An Integrated Interface Tool for the Architecture for Agile Assembly

Jay Gowdy and Zack J. Butler

The Robotics Institute
Carnegie Mellon University

Abstract

Developing automated assembly systems normally happens in two distinct stages: first an "off-line" stage in which the system is designed and programmed in simulated and then an "on-line" stage in which the simulation results are used to minimize the deployment and integration time of the physical machines. The distinction is so great that usually completely different software environments are used in the design phase than are used in the deployment and operation phase. We are developing The Architecture for Agile Assembly (AAA): a comprehensive, integrated framework that is designed to blur these stages together and ease the transitions between them. We have used the protocols of AAA to create an integrated interface tool which can be used throughout the life-cycle of a developing AAA factory, from its design to its operation. We have tested the integrated interface tool both in simulation and with our prototype hardware, which is designed for high precision four-degree-of-freedom assembly.

1 Introduction

Off-line robot programming promises to reduce the time necessary to produce automated assembly systems, since much of the design and programming work can be done in simulation, where experimentation is usually cheap and mistakes are generally not catastrophic[5, 9, 13]. Unfortunately, a problem inherent in off-line programming is evident in its name: for an "off-line" program to be truly useful, it must become an "on-line" program running on a physical assembly system. If the physical factory does not match the simulated factory to a sufficient degree in either geometric or functional characteristics, then the transition from the world of pixels and bits to the world of actuators and metal can be arduous.

This difficult transition has long been recognized as a problem with off-line programming.[3] Approaches to easing the transition include

- Calibration: sensing the factory configuration and adapting the programs to match[8].
- High fidelity modeling: using standardized models of components and component assembly constraints combined with physical modeling to reduce the number of surprises that can occur in the transition[4].
- Sensor feedback: using sensors such as machine vision or force feedback to overcome discrepancies between simulation and reality[1].

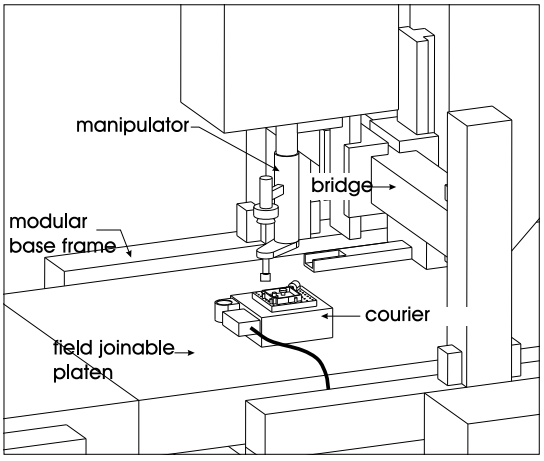

Figure 1: A minifactory segment

These techniques can ease the transition between simulation and reality, but they need to be embedded in a system which can move back from reality to simulation. If they are used in a "one-pass" approach, in which the transition from simulation to reality is done just once, then the full potential of the simulation will be lost. For example, if contact with the physical world necessitates large changes in programming and factory configuration, all of those changes must be made in the relatively unforgiving environment of the physical machines. Similarly, if the product or processes change, since the original simulated factory is not the same as the implemented physical factory it is difficult to meaningfully address those changes in simulation. To truly achieve *agility*—the ability to rapidly deploy factories to deliver a product to market quickly and to rapidly reconfigure factories to adapt to changing technologies and market needs—simulation and physical implementation need to be more closely coupled and done iteratively.

The overall goal of the Architecture for Agile Assembly (AAA)[12] is such agility: AAA is designed to allow real factories to be built incrementally and reconfigured often as manufacturing processes are perfected and needs change. Rather than having a strict separation in methods and mechanisms between design and execution, we have developed a single *interface tool* which bridges both. This integrated interface tool provides a 3D graphical environment that guides an evolving AAA factory through its various stages, allowing the user to view and interact with the factory as it is assembled, simulated, programmed, implemented, and operated.

AAA systems are composed of a set of 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* (Fig. 1) is focused on four-degree-of-freedom (DOF) assembly of high-value, high-precision electro-mechanical systems. In a minifactory there are courier agents that are "experts" in moving products in the plane of the factory floor (the platen), and manipulator agents that are "experts" in lifting and rotating products. The agents are physically, computationally, and algorithmically modular, and only when acting cooperatively can they perform the 4-DOF operations required to produce a product. In order to perform this kind of cooperation, each agent knows its own characteristics, abilities, and state, and can use the protocols of AAA to advertise that information to its partners.

It is this self-representation that is the key to the integrated interface tool. Not only do agents share their self-knowledge with each other, they also share it with external entities such as the graphical user interface. This paper describes how the AAA approach and protocols enable such a unified interface tool to shepherd a minifactory throughout its life cycle.

2 Loading Factory Components

A minifactory is made up of *components*, which include active agents such as manipulators and couriers, and passive support elements such as bridges, modular base frames, and field joinable platens (Fig. 1). Every component in a minifactory has a *description*, which is a database containing all of the pertinent information about that component, including how to view it, how to model its geometry and behavior, and how to constrain its assembly to other components.

In order for the simulated minifactory to quickly become a working minifactory it is vital that the component descriptions loaded into the interface tool match the real attributes of the physical components. An agent which cannot fit where its description says it can fit or cannot do what its description says it can do will drastically increase the time spent implementing and integrating the minifactory.

Fortunately, a fundamental aspect of AAA agents

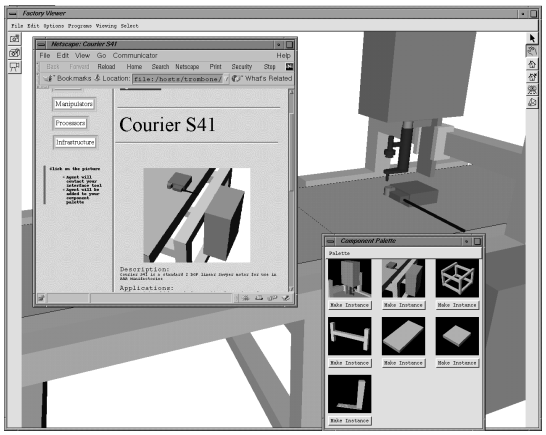

Figure 2: Interface tool interacting with a web browser

is that they all maintain and publish descriptions of themselves. If a factory designer wants to minimize the likelihood of inaccurate information, that designer should directly access the agent itself rather than any catalog or other mediator. There is no absolute guarantee that the agent's self-description is accurate, but it is more likely to contain accurate, up-to-date information specific to that agent than is a generic description of an agent of the appropriate class, or even a passive catalog of agent descriptions.

To demonstrate the direct-access approach, we have implemented a simple example of loading an agent's self-description into the interface tool (Fig. 2). Factory designers can use their World Wide Web (WWW) browser to find a catalog of our prototype agents. Each catalog entry is associated with one particular agent in our laboratory. Clicking on a button next to the catalog entry starts up a helper application which sends an agent "URL" to the interface tool. The interface uses the agent URL to contact the actual agent, and if it is active puts a reference to that agent in a *component palette*. The component palette is a list of iconified representations of the components along with buttons which allow the user to reserve a component and insert a representation of that component into the design environment. Designers can also use the component palette to insert "clones" of components into the design environment, *i.e.* the remote components will not be reserved for use, but simulated representations of them will be available for design experiments.

We have also implemented simple component servers which distribute and reserve descriptions of passive components, such as our prototype base frames and platens. These components do not have any active computational element as the agents do, so there is nothing uniquely associated with a passive component which the interface tool can contact. Thus, the component servers act as active catalogs which the user interface can access. This approach should suffice, since passive components should not change often, and thus

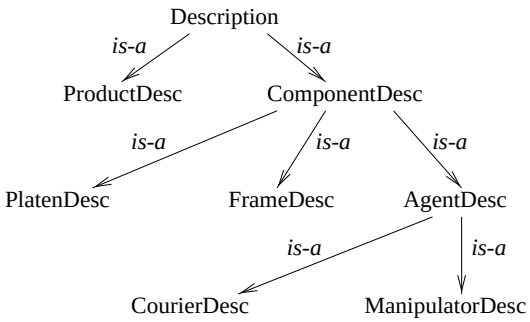

Figure 3: Class diagram for factory descriptions

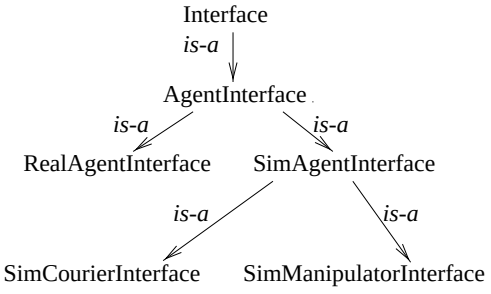

Figure 4: Class diagram for agent interfaces

can have stable, well cataloged geometric and functional attributes.

3 Constructing Factories

The interface tool enables factory designers to quickly take the component descriptions they have found and assemble them together into a variety of different factory configurations, snapping them together and pulling them apart as they design the final factory.

Each component description contains a specification of how it snaps together with other components. Currently we use a short cut which takes advantage of the object oriented capabilities of our implementation language, C++, to "hard-code" this specification. As the class diagram in Fig. 3 shows, all factory infrastructure component descriptions are descendents of the *ComponentDesc* class. All such subclasses must implement an `assembleTo` method, which takes another *ComponentDesc* as an argument and determines first if the target component is a valid candidate for component assembly, and if it is, performs the appropriate manipulations on the description to attach the original component to the target.

For example, if the user desires to assemble a given platen to a given base frame, then the user simply selects them both in the interface tool's 3D rendering and directs that they be assembled together. The `assembleTo` method of the platen description will be invoked with the base frame description as an argument. This `assembleTo` method will verify that the base frame is a valid type for assembling itself to, and will extract from the base frame's description where platens should be mounted. Finally, the platen description's `assembleTo` method will move the rendered platen representation so that it is mounted appropriately on the chosen simulated base frame.

4 Simulation and Programming

The interface tool lets a factory designer write and debug simulated agent programs while the factory is being constructed. The factory designer views a 3D rendering of the running simulated factory as a whole, and interacts with individual agents through virtual control panels, using them to observe their state variables and debug the individual agent programs.

Every agent description must be a subclass of *AgentDesc* which contains a field named `interface`. The `interface` field is itself a database which encapsulates the actual implementation of the agent—whether it is a simulated agent running in the interface tool or it is a physically instantiated agent which the interface tool is interacting with remotely. The entries in the `interface` database will be items such as state variables which can be monitored or parameters which can be changed to affect the simulated or physical agent operation.

The value of the `interface` field must be a subclass of *Interface* (Fig. 4). Any subclass of *Interface* must implement an `update` method. The interface tool maintains a list of agent descriptions that it is monitoring or simulating, and calls the `update` method of each agent description's interface as often as possible. The particular implementation of the `update` method appropriately moves the rendered parts of the agent description, such as the graphical representation of a manipulator's end effector.

Currently, the user specifies an agent's run-time behavior through the use of a program written in Python (a byte-coded, object-oriented programming language). These programs are similar to the applets used in WWW programming in that they instantiate objects with required methods rather than simply being a sequence of commands. The program objects specifically have a *bind* method which is used to indicate all of the global factory components the agent will use, and a *run* method which is the actual script that determines the agent run-time behavior[6].

The architecture for simulating agent behavior is shown in Fig. 5. When the user runs a script, the interface tool creates or contacts a scripting manager process which launches the thread that will actually execute the script. This script performs the necessary inter-agent cooperation and sets up and monitors *controllers*—the specification of the time-varying behavior of the agent. The controllers are actually simulated in the interface tool within the `update` method of the simulated agent interfaces, even though which

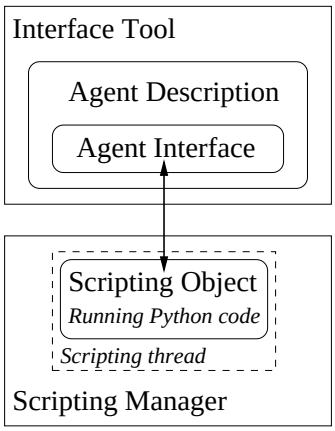

Figure 5: Architecture for simulated agents

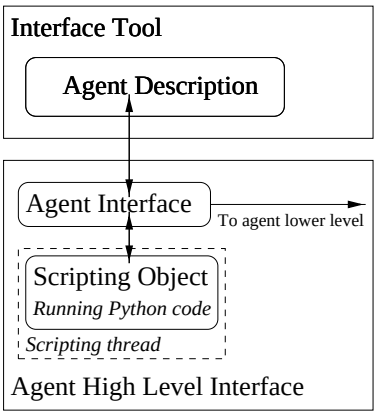

Figure 6: Architecture for physically instantiated agents

controllers are being run is specified by the external scripting thread. All such simulated agent interfaces are subclasses of *SimAgentInterface*, and have similar implementations of their `update` method: first the rendered position and orientation of the end effectors are used to generate current state variable values, then the simulated controller specified and parameterized by the scripting thread is run, and finally, the new set of estimated state variables is transformed into modified 3D renderings of the end effectors.

The simulations currently only model the motions of the agent end-effectors. These simulations are fundamentally kinematic, using a motion model with bounded velocity and acceleration ($\ddot{x} = u$, $\|\dot{x}\|_\infty \leq V_{max}$, $\|u\|_\infty \leq U_{max}$). The different controllers are sequenced using the same efficient hybrid control strategies for robust motion execution that are used in the physical agents[11].

5 Physical Instantiation

Simulation never matches reality, a fact which often limits the utility of off-line programming, since much of the effort is expended in the details of making a system work on physical machines well after the simulations are finished. AAA agents first address this gap by being able to calibrate themselves and explore their physical environments. As long as the simulated world and the real world are topologically compatible and the real world allows the same physical motions as the simulated agents, programs should be able to run in the newly calibrated environment unchanged.

The same interface tool that was used to program a simulated minifactory initiates the transformation into a real factory and monitors this calibration and exploration process. After exploration, the virtual factory rendered by the interface tool is in essentially perfect correspondence with the real factory. We expect users to then switch back to simulation to test the programs using the registered virtual environment before actually trying them out on the real equipment. Any fatal discrepancies, such as swapping manipulator positions or moving components too close together to allow safe passages for couriers, should be detectable and corrected by the user at this point.

We do not expect that this geometric registration will make the transition between simulation and reality seamless: simulations will always miss some aspect of reality that may necessitate reprogramming and possibly reconfiguration. We believe that lowering the cost of moving back and forth between simulation and reality and using the same development environment for both will naturally encourage an iterative approach to developing an automation system, in which major changes imposed by any unmodeled surprises or changes in requirements and capabilities can be addressed first in simulation, and then transferred to reality.

Switching from a simulation of an agent to using a real agent is a simple matter of changing the agent description's interface to one on a physical agent and transferring the program to that new interface. In simulation, the interface tool "owns" the agent description's interface, *i.e.* it completely resides within the interface tool process. The interface to a physical agent will be owned by the high level agent process running on the physical agent's computer, as shown in Fig. 6. Thus, for a physically instantiated agent, the agent description's `interface` field will contain only a reference to a remote database existing on the agent hardware itself. The same script will run on the physical agent as on the simulated agent, except that the scripting thread will be run by the agent rather than by any independent scripting manager and the controllers will be run using the actual control algorithms of the agent rather than being simulated by the interface tool.

Before programs run on the agents, the agents attempt to calibrate themselves and their environment. Part of every agent's calibration process is introspective, *i.e.* establishing and confirming their own phys-

ical and behavioral characteristics such as joint-limits or acceleration abilities. Additionally couriers, being the mobile factory agents, explore the exact geometric relationships between themselves and the factory components around them. Each courier is equipped with an optical sensor that can locate precisely placed LEDs to sub-micron resolution[7]. Multiple LEDs mounted on relevant components such as manipulator end-effectors or bridges are used to precisely determine their actual positions and orientations relative to the exploring courier's frame of reference.

We are currently experimenting with two scenarios to accomplish this exploration. In the first scenario, each courier is given the estimated locations of the relevant landmarks (*i.e.* components with locator LEDs mounted on them) by the interface tool. It then negotiates with other couriers for the right to confirm the positions of these landmarks, and builds up an accurate local map from the inaccurate one. The second scenario assumes no initial local map, but rather has each courier run a geometrically complete algorithm that covers its workspace. It will then have discovered the exact locations of all of the relevant components around it[2].

In either case, the interface tool audits the results of each courier's exploration and uses those local maps to generate a new accurate global map. In order to maintain consistency within the global map, the interface tool simply creates a tree of the geometric relationships between the factory components. When a courier precisely identifies the geometric transform between two components, the interface tool immediately adds this information to the tree if possible. If the new information describes a significant inconsistency with previous information, the user is notified. Whereas this solution is not capable of resolving redundant calibration information in an intelligent way, it is sufficient for the current implementation. More capable reconciliation algorithms based on geometric graph matching are under investigation.

6 Monitoring

Once a factory is instantiated, the interface tool should no longer be strictly necessary, since the programs run on each agent without need for any central control or resources. However, the interface tool can still provide a central place for monitoring factory operations in order to keep records, detect problems, and discover means of optimizing the factory.

The primary mechanism for monitoring the factory operations is watching the agents' state and monitoring the products that the agents are manipulating. The interface tool provides a 3D rendering of the whole minifactory as it performs assembly tasks, as well as providing virtual control panels to monitor less visible elements of an individual agent's state.

The 3D rendering of the physical agents' states is achieved through their interface's `update` method,

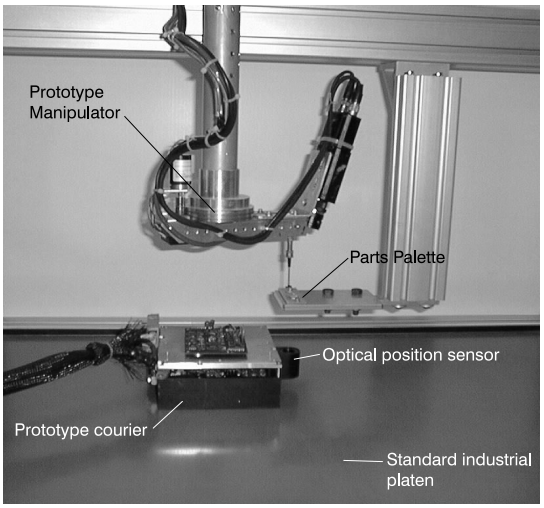

Figure 7: Prototype minifactory hardware

just as the 3D rendering of the virtual agents' states was. All physical agent interface field values are subclasses of the *RealAgentInterface* class, and their `update` method simply examines the current state values and converts those values into updates of the position and orientation of the rendered agent's end effector.

The interface tool monitors products flowing in the system by registering with each of its agents as a "product auditor," *i.e.* the interface tool will be notified when the agents pick up products, when they transfer products, and when they drop products. Thus the interface tool can track the progress of products through the factory, both visually, by presenting a 3D representation of the products as they flow through the factory, and statistically, by maintaining records on the products as they flow through the factory.

The interface tool may prove to be a communications bottleneck for monitoring factory operations, *i.e.* as a factory gets large, sending product information and state values to the interface tool may prove overwhelming. We have attempted to be efficient about the amount of information that is transmitted from the agents to the interface tool, but no matter what is implemented, it is inevitable that a single, central monitoring point will become overwhelmed for some large factory size. Fortunately, since the interface tool is not the central brain of the factory, overwhelming it does not cripple the factory. Furthermore, since the interface tool does not need to serve as the central coordinating resource, there is no need to have only one. For a very large factory, it may in fact make sense to have two or more interface tools monitoring sections of the factory to eliminate any potential bottlenecking problems.

7 Conclusions

We have tested the integrated interface tool with our prototype minifactory test-bed, which uses a fully instrumented courier and an overhead manipulator running on a standard platen and frame (Fig. 7). The integrated interface tool can contact these prototype agents and load their descriptions into its environment. The courier can explore its environment, locate landmarks which contain LEDs with a repeatability of ± 2 μm (which is comparable to the limit of repeatability of our current open-loop courier motion control) and transmit the updated positions of those landmarks to the interface tool, which adjusts the 3D rendering of the minifactory. Once closed-loop control algorithms[10] have been integrated onto the AAA couriers, the repeatability of the localization is expected to move closer to the sub-micron resolution of our optical and magnetic sensors. We have successfully developed programs in simulation within the interface tool and transferred them to the physical agents and run them, after exploration, with no change.

The most pressing deficiency in our current interface tool is the hard-coding of the assembly constraints and behavioral specifications as C++ methods, which means that no fundamentally new agents can be introduced to the system at run time. Another major problem with our current interface tool is that specifications of components and products must go through a tedious and somewhat error prone conversion process from their original CAD model to our current custom factory description format. We are investigating addressing both of these problems through use of the STEP standard (Standard Exchange of Product Model Data)[14]. Adopting STEP will give a means of importing models using an industry standard mechanism while also providing a means of adding the run-time flexibility we need, as extensible elements of STEP can be used to specify behaviors and assembly constraints of novel agents at run-time.

There are many other areas of research and development involving the interface tool, such as using geometric and physical modeling in the simulations or implementing better programming interfaces. We are attempting to build the foundation from which we can experiment in these various areas.

AAA is not only a comprehensive vision, it is an integrated vision which is involved in the development of an automation line throughout its life cycle, from conception to implementation. In fact, the goal of AAA is to not only be involved at the various stages of development, but to blur the distinction between the stages to allow incremental design and modification of automated factories. Such an integrated vision demands an integrated interface tool such as the one we have implemented. Fortunately, at the same time as AAA demands an integrated interface tool, it provides the mechanisms that make such a tool if not trivial, at least straight-forward to implement.

Acknowledgements

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

References

[1] B. Brunner, K. Arbter, and G. Hirzinger. Task directed programming of sensor based robots. In *Proceedings of IEEE Int'l. Conf. on Intelligent Robots and Systems*, pages 1080–1087, 1994.

[2] Z. J. Butler. CC_R: A complete algorithm for contact-sensor based coverage of rectilinear environments. Technical Report CMU-RI-TR-98-27, Robotics Institute, Carnegie Mellon Univ., 1998.

[3] S. F. Chan, R. H. Weston, and K. Case. Robot simulation and off-line programming. *Computer-Aided Engineering Journal*, pages 157–162, August 1988.

[4] J. J. Craig. Simulation-based robot cell design in AdeptRapid. In *IEEE Int'l. Conf. on Robotics and Automation*, pages 3214–3219, Albuquerque, NM, April 1997.

[5] S. Derby. GRASP from computer aided robot design to off-line programming. *Robotics Age*, 5(2):11–13, 1984.

[6] J. Gowdy and A. A. Rizzi. Programming in the architecture for agile assembly. In *IEEE Int'l. Conf. on Robotics and Automation*, 1999.

[7] J. W. Ma. Precision optical coordination sensor for cooperative 2-DOF robots. Master's thesis, Carnegie Mellon, 1998.

[8] R. S. McMaster and F. M. Ribeiro. Cell calibration and robot tracking. In *IEE Colloquium on Next Steps for Industrial Robots*, 1994.

[9] J. S. Mogal. IGRIP - a graphics simulation program for workcell layout and off-line programming. In *Robots 10 Conf. Proc.*, pages 65–77, 1988.

[10] A. E. Quaid and R. L. Hollis. 3-DOF closed-loop control for planar linear motors. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 2488–2493, Leuven, Belgium, May 1998.

[11] 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.

[12] 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.

[13] SILMA Division, Adept Technology Inc., San Jose, CA. *The Adept-Rapid Users Manual*, 1996.

[14] STEP: Standard for the exchange of product model data. ISO 10303, 1994.