicate about their behavior; *ii)* each agent makes use of a high-performance hybrid control system to manage its continuous behavior. The intent of this programming model is to provide a high level of expressiveness and flexibility for the agent programmer (be it a human designer or automated code generating tool) while ensuring a structured and reliable interface to the underlying control systems.

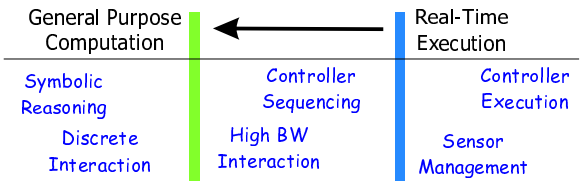

Figure 2: Shift of control responsibility for distributed programming.

One fundamental challenge for any robot programming system is that the "program" must specify both a system's discrete behavior as well as its continuous behavior – *i.e.*, it must coordinate both the continuous domain of the robot's control system and the discrete domain of event management associated with overall behavior. Most approaches to robot programming make a sharp distinction between these domains – where the continuous, state-based view is relegated to the execution of controllers, and all decisions about which controllers to run and when to run them are made by higher-level systems using a discrete, event based view. We choose to place this distinction at a slightly higher level and make it a more formal abstraction barrier than is normal. Figure 2 depicts a course decomposition of the continuum of tasks to be undertaken by a robot's control system, making explicit our intent to move the "barrier" between the real-time and general purpose tasks. In AAA, an agent's execution is divided into two distinct layers: a higher-level discrete layer responsible for the abstract semantics of factory operation, and a lower-level continuous layer that manages the sequencing and executing of specific control laws which, in turn, influence the physical environment of the agent. The critical distinction is that the continuous mechanisms are used to guide the transitions between controllers as well as to run the controllers, freeing the high-level agent programs to deal with the more relevant and abstract problem of deciding what to do and how to do it. This notion of automatically managing the transition between controllers was introduced in [7], abstractly fit to the domain of minifactory in [8], and recently demonstrated experimentally [9].

2. Distributed Programming

The distributed and cooperative nature of AAA has implications on the form of agent programs: an agent program is not simply a script, but rather defines an instance of a class that implements a number of specific methods. The program may define a new class to be instantiated, or a subclass from a pre-existing standard one, but the class must implement a standard interface. This concept is very similar to the Java "applets" that are used in world-wide web programming. For reasons including ease of porting and licensing issues, we have chosen Python[10], another object-oriented language which can be interpreted or byte-compiled, to program our robotic agents rather than Java.

Every robotic agent program must provide two methods: `bind` and `run`. They encapsulate the two major conflicting requirements of an agent program – it must specify behavior in reference to external entities, but must run in a completely distributed fashion and cannot rely on any centralized resource or database during execution. For example, a courier must be able to know it will be interacting with a particular manipulator, but the information on how to contact that manipulator must reside with the courier at run time. Similarly, a manipulator may need to know it will get parts of a specific type from a specific parts feeding device without having to contact a central database at run time to get the geometric and product-specific information it needs to perform its operations.

In the AAA environment, an agent program has two distinct phases in its life cycle. First, it is written and simulated within a centralized interface and design tool. This tool provides a factory developer with a global view of the factory system under development [11]. When executing within the centralized simulation environment, the `bind` method simply causes the relevant items to be looked up in the simulation database before proceeding to execute the `run`

```
# Agent class definition
class Program(CourierProgram):
  # Binding method
  def bind(self):
    # superclass has some binding to do
    CourierProgram.bind(self)

    # Bind to a particular manipulator
    self.source = self.bindAgent("FeederManip")
    # Bind to a particular factory area
    self.corridor = self.bindArea("CorridorA")

  # Execution method
  def run(self):
    # initialize the movement
    self.startIn(self.corridor)

    # block until manipulator is ready
    self.initiateRendezvous(self.source, "Feeding")

    # move into the workspace
    self.moveTo(self.sourceArea)

    # coordinate with manipulator to
    # get product from it
    self.acceptProduct()

    # The coordinated maneuver is done
    self.finishRendezvous("Loading")

    # move out of the workspace
    self.moveTo(self.corridor, blocking=1)

# instantiate the applet
program = Program()
```

Figure 3: A simple courier program

method. The second phase occurs when the factory developer downloads an agent program from the simulation environment to the physical agent. At this point, the agent program must be be "bound" with all of the global factory information the agent will require while executing the program. To bind a program, the interface tool executes that program's `bind` method, and uses the results to construct a small database containing the information necessary for the agent to locate, both geometrically and logically, all of the factory elements it will interact with. This small database serves as a starting point for an agent's self-initialization and exploration of its environment. For example, in the sample courier program (Figure 3) the `bind` method calls `bindAgent("FeederManip")`, which declares that the agent program wants to know about the manipulator named *FeederManip* and assigns the result of that binding to a local member variable for use in its `run` method. As a result of the invocation, the interface tool will add the relative position of *FeederManip* in the courier's frame of reference as well as the network address of *FeederManip* to the local database which is sent to the courier along with the program text.

The `run` method contains the "script" which actually runs during execution, implementing the discrete logic of the agent which is responsible for initiating and coordinating the behavior of this agent. For example, the `run` method in Figure 4 causes the agent to loop, transferring parts from a parts feeder to couriers that request them. The run method is written using

```
# Agent class definition
class Program(ManipProgram):
  # Binding method
  def bind(self):
    # bind a bulk feeder
    self.feeder = self.bindDescription("ShaftFeeder")
    # bind product information
    self.product = self.bindPrototype("ShaftB")

  # Execution method
  def run(self):
    while 1:
      # convenience function for getting a
      # product from a feeder
      self.getPartFromFeeder(self.product, self.feeder)

      # Wait for a courier to rendezvous
      # with the manipulator for feeding
      partner = self.acceptRendezvous("Feeding")

      # and transfer the product to the courier
      self.transferGraspedProduct(partner)

# instantiate the applet
program = Program()
```

Figure 4: A simple manipulator program.

convenience methods defined by the program's superclasses, which themselves cause the exchange of messages between agents using AAA protocols and the deployment of hybrid-controllers (described in Section 3. below). For example, the convenience method invoked by `self.getPartFromFeeder` is implemented in the parent class, `ManipProgram`. This convenience method extracts information from the product prototype and feeder instance passed into it, and sets up and monitors control policies which will robustly pick a product of that type from that feeder.

3. Control and Coordination

As we have already described, an agent program in AAA has two distinct but related run-time responsibilities: *i*) it must carry out semantic negotiations with its peers to accomplish work on behalf of the factory; and *ii*) it must properly parameterize and sequence the application of low-level control strategies to successfully manipulate the physical world. The programming model we are utilizing simplifies the relationship between these two responsibilities and minimizes their impact upon one another. Specifically, to reduce the complexity associated with writing agent programs, the low-level control strategies are now responsible for the details associated with switching and sequencing the various control policies available at any one moment.

3.1. Real-time control

To simplify the development of agent programs, the process of deciding exactly when and how to switch between low-level control strategies is removed from

the agent program and isolated from the high-level semantic negotiations that are the primary domain of the agent program. However it is important to note that the high-level agent program continues to maintain explicit control over the precise policies that are available for use at any given moment. The fundamental model we utilize for the execution of control strategies was presented in [8]. Briefly, rather than relying on the agent program to generate trajectories through the free configuration space of the agent, the program decomposes the free configuration space into overlapping regions and parameterizing control policies associated with each region. The left side of Figure 5 shows a simplistic "cartoon" rendering of this approach, where Φ_i represent the control policies which are guaranteed to safely move any state from anywhere in the associated shaded domain into the domain of the "next" control policy. A *hybrid control* system is then responsible for switching or sequencing between the control policies associated with this decomposition to achieve a desired overall goal, inducing a monotonically convergent finite state automata over the control policies such as that depicted on the right of Figure 5.

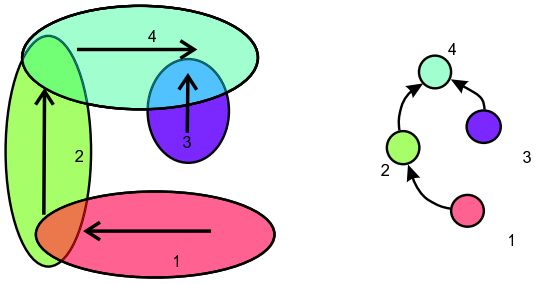

Figure 5: Example decomposition of a trivial planar configuration space, and the associated induced graph relating the control policies.

This scheme describes the behavior of any one agent in terms of a collection of feedback strategies based on the state of the system as perceived by the individual agent. The result is a hybrid on-line control policy (one that switches between various continuous policies) which makes use of the collection of control policies that have been passed to it by the higher-level agent program. By leaving the selection of goals and the associated prioritized decomposition of the state space to the agent program it remains possible to describe (at the program level) arbitrarily complex behavior without constructing code to undertake the complex real-time management of those behaviors.

Given this model for executing physical action, it remains the responsibility of the agent program (specifically the script defined by its `run` method) to create, parameterize, and manage the currently active set of controllers along with the associated sets of goals and domains. Thus the script is only responsible for choosing the current "overall" goal along with appropriate intermediate sub-goals, and providing parameterizations of control strategies to accomplish those goals. The complex and potentially error-prone problem of making real-time changes to the underlying control system is left to the hybrid control system.

The interface between the script and this *controller manager* is quite straightforward. The class from which a particular agent program instance is derived provides standard tools for creating and parameterizing controllers and their associated domains. These resulting controllers are then, at the direction of the script, placed into an ordered list of active controllers. Finally, the controller manager will select the appropriate control policy (from this list) to execute in real-time. The details of high-bandwidth monitoring and coordination of an agent and its peers' state is performed by these lower levels, utilizing a dedicated local communications network to share information between agents. This local network is used to pass relevant information between agents only about those variables that effect their execution, resulting in efficient utilization of the available communication bandwidth in a manner that is transparent to the agent program.

Communication of progress and completion of tasks back to the script is accomplished by use of either call-back functions or direct polling of the actual state of the agent. In general, the expectation is that scripts will submit a moderately-sized list of control actions along with a set of fail-safe and fall-back strategies capable of responding to the most dire circumstances, then *sleep* (wait for a call-back) until either progress has been made or a failure has been detected. When appropriate progress has been made the script will, while motion is still executing, append additional control actions to the "top" of the active controller list indicating new goals and delete those control actions which are no longer useful. If a failure has been detected the program will proceed in a similar fashion; only the actions added to the list will most likely attempt to recover from the problem.

By both parameterizing the specific controllers (setting the goal, defining the domain of applicability, specifying gains, etc.) and ordering their placement on the list of active controllers, a script is able to specify complex and efficient physical motion that is fundamentally robust. This provides a rich and expressive method for programs to specify physical motion while reducing the risks associated with writing those programs. Within the minifactory system, an agent presents a "pallet" of control policies, each with an

```
# submit actions to move from self.current to area
def moveTo(area):
  # get the goal at boundary of area
  # and self.current in self.current
  x,y = self.getBoundaryGoal(area)

  # create and submit action
  controller = self.goTo(x,y)
  domain = self.inArea(self.current)
  self.submit(controller, domain)

  # reserve area, blocking if necessary
  self.reserve(area)

  # get goal at boundary of area and
  # self.current in area
  x,y,overlap = self.getOverlapGoal(area)

  # create and submit action to cross into
  # the new area
  self.submit(self.goTo(x,y), self.inRegion(overlap))

  # create and submit action to drive to the
  # goal in area
  # note that a callback class is invoked when
  # this action starts which unreserves self.current
  self.submit(self.goTo(x,y), self.inArea(area),
              start=Unreserve(self.current))

  # keep track of current area
  self.current = area
```

Figure 6: Code fragment for `moveTo`.

associated domain of applicability, to the programming system. In addition to a standard collection of proportional-integral-derivative (PID) controllers the pallet typically includes other strategies for safely executing large motions in the presence of actuator and geometric constraints or undertaking cooperative assembly tasks with peer agents (*e.g.* for performing visually of force guided coordination).

In practice, the details of this interface are hidden from the programmer by a set of standard "convenience functions." For example the `moveTo(...)` call in Figure 3 would actually expand to the code fragment shown in Figure 6. It is here that a specific resource reservation protocol is implemented to ensure safe operation [6] and where a "standard" set of controllers are parameterized and placed on the list of active controllers. As a simple means to coordinate the overall activity of agents, minifactory makes use of a distributed resource reservation systems to guard against inter-agent collisions. In this particular instance a geometric region in the factory is reserved by the call to `self.reserve`, and a call-back method is registered to execute as the agent enters the destination area. The particular call-back method used here, `Unreserve(...)`, frees the reservation held on the current area as soon as the agent departs it, thus allowing its use by other agents in the system.

3.2. Agent interaction

Thus far we have focused on describing how individual agents can be programmed to accomplish specific

```
def acceptProduct(source):
    # slave to manipulator when it is ready
    controller = self.create("Slave")
    controller.master = source
    predicate = self.create("WatchPartner")
    predicate.is_grasping = True
    self.insert(ControllerAction(controller, predicate))

    # hold position after part placement and
    # manipulator withdrawal
    controller = self.holdPosition()
    predicate = self.create("WatchPartner")
    predicate.is_grasping = False
    predicate.min_height = source.offset
    action = ControllerAction(controller, predicate)
    action.addStartCallback(self.tag("PartPlaced"))
    id = self.insert(action)

    # wait until part placed
    self.waitFor("PartPlaced")
    # and clean up action list
    self.truncate(id)
```

Figure 7: Code for `acceptProduct`.

tasks. Construction of a useful AAA system, however, requires that agents successfully interact with one another to undertake cooperative behaviors. Not surprisingly, our approach to performing these cooperative actions utilizes the same protocols and programing constructs we have been describing. The notable difference is that the execution of the specific control strategies may require sharing of significant state information between the participating agents in order for their behavior to be tightly coordinated.

It makes sense to think of the two such cooperating agents as transiently forming an abstract *machine* consisting of their collective degrees of freedom, and a single program dictating the behavior of this abstract *machine* – a term we will use to describe a collection of agents that are actively coordinating their continuous behavior with one another, as would happen when a minifactory courier and manipulator cooperate to perform a visually guided insertion task. Of course, in actuality each agent will be executing its own program and associated control policies, and it is critical that the participating agents reliably enter into, execute, and dissolve their coordinated behavior. Clearly for this to happen the agents must arrange to execute low-level control strategies which are "compatible" – *i.e.* when executed in parallel they must communicate the appropriate state information to their peer(s) to jointly perform the desired physical operation.

As already described in Section 2. our programming system includes primitive tools to perform the necessary semantic negotiation – *e.g.*, the various forms of `rendezvous` statement in Figures 3 and 4. One simple case, with extremely asymmetric interaction, is a transient master/slave relationship between two agents, with one agent abdicating control of its actuators to the other, which monitors the state of both and passes force and torque commands to its own actuators as

```
def transferGraspedProduct(partner):
    # from the product information and the partner
    # courier attributes calculate the courier
    # position and the manipulator position
    # necessary to place the product on the courier
    (cour_x, cour_y, manip_z, manip_th) = \
             self.findPlacementTransform(partner)

    # submit action to get ready to place
    controller = self.coordinatedMove(cour_x, cour_y,
                        manip_z-self.offset, manip_th)
    self.insert(ControllerAction(controller,
                                 self.isGrasping()))

    # submit action to go down to place product when
    # both courier and manipulator have arrived at
    # pre-placement positions
    controller = self.goto(manip_z, manip_th)
    predicate = self.coordinatedAt(cour_x, cour_y,
                 max_z=manip_z-self.offset, manip_th)
    self.insert(ControllerAction(controller, predicate))

    # submit action to release part when force sensed
    controller = self.releasePart()
    predicate = self.forceThreshold(min_force = 0.5)
    self.insert(ControllerAction(controller, predicate))

    # submit action to back off from courier
    action = ControllerAction(
              self.goto(manip_z-self.offset, manip_th),
                        self.isNotGrasping())
    action.addStartCallback("FinishedPlacement")
    id = self.insert(action)

    # wait for placement to finish
    self.waitFor("FinishedPlacement")
    # register transfer
    self.registerTransferTo(partner)
    # clean up action list
    self.truncate(id)
```

Figure 8: Code for `transferGraspedProduct`.

well as those of the slave agent via a high-bandwidth local network. The details of this interaction, taken from a simulated minifactory, are illustrated in Figure 7 which shows the expansion of the `acceptProduct` procedure from Figure 3, and Figure 8 which shows the expansion of the `transferGraspedProduct` procedure. The `acceptProduct` method abdicates control of the courier's motions to a manipulator master until a product has been placed on it and the manipulator has backed off by a given offset. The `transferGraspedProduct` method coordinates the motions of the manipulator and courier and places the part it has already grasped onto the courier.

Here, the invocation of `acceptRendezvous` and `initiateRendezvous`, by the manipulator and courier respectively, has indicated the willingness of these agents to participate in the cooperative behavior, as well as having forced them to synchronize their execution. Immediately following this a set of control policies are submitted that cause the underlying control systems to synchronize and undertake the desired cooperative behavior. Implicitly, the submission and execution of these control policies (`Slave` and `coordinatedMove`) has created a high-bandwidth communications channel between the two agents. This communication channel is used to enable the supervisory controller manager on each agent to make transitions based on state variables contained within the peer, and to allow relevant state (and in this case command information) to pass between the two control algorithms.

Note that throughout this process the individual controller managers maintain authority over the specific policies that are executed by each agent, and that it is only through the submission of a compatible set of policies by both agent programs that the desired behavior is realized. We do not foresee this extremely asymmetric form of cooperation as the typical behavior of agents, but it serves to illustrate the point: typically the control policy will be segmented between the agents with each individual computing those terms relevant to its own behavior. The manual specification of these interactions can be a complex process. Fortunately a significant percentage of the interactions fall into stereotypic classes for which convenience functions can easily be provided.

4. Conclusions

The specifics of the AAA and minifactory environments have led us to consider a new model for programming distributed automation systems. This model incorporates features that standardize the specification of physical action and regulate interactions between cooperating entities. The ramifications of these features are manifest both in the form and function of the resulting programs, as well as the structure of the run-time environment in which they operate. As a result, agent programs execute in a structured environment with standardized means for coordinating their activity, making it possible to construct complex modular automation systems, while programming them by focussing predominantly on local behavior. We have demonstrated this capability both in simulation (coordinating the activity of upwards of 40 agents), and experimentally (performing precision placement with 3 cooperating agents) [9].

4.1. Future work

There remain many open practical questions about the implications of our programming model. For example, to produce a working factory, users must generate a functional set of distributed cooperating agent programs. Fortunately, the scope of any individual agent is limited, and each agent will include a set of powerful primitives as well as convenience methods to facilitate their use in stereotypical applications. Unfortunately, no matter how short or simple the programs may become as a result of these "libraries," the chore

of generating a semi-custom program for each agent in a system remains. Beyond the potential tedium of generating these programs, the programmer is essentially faced with producing a large, distributed program, with all of the known pitfalls of that domain, such as deadlock and livelock.

To overcome these issues a means of presenting the factory programmer different ways of viewing the programming problem must be developed. For example, a programmer may want to think in terms of a factory-centric view of the problem, where overall product movement through the entire factory may be specified through a work-flow representation, *i.e.* what processes have to occur and in what order. Alternatively, a user may want to consider a product-centric view, where product models are specified and annotated with process information to describe the relative motion between parts independently of the machines used to perform those manipulations. In the future, we envision an AAA programming environment that includes tools which support the smooth transition between these various views of the design and programming problems. Furthermore, they will rely on user-guided search and optimization methods to semi-automatically transform the information provided by these different views of the problem into factory layouts and distributed agent programs.

Regardless of what view the user has of factory programming, agent-centric, factory-centric, or product-centric, ultimately any distributed factory system must execute those programs on a set of agents which interacting with each other and with the product components to perform an assembly task. This paper has presented the programming model and protocols which will form the basic building blocks for future systems which can bring the vision of rapid deployment, reconfiguration, and reprogramming of automated assembly systems closer to reality.

Acknowledgments

This work was supported in part by the NSF through grants DMI-9523156 and CDA-9503992. The authors would like to thank all the members of the Microdynamic Systems Laboratory, and in particular Arthur Quaid, Zack Butler, and Patrick Muir, for their invaluable work on the project and support for this paper.

References

[1] T. Lozano-Perez. Robot programming. *Proceedings of IEEE*, 71(7):821–841, 1983.

[2] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[3] R. W. Harrigan. Automating the operation of robots in hazardous environments. In *Proceedings of the IEEE/RSJ Int'l Conf. on Intelligent Robots and Systems*, pages 1211–1219, Yokohama, Japan, July 1993.

[4] R. L. Hollis and A. Quaid. An architecture for agile assembly. In *Proc. Am. Soc. of Precision Engineering, 10th Annual Mtg.*, Austin, TX, October 15-19 1995.

[5] R. L. Hollis and A. A. Rizzi. Opportunities for increased intelligence and autonomy in robotic systems for manufacturing. In 8^{th} *International Symposium of Robotics Research*, Hayama, Japan, October 1997.

[6] A. A. Rizzi, J. Gowdy, and R. L. Hollis. Agile assembly architecture: An agent-based approach to modular precision assembly systems. In *IEEE Int'l. Conf. on Robotics and Automation*, pages Vol. 2, p. 1511–1516, Albuquerque, April 1997.

[7] R. R. Burridge, A. A. Rizzi, and D. E. Koditschek. Sequential composition of dynamically dexterous robot behaviors. *International Journal of Robotics Research*, 18(6):534–555, June 1999.

[8] A. A. Rizzi. Hybrid control as a method for robot motion programming. In *IEEE Int'l. Conf. on Robotics and Automation*, pages 832–837, Leuven, Belgium, May 1998.

[9] J. Gowdy, R. L. Hollis, A. A. Rizzi, and M. L. Chen. Architecture for agile assembly: Cooperative precision assembly. In *IEEE International Conference on Robotics and Automation Video Proceedings*, 2000. (submitted).

[10] G. van Rossum. *Python Tutorial*. Corporation for National Research Initiatives, Reston, VA, August 1998.

[11] J. Gowdy and Z. J. Butler. An integrated interface tool for the architecture for agile assembly. In *Proc. IEEE Int'l. Conf. on Robotics and Automation*, pages 3097–3102, Detroit, MI, May 1999.