Programming in the Architecture for Agile Assembly

Jay Gowdy and Alfred A. Rizzi

The Robotics Institute
Carnegie Mellon University

Abstract

The goal of the Architecture for Agile Assembly (AAA) is to enable rapid deployment and reconfiguration of automated assembly systems through the use of cooperating, modular, robust, robotic agents. AAA agent programs must be completely distributed and specify cooperative precision behavior in a structured, well known environment. Thus, the structure of agent programs is carefully designed to allow packaging of all the information necessary for coordinated execution when downloaded to a physical agent. To make the specification and execution of the potentially complex and fragile cooperative behaviors robust, our programs define ordered sets of control strategies and allow a low-level real-time hybrid control system to sequence the strategies rather than burdening the agent program with the management of this critical detail. This novel approach to programming automation systems has been tested both in simulation and on prototype hardware.

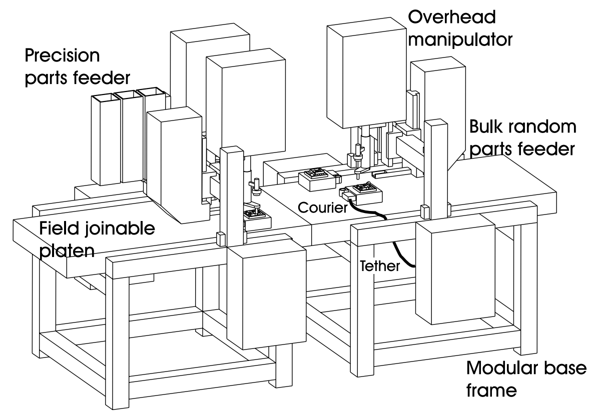

Figure 1: A minifactory segment

1 Introduction

The overall goal of the Architecture for Agile Assembly (AAA) is agility, — to enable both the rapid deployment of factories to deliver a product to market quickly and the rapid reconfiguration of factories to adapt to changing technologies and market needs. As described in [7], AAA achieves such agility by depending on modular robust robotic *agents*. Each agent operates in a deliberately limited domain, but possesses a high degree of capability within that domain. For example, our instantiation of AAA, minifactory, is focused on four degree-of-freedom (DOF) assembly of high-value, high-precision electro-mechanical systems (Fig. 1). In a minifactory there are agents (called couriers) that are "experts" in moving products in the plane of the factory floor, and other agents (called manipulators) that are "experts" in lifting and rotating products. The agents are physically, computationally, and algorithmically modular, and thus only when acting cooperatively in groups can they perform the 4 DOF operations required to produce a product.

In AAA, specifying factory behavior presents some unique challenges, since there is no central factory "brain", and thus there is no single program for an entire AAA factory. Instead each agent has its own program which must reliably execute without access to any central or global database. AAA does provide an integrated interface tool, described in [3], which allows centralized design, simulation, and monitoring of the factory, but this centralized tool need not be present for a factory to operate.

In practice an agent's execution in AAA is divided into two layers. A higher-level discrete layer is responsible for the semantics of factory operation and the associated discrete events. This layer must deal with such issues as resource negotiation, factory scheduling, and product flow decisions. In general these tasks require minimal communications bandwidth between agents and are not concerned with true real-time operation of the agent. A lower-level continuous layer is responsible for sequencing and executing the specific control laws used to effect the physical environment of the agent. This continuous layer not only executes individual parameterized control strategies, but also manages the transitions between a carefully selected set of parameterized controllers. The continuous layer may require high communications bandwidth, since often the states that it must monitor will be on other agents, and true real-time operation is critical. This notion of automatically managing the transition between controllers was introduced in [2] and applied, theoretically, to the domain of minifactory in [6].

The programs written by the user and downloaded to the agent form the upper half of this program hierarchy. The lower half is "hard-coded" in the form of a palette of real-time control strategies and a manager which executes them and administers their sequencing. The discrete layer programs parameterize and deploy the control strategies used by the continuous layer, and then views the continuous state of the agent through the discrete "lens" of monitoring transitions between controllers.

1.1 Programming Model

Most industrial robot programming languages are based on standard computer languages, with the addition of special primitives, constructs, and libraries to support the physical control of a robot[5]. These languages are usually targeted at the control of a single robot, and do not inherently provide support for a program which must be distributed across many different robots. While this model can be effective for "trade-show" or "laboratory" demonstrations of a single robot, it leads to significant complications when a robot must be integrated and coordinate with its neighbors in an actual factory.

More abstract programming models are available[1, 4]; typically these are either "task" or "process" based and are often utilized for programming of work cells — which may contain multiple robots. While such approaches can eliminate some of the problems associated with developing coordination strategies for arbitrary machines, they require a central system "controller," and are thus vulnerable to single point failures and bottlenecks.

Fundamentally, these approaches to robot programming make a distinction between the continuous domain of control theory and the discrete domain of event management. We choose to place this distinction at a slightly higher level and make it a more formal abstraction barrier than most. Traditionally, the continuous, state based view is relegated to running controllers, with all decisions about which controllers to run and when to run them made by systems using a discrete, event based view. Instead we make use of continuous mechanisms to guide the transitions between controllers as well as to run the controllers themselves, freeing the agent programs to deal with the more relevant and abstract problem off deciding what to do and how to do it.

2 Distributed Programming

As there is no central controlling program for a minifactory, the operation of the system results from the cooperation of programs running on each agent. The agents interact with each other and with the factory infrastructure to perform the desired assembly task in an efficient and reliable manner. The distributed, but cooperative nature of the program content has considerable implications upon the program form.

Currently, our agent programs are completely text based and written in Python, an object oriented language which can be interpreted or byte-compiled[8]. An agent program is not simply a script, but rather defines an instance of a class which has a number of specific methods. The program can define a new class to be instantiated, or subclass from a preexisting one, but the class must provide a standard set of methods to be valid. This concept is very similar to the Java applets that are used in world-wide web programming.

A key to writing and distributing an agent program is that even though each agent's program must execute without access to any central database, each individual agent program will necessarily reference other parts of the factory. For example, a courier must be able to know it will be interacting with a particular manipulator much as a manipulator needs to know it will get parts of a specific type from a specific parts feeder.

```
# 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 2: A simple manipulator program

In order to reference these factory components within the text of an agent program, users refer to these factory elements by names. Currently, the names must be unique in the factory, *i.e.* if a manipulator references a parts feeder named *ShaftFeeder* then there must be only one parts feeder with that name in the factory. A running physical agent can not resolve the name *ShaftFeeder* with a central resource, so the agent program is split into two segments, a "bind" step and a "run" step, which means that any valid program instance must have two methods, `bind` and `run`. The `bind` method declares what "global" entities the agent will utilize during its execution. The `run` method is the script which actually runs during execution, implementing the "high-level" discrete logic of the agent which initiates and coordinates the agent behavior. Figure 2 shows the definition of these methods for a

```
# "Applet" 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

sample manipulator program.

When executing a simulated agent program, the bind step is simply a matter of looking up the relevant items in the simulation database and proceeding to execute the `run` method. Before an agent program can be downloaded from the simulation environment to the physical agent it must be first be "bound" with all of the global factory information the agent will need to run the program. To bind a program, the interface tool executes that program's `bind` method, and uses the results to construct a small database of all the information necessary for the agent to locate, both geometrically and logically, all of the factory elements needed to run the program without reference to any global resources. For example, in the sample courier program (Fig 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 the `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 with the program text. After being downloaded with this database the physical agent can execute the `run` method, which in-turn interacts with the lower-level continuous layer to execute the desired physical behavior.

3 High level protocols

Just as there is no central database that agents can rely on during factory operations, there is no central coordinator to organize and direct the agents. Each agent must be programmed to coordinate with its peers to effect the appropriate product flow and assembly operations. In order to achieve this coordination, agents must know and understand common communications protocols. We identify several different types of protocols within our factory, such as built-in protocols that every agent must provide in order to make possible safe factory operations, and extensible protocols that are specific to a particular instantiation of AAA or particular solution approach.

3.1 Built-in protocols

A built-in protocol is one that will be necessary for any agent in any AAA system to produce and understand. For example, there can not be any central arbiter parceling out resources in any AAA system, so every agent has built-in the ability to negotiate with other agents over resource reservation. Agents must have this ability to negotiate for resources in order to ensure safe factory operations.

The primary shared resource that our agents negotiate over currently is space on the platen. We assume that a courier will only go where it says it will go, and that there are no "outside" influences which fail to reserve the resources they consume. These assumptions — which are reasonable in the highly structured, very stable, and well known minifactory environment — allow us to dispense with the inter-agent perception systems that would be necessary to implement completely "reactive" motion, in which agents would be required to observe other agents' positions and intentions (either with sensors or by querying) prior to taking action. The low cost and predictable behavior obtained through the use of a reservation system far outweighs the risk of our assumptions being violated and the minor efficiency losses which will inevitably be incurred. We foresee using a similar distributed reservation system to arbitrate the consumption of more abstract resources such as vibration, noise, thermal, or optical emissions.

Another example of a built-in information protocol is seen during a rendezvous, *i.e.* when a courier and a manipulator cooperate to perform some process on the products: When agents cooperate to perform the manufacturing process, they also must exchange information about that process. In AAA, there is no central database of product information, so product information must flow with the products themselves. Our products have two levels of information, prototype information — information that is true about all products of a certain type, such as nominal geometry, and instance information — information that applies only to a particular instance of a product, such as serial numbers or dimension variations. AAA provides built in protocols for passing product instance information between agents and for acquiring product prototype information either from peers at run-time or from a database at program binding time.

3.2 Extensible protocols

One more protocol that all agents share is the protocol for defining and extending semantic protocols. A particular semantic protocol may not be in use by all agents, but agents can negotiate to confirm whether they share the same semantic protocols before proceeding with operations.

For example, in our current approach to programming agents in minifactory, we view the agent programs as having two types of interactions, the rendezvous between a courier and a manipulator in which the manufacturing process is performed and information about the process is exchanged, and the gross courier motion, in which couriers move from rendezvous to rendezvous without colliding. Keeping the couriers from colliding into each other results from using the built-in geometry resource negotiating protocols, but deciding in what order couriers may rendezvous with manipulators, *i.e.* distributed factory scheduling, is the domain of an extensible protocol. An example of this protocol can be seen in the sample courier program (Fig. 3), which initiates a rendezvous to be accepted as specified in the sample manipulator program (Fig. 2).

This protocol is particular to minifactory, and particular to our current approaches to programming minifactory agents. It may not be useful in other AAA instantiations, and almost certainly will be significantly changed or augmented over time in our minifactory instantiation. Thus, a courier and a manipulator need to negotiate to ensure that they share a common rendezvous protocol before they can work together.

4 Low Level Programming

As outlined in Section 1 an agent program in a minifactory has two distinct but related run-time responsibilities: *i)* it must carry out semantic negotiations with its peers to perform the goal of the factory; *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 attempts to simplify the relationship between these two responsibilities and minimize their impact upon one another. The basics of how to accomplish high-level programming tasks was the topic of Section 3. Here we turn our attention to the low-level tasks and the programming of physical motion.

Specifically, in an effort to reduce the complexity associated with writing programs for agents we have adopted the notion of allowing the low level control strategies to become responsible for their own switching and sequencing. Thus the problem of deciding exactly when and how to switch between low level control strategies is removed from the agent program and is thus isolated from the high-level semantic negotiations that are the primary domain of the agent program.

4.1 Underlying Model

The fundamental model for the execution of control strategies was presented in [6]. Briefly, rather than ask the program to generate trajectories through the free configuration space of the agent, the program will be responsible for decomposing the free configuration space into overlapping regions and parameterizing controllers associated with each region. 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.

This scheme describes the behavior of any one agent in terms of a collection of feedback strategies based on the state of both the individual agent and its immediate peers. The result is a hybrid on-line control policy (one that switches between various continuous policies) which makes use of the entire collection of available policies to systematically make progress toward a goal based on an agent's estimate of both its own and its peers' state. To provide the desired level of system flexibility the selection of goals and the associated prioritized decomposition of the free space is left to the agent program.

More formally, given a set of controllers, $U = \{\Phi_1, ..., \Phi_N\}$, each with an associated goal, $G(\Phi_i)$, and domain,$D(\Phi_i)$ — where it is presumed that under the action of Φ_i any state that starts in $D(\Phi_i)$ will be taken to $G(\Phi_i)$ without leaving the set $D(\Phi_i)$. We then say that controller Φ_1 *prepares* controller Φ_2, denoted $\Phi_1 \succeq \Phi_2$, if its goal lies within the domain of the second $G(\Phi_1) \subset D(\Phi_2)$ [2, 6]. For an appropriately parameterized set of controllers, U, this relation induces a generally cyclic directed graph. Assuming that the overall goal, G, coincides with the goal of at least one controller, $G(\Phi_i) = G$, then by starting with Φ_i and recursively tracing the relation backwards through the corresponding graph, one arrives at $U_G \subset U$ — the set of all controllers from whose domains the overall goal might be eventually reached by switching between control policies. The domain of a properly conceived composite controller, should then be $\bigcup_{\Phi \in U_G} D(\Phi)$, and thus we have an "automatic" method by which to guide the system from any state in this union of domains to the goal.

Consider, for example, the trivial planar configuration space depicted in Fig. 4. Note that the free space has been decomposed into four separate regions with the overall goal located in the upper right corner of the configuration space (G_4). Here, Φ_1 is responsible for for taking all states in the lower convex region to

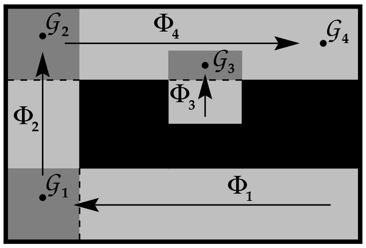

Figure 4: Example decomposition of a trivial planar configuration space.

G_1, and thus prepares Φ_2. Similarly the placement of G_2 and G_3 allow both Φ_2 and Φ_3 to prepare Φ_4 which regulates the state to G_4, the overall goal. It is the responsibility of the underlying hybrid control system, discussed in detail in [2], to switch between the individual Φ_i to achieve this overall goal. While this trivial example is illustrative it is important to note that we have only considered the configuration of the system in this example, while in general the actual domains, $D(\Phi_i)$, for the constituent controllers are defined over the state space of the system — the positions and velocities of the agent as well as those of its peers with which it is closely coordinating.

4.2 Programmatic Interface

Given the underlying model for executing physical action described in Section 4.1, it becomes 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, U, along with the associated sets of goals, $G(\Phi_i)$, and domains, $D(\Phi_i)$. 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 error prone problem of making real-time changes to the underlying control system is left to the hybrid control structure outlined above.

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. Having constructed a controller the script can then insert into an ordered list of active controllers from which the controller manager will select the appropriate instance 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, and utilizes a dedicated local network. 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 parameterizing (setting the goal, defining the domain of applicability, specifying gains, etc.) the specific controllers and ordering of 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 at the same time minimizing the risks associated with writing those programs.

```
# 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 5: Code fragment for `moveTo`.

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 Fig. 3, would actually expand to the code fragment shown in Fig. 5. It is here that the specific resource reservation protocol mentioned in Section 3 is implemented and where a "standard" set of controllers are parameterized and placed on the list of active controllers. Note the registration of a call-back method to indicate exit from the initial area and to free the reservation held on it.

5 Conclusion and Future Work

The requirements of AAA have led us to a new model for programming assembly systems. AAA agent programs must be completely distributed and specify cooperative precise behavior in a structured, well known environment. Thus, the structure of agent programs is carefully designed to allow packaging of all the information needed to execute when downloaded to a physical agent. The programs must use standard high-level protocols to initiate the required cooperative behavior. To make the specification and execution of the potentially complex and fragile cooperative behaviors robust, our programs define ordered sets of control strategies and allow a low-level real-time hybrid control system to sequence the strategies rather than burdening the agent program with the management of this critical detail.

We have tested this approach in simulation by constructing virtual factories with several couriers and manipulators cooperating to perform part of the assembly of small (2 millimeter) transducers. In addition, we have written agent programs which, both in simulation and hardware, exercise our prototype courier. We are currently integrating our prototype manipulator in order to implement multi-agent pick-and-place tasks. We will continue to validate the programming approach with real tasks as we develop additional hardware that supports those tasks.

There is much yet to do to address some of the practical implications of our programming model. For example, in order to produce a working factory, users must generate many correct cooperating agent programs. Fortunately, an individual agent's scope is fairly limited, and it has powerful tools for working within its scope so our hope is that each agent program will be relatively simple and short. Unfortunately, no matter how short or simple the programs, the fact remains that some factory programmer has to generate an individual program for each agent in the system. In addition to the potential tedium of generating dozens of programs, the user is essentially writing large, very distributed programs, with all of the known pitfalls of that domain, such as deadlocks or livelocks.

We could address this problem through the use of graphical programming techniques to ease the production of the individual agent programs, but we feel that any advantage gained would be purely cosmetic. Fundamentally, what is required is a method of presenting the factory programmer with a different way of looking at the programming problem. For example, users may want a factory-centric view of the problem, in which they can specify the factory behavior as a whole by inputting a work-flow model, *i.e.* what processes have to occur and in what order. Ultimately, users may want to take a product-centric view, in which they enter product models annotated with some process information. The AAA programming environment would have to provide semi-automatic, user-guided methods of transforming such centralized user views 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 an actual AAA factory must execute that user program as a set of completely distributed programs on a set of agents interacting with each other and with the product components to perform the assembly task. This paper has documented the programming model and protocols we have designed as a basic building block for future systems which can bring the vision of rapid deployment, reconfiguring, and reprogramming of automated assembly systems closer to reality.

Acknowledgements

This work was supported in part by NSF grant DMI-9523156. The authors would like to thank Ralph Hollis, Arthur Quaid, Zack Butler, and Patrick Muir for their invaluable work on the project and support for this paper.

References

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

[2] R. R. Burridge, A. A. Rizzi, and D. E. Koditschek. Sequential composition of dynamically dexterous robot behaviors. *International Journal of Robotics Research*, 1998. (to appear).

[3] J. Gowdy and Z. J. Butler. An integrated interface tool for the architecture for agile assembly. In *IEEE Int'l. Conf. on Robotics and Automation*, 1999.

[4] 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.

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

[6] 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.

[7] 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 20–25, Albuquerque, NM, April 1997.

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