GIA: An Agent-Based Architecture for Intelligent Tutoring Systems

Brant A. Cheikes

The MITRE Corporation
Artificial Intelligence Center
202 Burlington Road, M/S K302
Bedford, MA 01730-1420
E-mail: brant@linus.mitre.org

1. Introduction

Intelligent tutoring systems (ITSs) are complex, integrated systems that apply principles and methods of Artificial Intelligence (AI) to problems of education and training. In their role of *knowledge communication systems* (Wenger, 1987), ITSs strive to optimize learning of domain concepts and problem-solving skills by means of adaptive, individualized coached practice on domain tasks. ITS design has been a focus of research since the early days of work on AI, and over the years a variety of impressive results and implemented systems have been documented in the literature, e.g., SCHOLAR (Carbonell, 1970), GUIDON (Clancey, 1983), STEAMER (Hollan *et al*., 1984), the LISP Tutor (Anderson and Reiser, 1985), and SHERLOCK (Lajoie and Lesgold, 1989).

As the field has matured, it has become apparent that research is being impeded by the lack of modular system architectures, reusable components and sharable knowledge bases. Given the current "state of the art," each new research or development effort typically starts from scratch and builds their ITS application from the ground up. This situation creates several problems:

- **High development costs**. Researchers are forced to design their own system architecture, implement all system components, develop knowledge representation strategies and reasoning mechanisms, and acquire and encode all relevant domain and instructional knowledge. While one or more of these tasks may indeed be focal research concerns, others are merely drudge tasks needed to erect enough "scaffolding" to allow work to proceed on the core problem(s) of interest. But as ITS design becomes better understood, an ever-increasing amount of such scaffolding must be put in place before the real work can begin.
- **High maintenance costs**. ITSs today, especially laboratory prototypes, are designed and built as complex, integrated AI applications, with little attention paid to problems of evolutionary development or maintenance. Consequently, as design flaws are uncovered or new functional requirements identified, large portions of the system often need to be thrown out and re-implemented from scratch.
- **Lack of interoperability**. A wealth of general-purpose software tools exists both in the marketplace (e.g., spreadsheets, drawing tools) and in research labs (e.g., planning systems, text generators), but these tools cannot be easily imported and used in ITSs. Market forces are tending towards modular, interoperable software designs, yet little is currently known regarding how ITS architectures can take advantage of these trends.
- **Difficulty to share and compare results**. Current ITS designs employ idiosyncratic knowledge representation formalisms, architectures and internal data and control flow schemes. As a result it can be difficult or impossible to ascertain the strengths and weaknesses of individual tutoring approaches, to identify and reuse functional units across domains, and to share instructional materials and tutoring strategies.

Motivated by these concerns, we have been developing GIA (Generic Instructional Architecture), an agent-based software infrastructure—a shell—intended to support rapid development of ITS applications.

This short paper presents a summary of GIA's design and attempts to identify some of the problems addressed by our agent-based approach.

2. GIA: A Generic Instructional Architecture

This section begins with a summary of the objectives of the GIA project (section 2.1), then gives an short introduction to the general agent-based approach we have adopted (section 2.2). Section 2.3 presents the architecture of the GIA toolkit. In section 2.4 we characterize a specific development/maintenance problem facing the ITS community, and in section 2.5 we discuss how GIA's agent-based design provides a basis for solving that problem.

2.1 Objectives

The objective of the GIA project is to develop modular software architectural components that will provide an evolvable development foundation both for Intelligent Tutoring Systems (ITS) and traditional Computer-Based Training (CBT) systems. Such a foundation should support multiple computing platforms (e.g., Sun workstations, IBM-PC compatibles and Apple Macintoshes), promote reuse of components, support easy integration of components that might not have been specifically designed to be part of an ITS (e.g., commercial device and process simulators), and provide a flexible environment for training research by permitting easy addition, modification and replacement of all system components.

To these ends we have adopted an *agent-based software engineering* methodology. We believe that ITSs can usefully be decomposed into collections of independent agents with the ability to exchange requests and information using an expressive formal language, and that such a decomposition will foster the modularity and interoperability which we seek. The key to our success, however, lies in the language design: we need to ascertain both the quality and degree of expressivity that is required of such a language to support the demands of ITS applications. That difficult research task remains in its preliminary stages; the remainder of this paper focuses on GIA's high-level design.

2.2 An overview of agent-based design

In developing GIA we are building on the results of the ARPA-sponsored Knowledge Sharing Effort (KSE). (For more information on the KSE, the reader is directed to the World-Wide Web homepage at `<http://www.cs.umbc.edu/kse/>`.) In brief, an *agent* is defined behaviorally as "any piece of software that communicates using an Agent Communication Language" (ACL). Working groups of the KSE have defined an ACL that comprises three parts: a protocol layer called the *Knowledge Querying and Manipulation Language* (KQML) `<http://www.cs.umbc.edu/kqml/>`, a content layer called the *Knowledge Interchange Format* (KIF), and a domain-specific ontology that provides a semantic grounding for content-layer expressions.

KQML is a message format (currently with a Lisp-like syntax) that also defines a set of linguistically-influenced, domain-independent *performatives* corresponding to semantically-distinct message categories. Examples of KQML performatives are ASK-IF, REPLY, TELL, and ACHIEVE. KQML "wrappers" enclose the message content, which is expressed in KIF. (KIF is a formal language based on first-order logic.) Agents use KQML/KIF to convey their states, beliefs, etc., to make requests of various kinds, and to coordinate their activities. In the context of an ITS, one might imagine the following message interchange between a pair of system components:

```
(ask-if :sender instructional-planner :receiver student-model
        :reply-with q46
        :content '(knows student
                     (<= (standard-miles 100)
                         (kappa ?d (altitude low-earth-orbit ?d))
                         (standard-miles 300)))
```

```
(reply :sender student-model :receiver instructional-planner
       :in-reply-to q46 :content true)
```

The first message, sent from a (hypothetical) instructional-planning agent to a student-modeling agent, uses KQML's ASK-IF performative to ask whether the receiver believes the embedded content expression (which we might gloss as, "the student knows that the altitude of a low-earth orbit is between one hundred and three hundred standard miles"). The recipient in this example uses the REPLY performative to respond in the affirmative.

2.3 GIA's agent-based design

In the GIA shell, each component in the ITS is designed as an agent that uses ACL messages to issue requests and respond to queries from other component agents. The architecture distinguishes two major functional units, the *kernel* and the *student interface*. Each unit consists of a set of independent agents. The agents comprising the kernel (Figure 1) provide the essential services of an intelligent tutoring system. These services include student-record administration (Training Administrator), curriculum and lesson planning (Curriculum Manager), activity/exercise administration (Activity Manager), coaching-support decision making (Coaching Planner), and modeling of student knowledge and problem-solving behavior (Student Model). This is not the only possible functional breakdown for an ITS's "intelligence", but it is an advantage of the agent-based approach that it does not demand we commit to one.

The student interface is modeled as a collection of *interface agents* (Figure 2), each responsible for managing the visual presentation and behavior of the instructional resources used by the student. The Executive (on the kernel side) and the Screen Manager (on each student's workstation) facilitate the exchange of KQML messages across the network.

This modular approach to interface design seems very natural in the context of an agent-based system, yet it is significant in that it cleanly addresses one of the problems of the ITS community leading to high maintenance and development costs. This problem is described next.

2.4 The problem of throw-away interfaces

As ITSs have grown in scope and sophistication, their interfaces have become commensurably complex and elaborate. Often the student interface aims to simulate or approximate the actual tools or devices that the student is being trained to operate. In order to observe, record and model the student's problem-solving behavior, ITS designers typically require fine-grained control over the interface. This is because every operation on an interface widget (e.g., a button click or check-box selection), indeed every mouse movement, might be of interest to the reasoning component of the ITS. (Henceforth I will refer to the ITS reasoning component as the *ITS kernel*, or simply the *kernel*.)

In experimental ITS prototypes, this demand for fine-grained control is usually addressed by giving the kernel direct access to the application programmer interface (API) of the software toolkit used to implement the interface. For example, the module of the kernel that is concerned with authenticating students' identity might directly invoke the window-management routines needed to display an appropriate data-entry form to the student. During problem solving, the coaching module might similarly manipulate the interface directly through the API should it decide to perform (and visually demonstrate) the next step for the student.

In such an approach, the implementations of the student interface and the ITS kernel can easily become inextricably intertwined. Assumptions about the interface and the instructional model that it supports may become so deeply embedded in the design of the kernel that any attempt to reuse it in a different ITS application would necessitate extensive revisions.

Clearly, while such an approach might be entirely reasonable for an exploratory ITS research project, it is undesirable for an ITS shell design. An ITS shell like GIA should be able to support the development of a variety of ITS applications, where each may have quite different interface requirements. Thus it is crucial for designers of ITS shells to devise techniques that (1) enable the reasoning component to manipulate the interface "at a distance" using appropriate abstractions, and (2) support the reuse of ITS kernel components across interfaces of varying designs. We believe that the agent-based architecture being developed for GIA is promising in both respects. We discuss this point further in the next subsection.

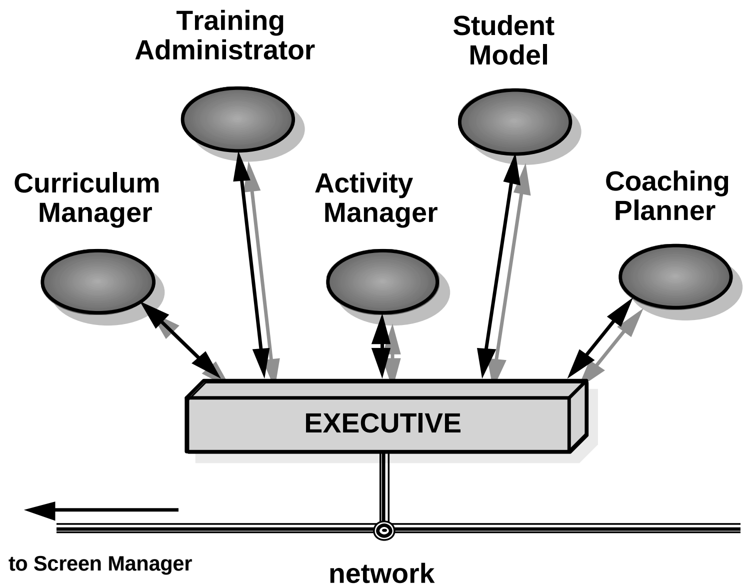

Figure 1: The GIA Kernel Agents

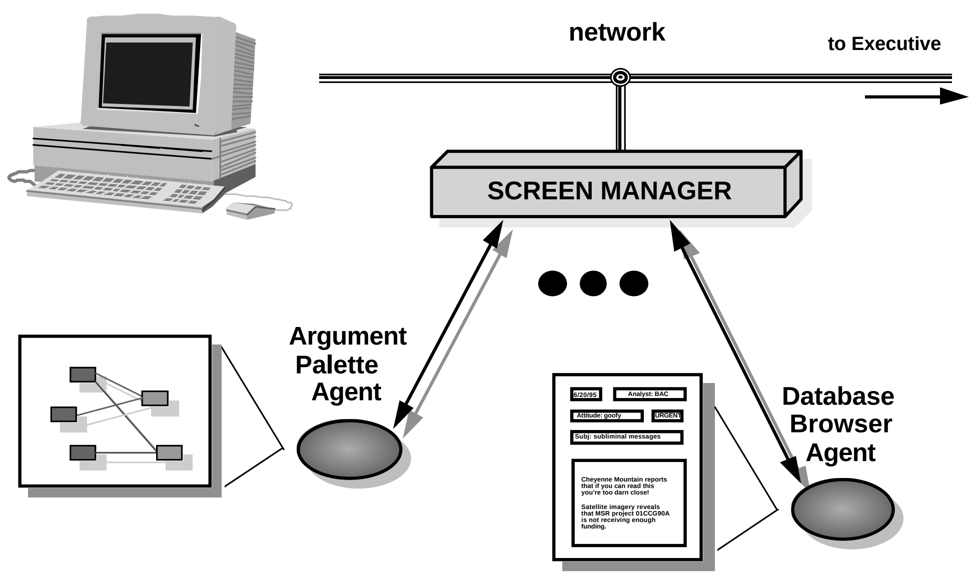

Figure 2: Student Interface Agents

2.5 GIA's agent-based interface

A key feature of GIA's design is that the student interface is no longer viewed as a monolithic application, but rather as a collection of (more or less) independent agents, each having an associated visual presence. In general, GIA interfaces follow a *workspace paradigm* where students are situated in a graphical environment that provides them with a variety of tools that they can use as needed to solve problems and carry out their assigned tasks. The workspace paradigm conveniently lends itself to an agent-based organization. Each tool in the workspace is managed by a separate agent. Each agent controls its visual component, and manipulations of that component by the student are reported by the controlling agent to the kernel (via the Screen Manager) in messages encoded in KQML/KIF. The semantics of the KIF content expressions are defined in a common knowledge base (KB) accessible to all GIA agents. This common KB contains, *inter alia*:

- definitions of terms for describing elements of the interface (the *interface ontology*);
- definitions of terms for describing agents and properties of agents in the GIA-based system (the *system ontology*);
- an ontology of the events that interface agents may "experience" due to student actions on their visual components (the *event ontology*);
- for each ontology, definitions of valid relations in which instances of terms may participate.

For example, the interface ontology defines terms such as *window*, *widget*, *button* (a subclass of widget), etc.; the system ontology defines the agents shown in Figures 1 and 2; the event ontology defines events such as *student-login* (which occurs when the student completes the login form presented by the *session login* interface agent) and *add-evidence* (which occurs when the student records a piece of evidence on the problem-solving tool called the *argument palette*, used for constructing representations of arguments for/against particular hypotheses). The *state* relation is defined to associate instances of windows with terms encoding their current state (e.g., *open* or *iconified*); the *login-name* and *password* relations are defined to associate instances of student-login events with the string representations of the login name and password entered by the student.

In combination with the KIF language definition published by the KSE (Genesereth *et al*., 1994), these ontologies define the space of valid KIF expressions that GIA agents may embed in KQML messages and use to describe their beliefs and requests. In this way, it is possible to design the ITS reasoning components to manipulate the interface in a manner abstracted from the details of any particular windowing system. Note that "ontology engineering" remains a challenging research problem, and useful ontologies are likely to evolve only through cycles of testing and refinement. We have developed an preliminary set of ontologies for use in GIA, and we will be evaluating them over the coming year as we apply GIA to real tutoring applications.

3. Current Status

GIA's core message-passing infrastructure has been implemented in ParcPlace Systems' VisualWorks Smalltalk (release 2.0), and runs on Sparcstations (SunOS and Solaris) and PC-compatibles (Windows 3.1, NT 3.5 and OS/2 Warp). Figure 3 (below) shows a snapshot of the GIA kernel in operation.

We have implemented support for these KQML performatives: TELL, ASK-IF, ASK-ABOUT, REPLY, ADVERTISE, SUBSCRIBE, and MONITOR. All system agents consult the shared knowledge base to lookup their "interests" (system events whose occurrence they wish to be informed of) and "abilities" (services they can provide to other agents). Agents issue SUBSCRIBE performatives to their facilitator for each of their interests, and ADVERTISE performatives for each of their abilities. By default, agents that "experience" events do not routinely issue corresponding TELL messages unless they have previously received an associated MONITOR message. Since agents (especially those that control interface elements) typically may experience many more events than might be of interest in any given ITS context, our use of explicit MONITOR messages to enable event reporting on demand helps us reduce the flow of unnecessary messages within the system.

We are currently beginning to apply GIA to a real ITS application. We expect this effort to yield substantial insight on the use of KQML/KIF to support the sorts of agent collaborations needed to provide ITS services.

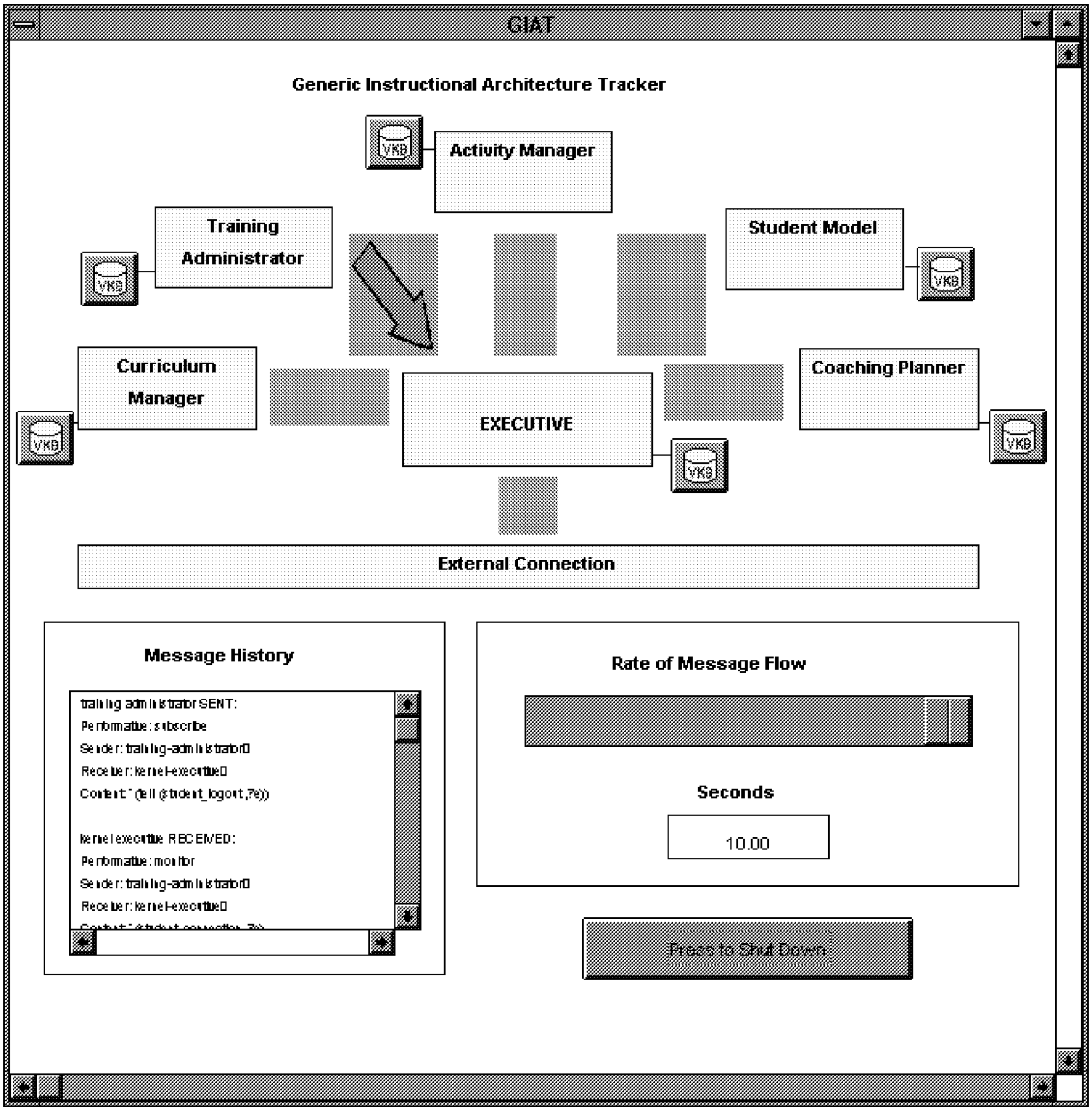

Figure 3: Screen image of GIA Kernel

4. Concluding Remarks

The GIA agent-based architecture aims to address problems of high development/maintenance costs and lack of interoperability associated with conventional ITS development approaches. Our method of breaking the interface up into agents that control individual instructional resources appears to be a particularly promising means of substantially reducing the effort involved in building new interfaces.

Overall, the key to GIA's success as an ITS infrastructure will depend on the generality and reusability of its ontologies of agents, and tutoring system and domain events, concepts and relations. A preliminary set of these ontologies has been built and we are in the process of refining them as we apply GIA to real tutoring applications.

5. References

Anderson, J. R. and Reiser, B. J. (1985) The LISP Tutor. *Byte*, vol. 10, no. 4, pp. 159–175.

Carbonell, J. R. (1970) AI in CAI: an artificial intelligence approach to computer-assisted instruction. *IEEE Transactions on Man-Machine Systems*, vol. 11, no. 4, pp. 190–202.

Clancey, W. J. (1983) GUIDON. *Journal of Computer-Based Instruction*, vol. 10, no. 1, pp. 8–14.

Generesereth, M. S., and Fikes, R. E. (1992) *Knowledge Interface Format Version 3.0 Reference Manual*. Logic Group Report Logic-92-1, Stanford University, Computer Science Department, Stanford.

Hollan, J. D., Hutchins, E. L., and Weitzman, L. (1984) STEAMER: an interactive inspectable simulation-based training system. *AI Magazine*, vol. 5, no. 2, pp. 15–27.

Lajoie, S. P. and Lesgold, A. (1989) Apprenticeship Training in the Workplace: Computer-Coached Practice Environment as a New Form of Apprenticeship. *Machine-Mediated Learning*, vol. 3, no. 1, pp. 7–28.

Wenger, E. (1987) *Artificial Intelligence and Tutoring Systems: Computational and Cognitive Approaches to the Communication of Knowledge*. Los Altos: Morgan Kaufmann.