NORTHWESTERN
UNIVERSITY

Computer Science Department

Technical Report
NWU-CS-02-10
August 2002

RoboTA: An Agent Colony Architecture for Supporting Education

Sven E. Kuehne
Kenneth D. Forbus

Abstract

Resources in education and training are always limited, and instructors almost never have enough time to spend on the problems of their students. Using computers to complement human instructors has been a long-standing motivation for research on AI in education. Although software coaches typically are not as good as the best human instructors, the ability to provide feedback on the student's schedule rather than the instructor's can make them a useful component in an educational system. We believe that using an agent architecture to provide coaches that operate outside software installed on the student's machine can provide valuable advantages in education and training. This paper describes RoboTA, an architecture for a colony of agents aimed at supporting instructional tasks. RoboTA's agent colony architecture provides the sharing of communication though a central Post Office, an Agent Toolkit for the easy construction of new agents, and the ability to host agents on remote machines on a network

This research was supported by the National Science Foundation (NSF) and the Office of Naval Research (ONR).

Keywords: distributed coaching, agent architectures

A shorter version of this paper originally appeared as:

Forbus, Kenneth D. & Kuehne, Sven E. (1998). RoboTA: An Agent Colony Architecture for Supporting Education. *In: Sycara, Katia P. & Wooldridge, Michael (eds.).* Proceedings of the Second International Conference on Autonomous Agents (Agents 98*). May 9-13, Minneapolis/St. Paul, MN, pp. 455-456.*

RoboTA: An Agent Colony Architecture for Supporting Education

Sven E. Kuehne
Kenneth D. Forbus
{skuehne,forbus}@northwestern.edu

Abstract

Resources in education and training are always limited, and instructors almost never have enough time to spend on the problems of their students. Using computers to complement human instructors has been a long-standing motivation for research on AI in education. Although software coaches typically are not as good as the best human instructors, the ability to provide feedback on the student's schedule rather than the instructor's can make them a useful component in an educational system. We believe that using an agent architecture to provide coaches that operate outside software installed on the student's machine can provide valuable advantages in education and training. This paper describes RoboTA, an architecture for a colony of agents aimed at supporting instructional tasks. RoboTA's agent colony architecture provides the sharing of communication though a central Post Office, an Agent Toolkit for the easy construction of new agents, and the ability to host agents on remote machines on a network

Introduction

In education and training there are never enough instructors to go around. Resources are always limited, so using computers to complement human instructors has been a long-standing motivation for research on AI in education. Although software coaches typically are not as good as the best human instructors, the ability to provide feedback on the student's schedule rather than the instructor's can make them a useful component in an educational system. Such coaches have typically been hard-wired into specific software tutors and learning environments. We believe that using an agent architecture to provide coaches that operate outside software installed on the student's machine can provide valuable advantages in education and training. RoboTA is a colony architecture for software coaches in educational environments. A RoboTA agent colony has some specialized agents, plus agents written for specific courses. Students will interact with RoboTA via email. Instructors will interact with RoboTA via email and private web sites. We see the key advantages of the RoboTA architecture as

- *Exploiting communication media that students are already comfortable with*. In modern university environments students are already familiar with email, and this familiarity is now spreading to high schools in some areas. Responding to student questions by a combination of personalized email plus pointers to relevant web-based resources is easy for software coaches to do and familiar to students.
- *Simplified creation of new software coaches*. By providing a common communications infrastructure, it should be easier to add a new coaching agent to a RoboTA colony than it would be to create a standalone agent from scratch.
- *Simplified evolution of coaching services*. Updating a coach to handle new cases or fix bugs only requires fixing the software on the computer hosting that agent, rather than having the entire student population download and install an update.
- *Data recording*. Research in education requires gathering data about how students actually use experimental software. While the widespread use of microcomputers provides revolutionary opportunities for creating new educational systems, it has also made gathering data about how well they work far more difficult. Gathering data as a side-effect of providing services that students and instructors find useful should facilitate formative evaluation processes.
- *Grading services*. By operating coaches on trusted machines, software coaches can reduce the burden of the clerical aspects of grading. Students can turn in assignments via email, which is then checked by the coach, which produces on-line reports for the instructors. Doing the mechanical aspects of

grading via software (e.g., seeing whether a student's power plant is correctly analyzed and meets the quantitative specifications of the assignment, or testing the student's program on important cases) gives the instructors more time to focus on providing higher-level feedback (e.g., how elegant and creative is the student's design?).

Development of the RoboTA colony architecture has been driven by need. One of our research projects involves creating *articulate virtual laboratories* (Forbus & Whalley, 1994; Forbus, 1997). One of our prototypes, CyclePad, helps students learn engineering thermodynamics. The AVL approach is design-oriented, on the grounds that students find design tasks highly motivating and such tasks require them to learn the fundamentals more deeply. CyclePad is now used by a number of universities on an experimental basis, some of them with large populations of student users (e.g., around 60 students per year at the US Naval Academy) and some of them quite distant (e.g., University of Queensland in Brisbane, Australia). Providing support and gathering data from a diverse set of remote sites has proven to be a challenging problem. Also, our desire to provide improved coaching must be balanced with our user community's need for stability and keeping runtime systems small enough to work well on the PC's our student populations have. The ability to evolve our coaching software while giving our users the stability they desire, along data gathering, were the initial driving considerations for RoboTA. We soon realized that moving from a single-agent model to a colony model, where some specialized agents provided infrastructure services for the rest, should enable the system to support multiple projects with only a small amount of additional complexity.

The constraints in the AVL project are not unique: Other educational research software often has the same problems. Moreover, there are many courses that could benefit from software coaches. Some programming courses use server-based systems to provide student feedback (Stern, 1997; Brusilovsky, Schwarz & Weber, 1996). A proof-checker for logic courses is now available on the web (Allen & Hand, 1992; Allen, 1997). Such coaches would be easier to build if the communication infrastructure were simplified and standardized. The goal of the RoboTA architecture is to provide such services, so that educational institutions can "grow" their own colony of support agents.

The next section describes the RoboTA architecture and its implementation. Then we outline two coaching agents being constructed for deployment within the first RoboTA colony, at Northwestern. We close with some observations and our schedule for deployment.

Concept and Structure

When we designed RoboTA one of the most important aspects was to create a system that can easily be extended. To make RoboTA useful (and useable) for different kinds of applications, we split the system into a central server process and an application-specific agent processes.

The server process is called the PostOffice because its major function is the communication with the users of the system. Users send their requests by email to the PostOffice and will receive responses from the PostOffice as well. The fact that the real work is done by an agent working on behalf of the PostOffice is hidden from the user. This has the advantage that there is only one email address the user has to remember and she also doesn't have to care about which agent might be responsible for which kind of task: this is the job of the PostOffice. Once it retrieves messages from the users it runs a set of rules on the content of the individual messages. The result of the rule check should be the name of an agent that will handle the message. The PostOffice then forwards the message to the specific agent – or returns it to the user if no appropriate agent was found. It is possible that more than one rule fires and the rule check results in a number of agents who can handle the message. In these cases the PostOffice will forward the message to all of these agents. This allows separate agents to process the same input message for different purposes. A message from a student to sign up for a course might be processed by separate agents who maintain class lists, registration information and send out the necessary reading material or software for the course.

The agent receives the message from the PostOffice over a TCP/IP socket connection. This enables us to run agents on other machines on our network. In many cases it is advantageous to have the agent running

on a machine that is directly administered and controlled by the person who wrote the agent. If a TA wrote an agent to maintain his course lists, RoboTA allows the TA to run his agent on his own machine. This makes changing and updating the agent code much easier, because it does not require any action on behalf of the administrator of the PostOffice.

After the agent has processed the message it might generate a reply to the user. This reply message is send to the PostOffice over a socket connection. The PostOffice then takes care of the message and forwards it by email to the user.

Figure 1 shows an overview of the architecture of RoboTA. The following sections will provide a more detailed look at the internals of RoboTA, the design decisions, and the tools we provide for writing your own agent.

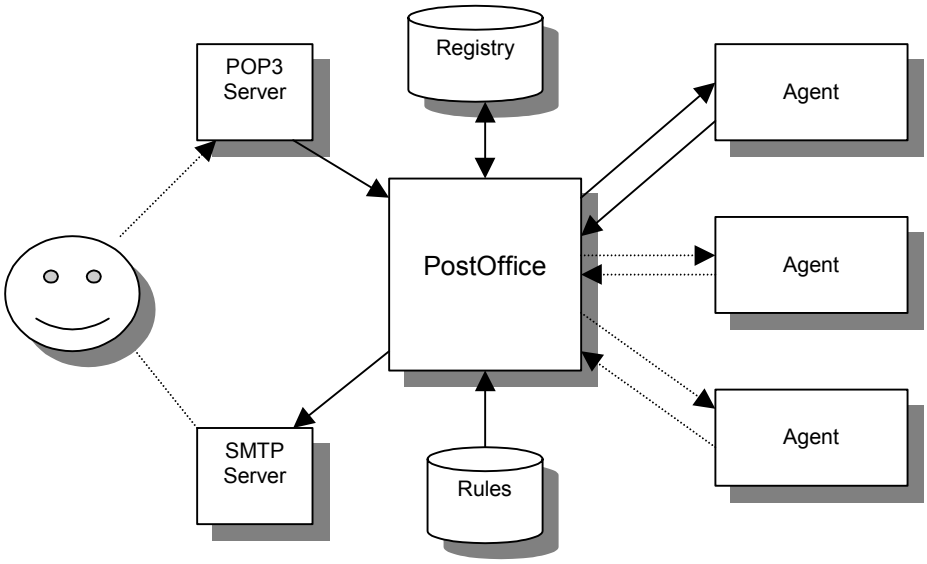

Figure 1: RoboTA architecture

The architecture of RoboTA

At its heart RoboTA is a client - server based system, with one central server, the PostOffice, and multiple clients, or agents. RoboTA uses TCP/IP socket communication for sending and receiving messages between the PostOffice and its agents. Each component has its own designated port on which it listens for incoming messages. All communication with the user will go through the PostOffice. Users will never communicate directly with any agents.

PostOffice

The PostOffice is the central component of RoboTA. It serves the following major functions:

- *Retrieving mail sent by users from a designated POP3 server.*
  RoboTA periodically checks its account on a POP3 server for mail. All new messages will be retrieved at once, stored in a queue and then sequentially processed after the POP3 session is finished.
- *Determining the type of message and the agent for handling it.*
  The content of every new message from the POP3 server has to be tested against a set of rules to determine the agent that handle the message. If the test procedure of a rule fires the accompanying action procedure will return the name of an agent that can handle the message.

- *Forwarding the message to an agent.*
  All email messages that can be handled by an agent will be transformed into a KQML message object and then send over a TCP/IP socket connection to the agent. The PostOffice maintains a registry that contains information about all known agents. A registry entry contains the name of the agent, its port number and the name of the host the agent is running on.
- *Sending the results from an agent back to a user.*
  The PostOffice also receives KQML messages from agents over TCP/IP socket connections. In general, these messages contain responses to the message of a user. It is the job of the PostOffice to forward these results as an email message to the user via a SMTP server.

Agents

While the PostOffice is mainly concerned with forwarding and transforming messages the 'real work' in RoboTA is done by the available agents. Agents for RoboTA have to register with the PostOffice when they start up. It is also required that there is a rule installed which fires whenever a message intended for the agent is received by the PostOffice. The agents are generally message-processing programs that read a text message as their input and generate a reply for the user. It is not required that a reply is send back to the user, however it is highly desirable to give the sender of the original message some feedback about what happened to her message.

Writing a new agent for RoboTA

One of the key features for the design of RoboTA was that it should be easy to extend the functionality of the system. This will allow the use of RoboTA for a wide variety of purposes. Adding a new agent requires two basic steps:

- Writing a rule that matches an incoming mail message and identifies the appropriate agent.
  The rule will be run on the content of every new incoming email message. If the rule matches the content it will return the name of an agent to handle this message.
- Writing an agent that processes the mail message and generates a result for the user.
  The agent receives its input and sends its output as a KQML message over a TCP/IP socket connection. The present version of RoboTA includes a skeleton agent that already provides the functions for communicating with the PostOffice. When writing a new agent there are three major steps:
  - Reading the `:content` field of the received KQML message line by line.
  - Processing the input.
  - Generating the `:content` field for a reply message.

In many cases it is relatively easy to integrate existing programs in RoboTA. Since the reply of an agent can include any arbitrary text many existing systems need just minor changes to transform them into an agent for RoboTA.

Routing Rules

RoboTA uses routing rules to figure out what agents are appropriate for each message. A routing rule for an agent includes two components: A *test rule* and an *action rule*. The test rule ascertains if the message type is appropriate for that type of agent. If the test rule is satisfied, the action rule is used to determine which agent to send it to. (Different classes coached by a particular type of agent may be handled by different instantiations of that agent, for instance.) Routing rules are loaded during the initialization of the PostOffice, and are never modified while the PostOffice is running, to ensure consistent handling of messages. This is a reasonable policy because adding a new type of agent to a RoboTA colony will be a relatively rare event.

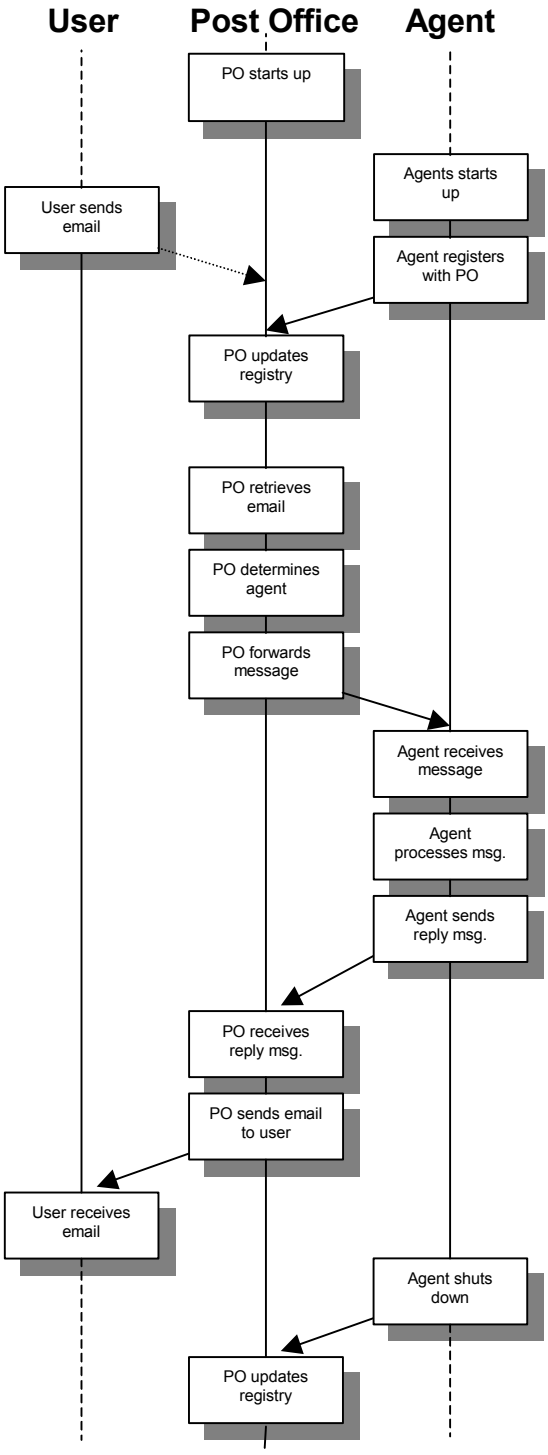

Figure 2: RoboTA message processing

The Sample Agent

The PostOffice will forward each message to every agent deemed relevant by the routing rules. Most agents will be based on the sample agent code contained in the RoboTA Agent Toolkit. The sample agent provides the functions necessary for the communication with the PostOffice for retrieving and sending messages over a TCP/IP socket connection.

The sample code provides a hook for user-defined procedures in PROCESS-KQML-MESSAGES. This procedure takes a number of received KQML messages and processes them sequentially. According to the performative of the KQML message some user-defined procedure is executed. In most cases the agent code should generate and send a reply to the sender of the original message.

An example for `PROCESS-KQML-MESSAGES` can be found in the Unregistered Agent. This agent takes care of returning messages to the original sender when a rule matched the message but no active agent could be found in the registry of the PostOffice. The Unregistered Agent calls the function `GENERATE-REPLY` for the content field of the reply message. This function generates a 'fortune cookie' from a Unix `fortunes` file and returns both the fortune cookie and the original message.

```
(defmethod process-kqml-messages (messages)
   (dolist (msg messages)
      (setf (content msg) (kqml-insert-newline (content msg)))
      (let ((perf (performative msg)))
         (cond ((string-equal perf "achieve")
                (let ((reply-msg (kqml-build-message 'achieve
                                   :sender 'unregistered-agent
                                   :receiver 'post-office
                                   :content (generate-reply msg)))
                      (socket (send-socket *agent*)))
                  (send-message (send-socket *agent*) reply-msg)))))))
```

Before the rule (and possibly the agent code) get submitted to the RoboTA administrator, the author should check that

- the rule only matches the intended messages. It will usually cause problems for the agent if messages get forwarded that aren't intended for the agent. When writing an agent it is in many cases a good idea to check the content of the message again before processing it.
- the agent code generates the correct response.

Once the rule and the agent code have been submitted, it is the responsibility of the administrator of RoboTA to

- check the agent code for the assigned port address. Each agent must have its unique and permanently assigned port address.
- add the rule to the file of routing rules for the RoboTA colony's PostOffice.
- restart the PostOffice to load the modified rule file.

Message Formats

After the PostOffice has retrieved a message from the POP3 server, the message will be transformed into a KQML message object. The structure of these objects resembles the syntax described in Labrou & Finin (1997). We are using a very small subset of the proposed KQML performatives:

- ACHIEVE
  Messages that are forwarded between the PostOffice and any agent use the ACHIEVE performative. Since we are not building up knowledge bases, the use of TELL or INSERT seemed inappropriate. We didn't use FORWARD because it will still need a performative for its :content parameter to be routed to another agent.

  The ACHIEVE performative in RoboTA uses the following parameters:

:sender	Name of the agent or PO (for PostOffice)
:receiver	Name of the agent or PO (for PostOffice)
:language	RoboTA
:ontology	RoboTA
:content	email address of the original sender subject of the email message complete body of the email message

- REGISTER
  When an agent starts up it will send a REGISTER message to the PostOffice. The :content field contains a single line consisting the agent name, it's port number and the name of the host the agent is running on.

  The REGISTER performative uses the following parameters:

:sender	Name of the agent
:receiver	PO
:language	RoboTA
:ontology	RoboTA
:content	name port hostname

- UNREGISTER
  If an agent is shut down for any reason it is supposed to send an UNREGISTER message to remove its information from the PostOffice's registry. The :content field of an UNREGISTER message contains the name of the agent, its port number and the hostname.

Security

In its present state RoboTA does not emphasize security. However, for an open system with agents running on remote machines, security is a requirement. Even without using complicated authentication schemes it is relatively easy to prevent unauthorized agents from connecting to the PostOffice. Keeping the port number secret and using an additional pass-phrase might provide a minimal system security that is sufficient for most applications.

Another level of system security is provided by splitting the routing rules into a test rule (which just tells if a message of the desired type) and an action rule (which yields the name of an agent that should handle the message). Even if the test rule fires on the content of the message, there might be cases in which we don't want to forward the message directly to an agent.

An agent A might only be willing to handle messages that originated from a certain group of senders (say, those registered for the course). All other messages should be returned. The test rule for agent A will fire whenever a message is received that contains information agent A could handle. Even if the message passed the first test, there is a second hurdle: the action rule will check the message for the sender of the message (or any other information). If the message originated from one of the desired senders, the action rule will return the name of the handling agent A. If the sender of the message couldn't be authenticated, the action rule will return the name of the standard return agent and add an error message to the :content field of the message.

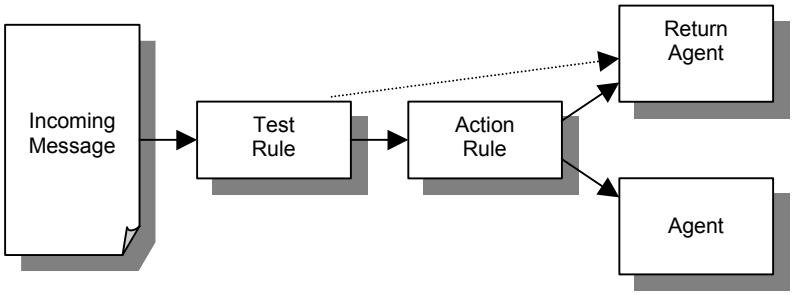

Figure 3: Using rules for additional security

The advantage of this procedure is that the message from an unauthenticated sender will not be forwarded to the agent A. It will not get past the PostOffice or the return agent. Because Agent A will never see the message we eliminate the potential risk of forwarding messages that could compromise the security of an agent.

A member of the first RoboTA colony: The CyclePad Guru

The CyclePad Guru is an agent belonging to the RoboTA colony that handles student requests about their CyclePad assignments. Requests are made via an email facility built into CyclePad. The simplest kind of request is turning in an assignment. An authoring environment available to instructors enables them to provide a set of desiderata that a student's design must satisfy (e.g., produce at least 50 kW of power and cost no more than $100,000). When a student turns in an assignment, the Guru produces a report about how well the student's work satisfies the assignment's desiderata, and uses the Post Office to email these results to the instructor. (Some of our instructors prefer a web-based interface, so we are also extending the Post Office to optionally update a private web site so that instructors can view results that way as well.)

The other requests are requests for advice. The dialog in the email system supports three types of questions:

- "I'm having trouble completing my analysis of the design" (*Stuck* request)
- "I have a contradiction that I can't resolve" (*Contradiction* request)
- "I need to <*increase/decrease*> the <*property*> of <*part of cycle/whole cycle*>" (*Design* request)

The Guru does not presume that the student's request is necessarily appropriate. For instance, a design request only makes sense for a cycle whose analysis is complete and non-contradictory. If the student's design is contradictory then advice about resolving the contradiction is generated, and if the student's design is non-contradictory but incomplete, then advice about how to proceed in the analysis is generated. (Any discrepancy between the student's request and what the Guru does is pointed out to the student as part of the reply.)

Since the algorithms used to generate advice are very domain-specific, we only sketch some highlights here that may be of general interest. Advice for Stuck requests is generated based on a model of analysis strategies in the domain, formulated by extensive interaction with expert instructors. Advice for contradiction requests is very generic, since all of the cases we have identified as commonly occurring in student exercises so far now have special-case contradiction handlers (c.f. Forbus & de Kleer, 1993) installed in CyclePad itself. We fully expect that, by hand-analyzing the contradictions being generated by our now substantially larger user community, we will identify new common patterns of contradictions. Handlers for these contradictions will be tested by incorporating them first in the Guru, and as they prove their worth they will be migrated into CyclePad itself. Advice for design requests is generated by a case-based coach, using cognitively-motivated analogical processing techniques (Gentner & Stevens, 1983). The case library is generated by instructors using CyclePad in "watch me" mode, using a set of dialogs to express what the problem is that the case addresses, and representing the transformation the instructor

makes to solve the problem. The student's design plus the problem they are having with it (e.g., outlet temperature of a turbine being so high that it has to be made of Unobtainium, an extremely expensive material) are used as a probe to MAC/FAC (Forbus, Gentner & Law, 1995), which retrieves potentially relevant cases from the library. (An important reason for using MAC/FAC is that it does not require hand-indexing of cases, which simplifies the cause authoring tools for instructors.) The candidate inferences generated by the analogical match between the student's design and retrieved cases are used to generate suggestions for design improvements. Matching cases whose inferences do not include a transformation are eliminated, and the best inference of those remaining is chosen for generating advice (Forbus, Everett, Gentner & Wu, 1997). In addition to the formal representation of a case generated automatically by CyclePad's instructor environment, a web page describing the case designed for student access and perusal must also be provided for each case in the library. The advice to students includes a URL for the web page corresponding to the case used, as well as how the transformation used in that case can be applied to the student's specific situation.

Another example: The Scheme Coach

The Scheme Coach processes two types of requests: Turning in an assignment and critiquing a student's program. When an assignment is turned in, the student's program will be tested (in a "sandbox" interpreter) on sample data provided by the instructor. The results will be reported to the instructor via email or private web site, as desired. Critiquing a student program involves first using simple rules that detect common types of student mistakes (e.g., putting parentheses around arguments to a procedure). If no mistakes are found at this level, a symbolic evaluation of the student's program will be performed to look for more subtle bugs (e.g., assuming in one part of a program that a value is a number and in another part of the program taking the CAR of this value). The critique is only aimed at finding blatant bugs in student programs; no problem-specific information is used in generating a critique (i.e., we don't try to generate advice about how to make their program more correct). This is a more conservative strategy than generally used in intelligent tutoring systems for teaching programming (Benford et al, 1994). We believe this strategy is appropriate given our plans to use this coach with several courses and with complex examples (e.g., web spider, adventure game).

Discussion

This paper describes RoboTA, an architecture for a colony of agents aimed at supporting instructional tasks. Two critical advantages provided by the agent colony architecture are (1) the sharing of communication features through the Post Office and providing a RoboTA Agent Toolkit will, we hope, simplify the construction of new agents for other courses, and thus facilitate the creation of coaches for more courses. (2) The ability to host agents on different machines means that a RoboTA colony can be implemented using a pool of "cast off" machines, a practical concern that can be critical given the budget constraints of many educational institutions.

Currently the Post Office and the RoboTA Agent Toolkit have been implemented and tested. The CyclePad Guru has been implemented and tested, with current work focusing on extending the case library to cover more of the common situations occurring in design-oriented courses, and embedding the Guru into an Agent by using the Agent Toolkit. The Scheme coach is currently being implemented. We expect the first fully functioning RoboTA colony to come on-line in November 1997, and we will revise the paper accordingly to reflect this and our experiences in having students use it.

References

Forbus, K. & Whalley, P. (1994). *Using Qualitative Physics to Build Articulate Software for Thermodynamics Education*. Proceedings of the 12th National Conference on Artificial Intelligence, Vol. 2. AAAI Press/MIT Press, Cambridge, Mass., pp. 1175-1182.

Forbus, K. (1997). *Qualitative Physics to Create Articulate Educational Software*. IEEE Expert Intelligent Systems & Their Applications, Vol. 12, No. 3: May/June 1997, pp. 32-41.

Stern, M. (1997). *The Difficulties in Web-Based Tutoring, and Some Possible Solutions*. Proceedings of the Workshop on Intelligent Educational Systems on the World Wide Web. 8th World Conference of the AIED Society. Kobe, Japan, August 18-22

Brusilovsky, P., Schwarz, E. & Weber, G. (1996). *ELM-ART: An intelligent tutoring system on World Wide Web.* In Frasson, C., Gauthier, G., and Lesgold, A. (eds.), Proceedings of the Third International Conference on Intelligent Tutoring Systems (ITS-96). Berlin: Springer, pp. 261-269.

Allen, C. & Hand, M. (1992). *Logic Primer*. MIT Press

Allen C. (1997). *The Logic Daemon*. Available at http://logic.tamu.edu

Labrou, Y. & Finin, T. (1997). *A Proposal for a new KQML Specification*. TR CS-97-03, University of Maryland Baltimore County.

Forbus, K. & de Kleer, J. (1993). *Building Problem Solvers*. MIT Press.

Gentner, D. & Stevens, A. (1983), *Mental Models*. Lawrence Erlbaum Associates, Mahwah, N.J.

Forbus, K., Gentner, D. & Law, K. (1995), *MAC/FAC: A Model of Similarity-Based Retrieval*. Cognitive Science, Vol. 19, No. 2, Apr.-June, 1995, pp. 141-205.

Forbus, K., Everett, J., Gentner, D. & Wu, M. (1997). *Towards a Computational Model of Evaluating and Using Analogical Inference.* Proceedings of the 19th Annual Meeting of the Cognitive Science Society, Stanford, August 7-10.

Benford, S. D., Burke, E. K., Foxley, E., Gutteridge, N. & Zin, A.M. (1994), *Ceilidh as a Course Management Support System*, Journal of Educational Technology Systems Vol.22, No.3, pp. 235-250.