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 Artificial Intelligence (AI) techniques to problems of education and training. In their role as *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 an active focus of research since the early days of AI, and over the years a number of impressive results and prototype systems have been discussed in the literature, including SCHOLAR (Carbonell, 1970), GUIDON (Clancey, 1983), STEAMER (Hollan *et al.*, 1984), the LISP Tutor (Anderson and Reiser, 1985) and SHERLOCK (Lajoie and Lesgold, 1989).

Although research in ITS has produced full systems such as those listed above, researchers have understandably expended the bulk of their efforts developing enabling technologies for ITS, e.g., student modeling and lesson planning techniques. In fact, descriptions of complete ITSs—systems that attempt to teach some body of material however large or small—appear quite infrequently in the ITS literature taken as a whole.

While this might ordinarily be considered a perfectly normal stage in the development of an emerging technology, in recent years, forces both within and outside the ITS community have begun to push the field toward more widespread development of full- or near-full scale systems and prototypes. Within the field, an increasing emphasis is being placed on *evaluation* of ITSs. Evaluation studies typically require researchers to collect data on how their systems affect real students, and this demands that the systems in question be reasonably complete and robust. In the world at large, there is both a need and a market for tools and techniques that improve public education and/or industrial training. In 1994 alone, U.S. businesses spent approximately $50.6 *billion* on training (up 5% from 1993): $36.5B on salaries for training staff, and $14.1B on hardware, materials, facilities, and seminars/conferences (Industry Report, 1994). In addition, with funding agencies facing shrinking budgets, researchers are under pressure to defend continued investment in ITS technology. To do so, sponsors must be convinced that ITSs are feasible and commercially viable, and to be convinced, they often need to see complete, running systems.

Unfortunately, because development of complete ITSs today takes too long (on the order of years) and costs too much (many hundreds of thousands of dollars), such systems remain few and far between. There are several underlying causes for this, some unique to ITSs, and others common to AI-software development efforts more generally. First, ITSs have typically been designed as custom software applications, meaning that each system is built from the ground up starting from scratch. For each new system, researchers must design their own system architecture, implement all required components, develop knowledge representation and reasoning approaches and tools, and acquire and encode all relevant domain and instructional knowledge. All this work takes time and the efforts of teams of scientists and engineers.

Second, a "made to order" mentality pervades the ITS community, and thus sharable design tools, system architectures and conventions do not exist. Instead, ITS builders employ idiosyncratic knowledge representation formalisms, architectures and internal data and control flow schemes. As a result, the software and methods developed at one institution rarely can be reused by developers elsewhere, leading to costly duplication of effort. Surprisingly, the notion that ITSs should be built from modular, reusable components seems to be a rather foreign idea in the ITS community despite its wide acceptance in the commercial software world.

Third, ITSs are rarely designed to evolve. Yet ITS development is widely understood to be a process of iterative refinement. As a result, it is common for developers to discard large chunks of software on each development iteration. Each line of code discarded represents costly project overhead. While some of this code must necessarily be thrown out, the overall amount of discarded code might be reduced if the system were to be designed from the start in a modular fashion that isolated dependencies as much as possible.

Fourth, ITS designs today are largely unable to incorporate commercial "off-the-shelf" (COTS) software. (Though some recent work in this direction has been described by Lentini *et al*. (1995) and Ritter and Koedinger (1995).) Examples of such COTS software include word-processors, spreadsheets, and simulators. With complex information systems proliferating in the workplace, there is great demand for "embedded" training, i.e., training systems embedded in the information systems themselves. The problem for ITS developers is that until recently, COTS packages have had limited external interfaces (or none at all). This made it difficult or impossible for ITSs to effectively monitor and interpret student actions on the COTS application. The response from developers has been either to ignore embedded training as a research area, or to spend time and money building mockups of the actual COTS packages for use in the ITS environments.

While these are not all the causes for long, costly ITS development cycles, they are among the most serious. The secondary effects of such lengthy cycles are also serious, and should be of concern to researchers: (1) evaluation efforts are hampered because the cost of building evaluable systems is prohibitive, and (2) ITSs risk being viewed as "not ready for prime time", possibly infeasible and unworthy of investment. The ITS community cannot afford to have its work pigeonholed as a laboratory novelty. We need to find ways to build ITSs more rapidly and at less cost.

Motivated by these concerns, we have been developing GIA (Generic Instructional Architecture), an agent-based software infrastructure intended to support rapid development of ITS applications. While some ITS-development costs and sources of delay are relatively fixed, such as the costs of knowledge acquisition and representation, we believe that others can be substantially reduced, in particular, the costs and time required to implement ITS software. Our approach is to develop techniques that enable and encourage ITS developers to share and reuse their software. This paper presents a summary of our work to date on GIA and discusses some of the problems our approach promises to address and those that it leaves open. In section 2 we set the context for our work with a general overview of agent technology and agent communication. Section 3 presents a detailed description of GIA, and section 4 presents conclusions, open problems, current status and future work.

2. A Brief Overview of Agent-Based Design

In the GIA project we are applying and extending *agent technology* developed by the ARPA-sponsored Knowledge Sharing Effort (KSE).[1] In brief, an *agent* is defined behaviorally as "any

[1] For more information on the KSE, see the web page at <http://www.cs.umbc.edu/kse/>.

piece of software that communicates by means of messages encoded in an Agent Communication Language" (ACL). KSE working groups have developed an ACL having three parts: a protocol layer that defines message syntax and methods of information exchange, a content layer that contains the actual information being exchanged, and a domain-specific *ontology* that provides a semantic grounding for expressions in the content language. We discuss these three layers next, and conclude the section with an example.

2.1 ACL protocol layer

The KSE is developing a particular language called KQML (Knowledge Query and Manipulation Language) for use at the ACL protocol layer. The KQML specification (Finin *et al.*, 1992) defines both a message format and a set of linguistically-motivated, domain-independent *performatives* corresponding to semantically-distinct message categories.[2] Examples of KQML performatives include **tell, ask-if, reply** and **achieve**. In this subsection we summarize KQML's key ideas.[3]

KQML messages consist of several parts, some required, some optional. Every message must contain a performative. KQML messages typically include fields identifying the message's sender and intended recipient, and may additionally include information useful for relating replies to requests. The KQML specification assumes both that agent communication is *reliable*, meaning that messages do not get lost or garbled in transit, and that message transfer is asynchronous with respect to agent operations.

Of course, every message has some content, the information being **tell**ed, the query being **ask-if**ed, a description of the goal to **achieve**, etc. KQML, however, is neutral to the method used to encode the message content. The content could be specified in anything from arbitrary strings to industry-standard SQL expressions. KQML does, however, provide facilities for agents to identify the language in which the message's content is encoded. In some sense KQML acts as a "wrapper" around expressions in the chosen content language.

2.2 ACL content layer

In a related KSE activity, Genesereth and Fikes (1992) have developed KIF—Knowledge Interchange Format—for encoding the content of KQML messages. KIF is a language based on first-order predicate logic that contains a number of extensions to support meta-level reasoning and non-monotonic inference. The KIF specification defines the language's syntax, semantics, and a set of useful operators. Although KQML makes no commitments to the choice of content language, the KSE seeks to promote KIF as a standard. Further details on KIF are available from the KSE web page.

2.3 Ontologies

Although KIF gives us a syntax and semantics for content expressions, it does not presume to specify the terms, relations, functions, etc., needed to meet the needs of arbitrary agent-based systems. These elements must come from an *ontology*. Ontologies define the terms, relations, functions, etc., needed to model particular domains. Although *ontology engineering* is still something of an art, a number of useful heuristics and guiding principles have been developed. One of the goals of the KSE is to promote the development of modular ontologies that can be shared and reused across AI applications. Gruber (1993) provides a nice discussion of the work on sharable ontologies being conducted as part of the KSE.

---

[2] At least the *intention* is that message categories be semantically distinct. Cohen and Levesque (1995) have identified several meaning-related problems with the current KQML specification.

[3] For full details, see <http://www.cs.umbc.edu/kqml/>.

2.4 Example

As an example of how KQML and KIF might be employed by an ITS, consider the following hypothetical message exchange between a pair of ITS components:

```
(ask-if :sender instructional-planner :receiver student-model
        :ontology orbital-mechanics :language kif :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
       :language kif :in-reply-to q46 :content true)
```

The first message, sent from the instructional-planning component to the student-modeling component, uses KQML's **ask-if** performative to ask whether the receiver believes the embedded content expression (which might be glossed 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. Note that the KQML "wrapper" permits the sender to specify that the message content is encoded in KIF, and that the embedded terms and relations are defined in the "orbital mechanics" ontology (which both the sender and receiver should understand). The terms **standard-miles**, **altitude** and **low-earth-orbit** should be part of that ontology.

This concludes our summary of the general agent framework developed by the KSE. In the next section we describe how these ideas are being applied in GIA.

3. GIA: A Generic Instructional Architecture

The objective of the GIA project is to develop both an architecture and a set of modular software components that will provide an evolvable development foundation both for ITSs and traditional computer-based training systems. To this end we are pursuing an *agent-based software engineering* approach. Our hypothesis is that ITSs can effectively be decomposed into collections of independent agents that collaborate and exchange information using an expressive formal language, and furthermore, that such a decomposition will promote modularity and interoperability. The key to our success, it should be clear, lies in the language design: our principal research task is to identify the properties that are required of such a language to meet the demands of ITS applications. While we do not yet have a complete answer to this question, the work of the ARPA KSE on agent communication language appears to have provided the tools we need to produce one.

Although we are building on the work of the KSE, we are not adopting their approach completely. The mission of the KSE is to develop technology suited to the needs of arbitrary distributed, collaborating heterogeneous intelligent agents. As such, we believe it is more complex and powerful than is required for the more limited needs of ITS applications. In the GIA project we are working to identify the minimal set of KQML constructs and KIF language features that ITS developers need to have. In addition, we are beginning to develop ontologies that we hope will capture properties of ITS systems and domains of instruction that can be reused across many different ITS applications.

This section describes GIA in more detail. We begin in section 3.1 with a discussion of several key ideas on which our approach is founded. Section 3.2 describes GIA in detail, and section 3.3 illustrates the system's operation with an annotated example.

3.1 Key elements of GIA's system design

In the GIA project we are adapting the work of the KSE to the needs of ITSs. In this section we discuss the agent communication language we are using, and introduce the concepts *federated architecture*, *facilitators*, *agent community*, *virtual knowledge* base, and *common knowledge base*.

3.1.1 Agent communication language

In GIA we are using a subset of the performatives and parameters defined in the KQML specification. Our choice of the subset is being driven by in-house ITS prototyping efforts; as we are just beginning, it is too early to say whether we need to reduce or enlarge the set. Thus far we have found it necessary to support these performatives: **achieve, advertise, ask-about, ask-if, monitor, reply, sorry, subscribe, tell**. Together, these performatives enable *agent self-description* (section 3.1.2), *content-based routing* (section 3.1.2) and basic information exchange. With respect to KQML message parameters, we dropped the **:language** keyword because for our purposes we can assume that all agents will speak KIF; we dropped the **:ontology** keyword because we are assuming that all agents in GIA will share a common knowledge base (section 3.1.5). Because GIA is designed to simultaneously manage the instruction of multiple students, we have added a **:context** parameter which permits us to designate the particular student to which messages are related.

As just mentioned, all GIA agents will encode message content using KIF. Once again we are striving to identify the minimal features of KIF that we need. For the present we have restricted content-language expressions to the power of Horn-clause logic. We have also found it useful to modify the syntax of KIF slightly so that we can syntactically distinguish between constant symbols, domain terms, and KIF reserved words.

3.1.2 Federated architecture and facilitators

In GIA we have adopted a *federated architecture* for agent communication (Genesereth, 1993). In such an architecture, a distinguished agent called a *facilitator* provides services to other agents, including other facilitators. A key service provided by facilitators is *content-based routing* (CBR), that is, routing of messages among agents based on interpretation of the message content. In order to support CBR, the agent communication language must provide facilities for *agent self-description*. That is, agents must be able to inform the facilitator of their (informational) needs, and to describe the services that they are willing to provide to other agents.

Traditional ITSs do not have facilitator-like components that manage data exchange among the system's modules. Rather, system components communicate directly with one another, i.e., data and control-flow paths are fixed and embedded in the code. In GIA, facilitators make it easy to add and remove system components and change data and control paths accordingly.

3.1.3 Agent community

An *agent community* consists of a set of ordinary (non-facilitator) agents served by a common facilitator. All agents in a community typically reside on a single computing host (though this is not strictly necessary). Each agent in the community (including the facilitator) has a unique symbolic name. Each ordinary agent knows both its own name and that of its facilitator. The agents served by a facilitator are said to be in the facilitator's *jurisdiction*.

GIA supports two configurations, *standalone* and *networked*. In the standalone configuration, two sets of agents—the *kernel agents* and the *interface agents* (see section 3.2.1)—are served by a single facilitator called the *standalone executive*. This configuration corresponds to the traditional ITS design in which the system manages the instruction of a single student. In the networked configuration, the *kernel* agent community communicates over a high-speed network with one or

more *interface* agent communities. The *kernel executive* and *screen manager* serve as facilitators for the kernel community and interface communities respectively. This configuration allows a single set of core reasoning components (the kernel) to manage the instruction of multiple remote students simultaneously. The agent-based approach employed in GIA makes it easy to support these two configurations; for traditional ITS design approaches, it would be both conceptually and practically much more difficult to develop a single ITS that could support both one-to-one and one-to-many modes of instruction.

3.1.4 Virtual knowledge bases

Agent communication implies information storage and retrieval by the agents. Each agent is viewed as maintaining a private, local knowledge base. Agents use their local knowledge bases to keep track of their state, information they have been given by other agents, etc. In GIA, all agents must behave as if their local knowledge base contains sentences in the common content language.

It is not necessary that agent *implementations* explicitly maintain local knowledge bases in this way. Indeed, for matters of efficiency, simple agents might be built to manipulate a small number of state variables rather than manage a collection of symbolic expressions representing content-language statements. Thus we call these knowledge bases *virtual knowledge bases* (VKBs). (In the GIA implementation to date, we have found it convenient to implement a knowledge-base management system based on Horn-clause logic that is used by all agents.) Information about the contents of agent VKBs is available to other agents only via queries in the agent communication language. Use of the VKB is at each agent implementer's discretion.

By default, agents do not inform other agents when they modify their VKB. We use the KQML **monitor** performative to selectively enable reporting of VKB changes. This is intended to control the amount of message traffic that complex agents might otherwise generate.

3.1.5 Common knowledge base

In GIA, every agent has access to a common knowledge base (CKB). The CKB contains at least shared *terminological* knowledge (knowledge about the terms and relations used to describe the world), but may (and is likely to) contain *assertional* knowledge (facts and rules) as well. The CKB supplements each agent's VKB. The CKB is "read only" to prevent agents from communicating indirectly with one another through operations on the CKB.

Four ontologies are fundamental to systems built using GIA, and are expected to be contained in the CKB: (1) the *system ontology*, (2) the *pedagogy ontology*, (3) the *domain ontology*, and (4) the *student ontology*. The system ontology defines the terms and concepts needed to describe the system itself. (By designing GIA agents to configure themselves using information found in the CKB, we make significant gains in software reusability and ease of modification.) The pedagogy ontology defines the terms needed to describe instructional concepts and processes, particularly decisions about topic/exercise selection and coaching/feedback. The domain ontology defines the terms and concepts of the subject-matter domain (e.g., orbital mechanics, LISP programming). Finally, the student ontology defines the terms and concepts that are used to construct the student model.

Populating these ontologies poses many challenges. Thus far in the GIA project we have developed a reasonably mature system ontology (the essence of which is described herein) and rudimentary pedagogy and student ontologies. While we believe that the system ontology can be reused nearly unchanged across a wide range of ITS applications, we make no such claims regarding our pedagogy and student ontologies. Indeed, development of these latter ontologies should be a major thrust for ITS research activity.

With these conceptual tools we can now turn to the details of GIA.

3.2 GIA in detail

Thus far we have motivated the GIA project and discussed its central idea, namely, the application of agent technology to foster the development of sharable and reusable ITS-specialized ontologies and components. In the previous subsection, we described several ways in which we are adapting the KSE's agent-communication scheme to fit the more restricted needs of ITS. In addition, we described the particular class of agent architecture—the federated architecture—that we have selected for use in "GIA-based" ITSs.

To evaluate these ideas, we have been implementing an ITS testbed following the GIA philosophy. The testbed provides facilitator-supported communication among agents in our restricted agent communication language, and in addition commits to a particular functional decomposition of ITS processes into agents. (Note that the agent-based approach does not force us to commit to a particular functional analysis.) Section 3.2.1 presents an overview of our testbed architecture, section 3.2.2 describes the system's basic operating principles, section 3.2.3 summarizes the system ontology, and section 3.3 presents an example of the system in operation.

3.2.1 Testbed architecture overview

Our GIA testbed, illustrated in Figures 1 and 2, currently supports a networked configuration (cf. section 3.1.3). Figure 1 shows the agents comprising the kernel community, Figure 2 the agents comprising the interface community.

The agents comprising the GIA testbed 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).

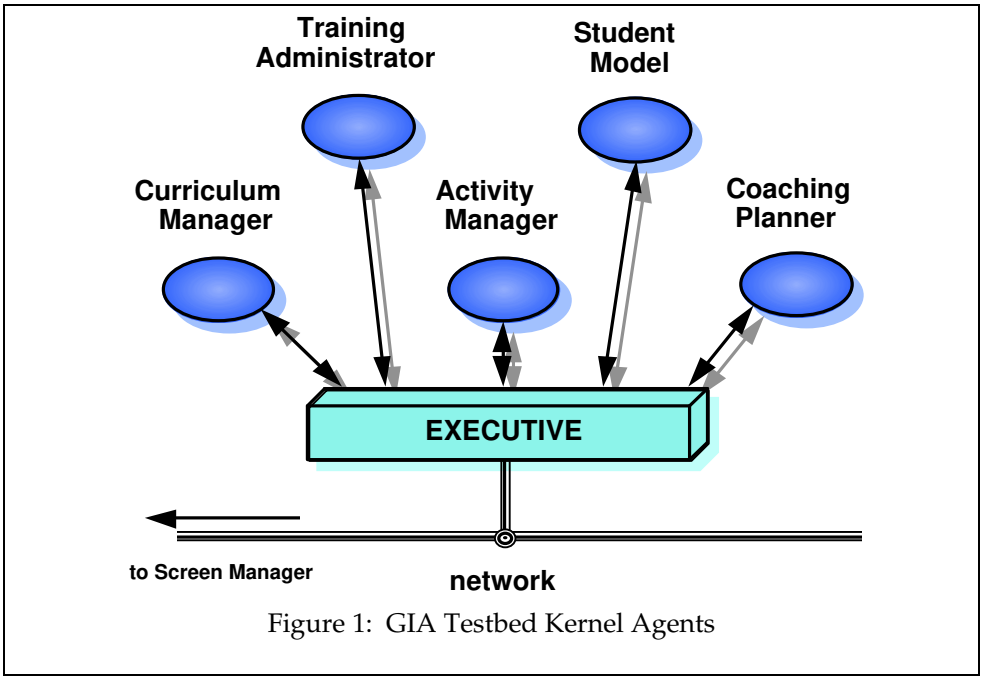

Figure 1: GIA Testbed Kernel Agents

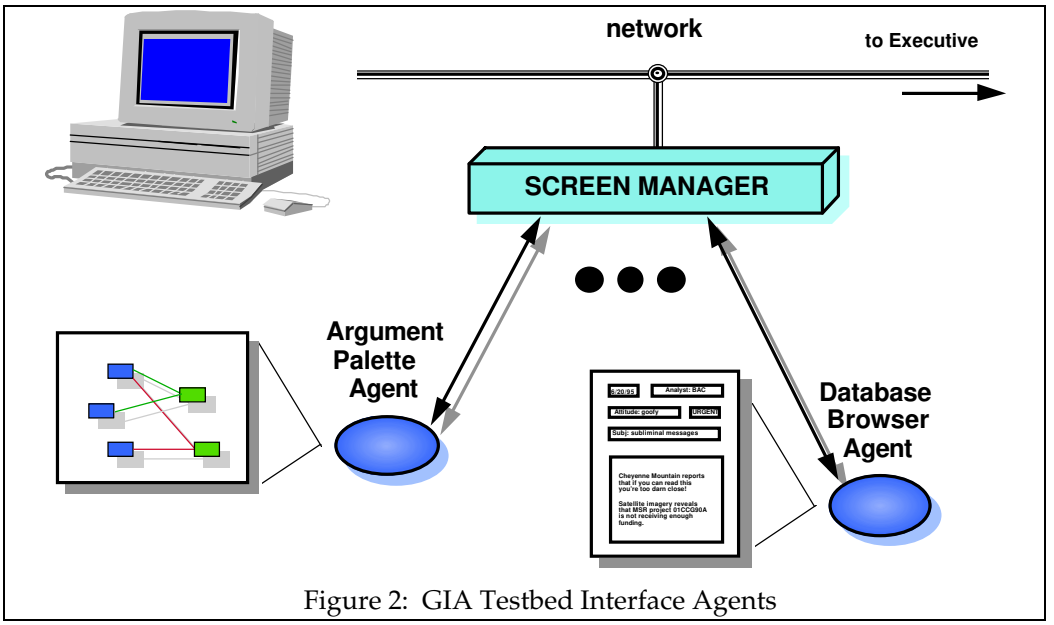

Figure 2: GIA Testbed Interface Agents

A novel aspect of the GIA approach is that we model the student interface not as a monolithic application, but rather as a collection of *interface agents* (Figure 2), each responsible for managing the visual presentation and behavior of individual 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.

The next section summarizes the major operating phases of the networked configuration of the GIA software that we have implemented to date.

3.2.2 Theory of operation

In the networked configuration, the GIA testbed kernel must be activated first. Before the kernel can accept connections from remote screen managers, it must proceed through these steps:

1. **Initialization of kernel executive.** When the kernel executive (henceforward, the executive) is activated, it connects to the common knowledge base and looks up its description. (The relations used to encode agent descriptions are described in section 3.2.3.) One of the first things it retrieves is its jurisdiction, a set of pointers to other agent descriptions in the CKB.

2. **Initialization of kernel agents.** Once the kernel executive knows which agents are in its jurisdiction, it proceeds to activate each one in turn.

3. **Agent startup.** When an agent is activated, it connects to the CKB and looks up its description (to do this, each agent is hardcoded to know its own "name" in the CKB). In that description, the agent will find statements of its *interests* and *abilities* (more on this in section 3.2.3). Based on these statements, the agent constructs appropriate KQML **advertise** and **subscribe** messages and sends them to its facilitator (in this case, the kernel executive).

4. **Processing of advertise/subscribe messages.** The kernel executive processes the **advertise** and **subscribe** messages that begin to flow in from the newly-activated kernel agents. The executive constructs tables of abilities and interests based on **advertise** and **subscribe** messages respectively. When the executive notices that one of its agents has **subscribe**d

to an event report that another agent has **advertise**d an ability to generate, the executive sends a **monitor** message to the original advertiser. This tells the advertiser to issue **tell** messages whenever it records events of the requested type.

Once these steps have been completed, the executive begins listening for incoming network connections. When a remote screen manager connects, a brief handshaking interaction occurs during which the executive and screen manager identify themselves to one another, after which the executive forwards its table of interests to the screen manager. This information enables the screen manager to activate event reporting as needed by its local agents to satisfy interests that have been expressed by kernel agents.

Message passing in the GIA testbed differs from the general agent-communication framework (defined in the KQML specification) in two respects. First, the executive does not transfer its table of abilities to remote screen managers when they connect. Second, remote screen managers do not transfer either of their tables to the executive. The executive does not need to inform remote screen managers of local (kernel agent) abilities because (in our implementation) any interface-side request that cannot be locally satisfied (by an interface agent) is automatically passed on to the kernel executive. The kernel does not need to know about remote interests or abilities because in our implementation kernel agents are able to specify (via the KQML **:context** parameter) the particular screen manager to whom a message is to be directed. (This means that content-based routing is not applied on the kernel side to determine which of several remote screen managers should receive a particular message; that information is recorded explicitly in the message. Once the message reaches the remote screen manager, however, content-based routing is used to direct the message to the proper recipient(s).) Whereas these simplifications might be undesirable in general, they are adequate for the demands of an ITS infrastructure.

3.2.3 System ontology

Using the GIA testbed as a foundation for building an ITS commits the developer to certain design decisions. These decisions can be expressed and are encoded in the system ontology. The system ontology describes the kinds of entities that comprise a GIA-based system and the relationships that may hold among them. This ontology also has implications for system operation. We describe the main elements of the system ontology in the following paragraphs.

3.2.3.1 Types of agents

Agents in a GIA-based system are of one of three types: **facilitator**, **kernel agent** or **interface agent**. In the networked configuration, two subtypes of **facilitator** are in operation, the **kernel executive** and the **screen manager**. In the standalone configuration, all non-facilitator agents are served by the **standalone executive**.

3.2.3.2 Agent interests and abilities

All agents have "interests" and "abilities". The **interested-in** relation holds between agents and sentences in the content language, and defines the propositions about which the agent wants to receive notification when they become true. In operation, each **interested-in** statement defined for an agent causes the testbed to issue a corresponding **subscribe** message to the agent's facilitator. The **able** relation holds between agents and *sentences representing KQML performatives*. By means of this relation, developers are able to define the requests that each agent is able to satisfy.

3.2.3.3 Events and event reporting

In an ITS, *events* play a critical role. For example, each student action on the interface corresponds to an event that may be meaningful to an agent in the system. In addition, agents may

generate events that represent important changes in operational state, for example, the **curriculum manager** agent might generate a **topic selected** event when it has completed that phase of its planning operation. Thus the GIA system ontology includes a hierarchy of events, and defines the relations **can-report** and **handles** between agents and events.

The **can-report** relation is used to define the kinds of events that an agent might report during the course of its operation. For each statement of such a relationship, the testbed software causes the agent to issue a corresponding KQML **advertise** message to its facilitator. The **handles** relation defines the kinds of events that a particular agent wants to receive notifications about. For each statement of such a relationship, the testbed software causes the agent to issue a corresponding KQML **subscribe** message to its facilitator. The event hierarchy combined with the **can-report**/**handles** relations and the underlying agent communication infrastructure provides the means for agents in an ITS to notify one another when events of interest occur.

3.2.3.4 Interface agents and instructional resources

Interface agents are distinguished from kernel agents in that they may have a visual presence on the student interface. For example, a particular interface agent might control an interface widget that provides the functions of a simple calculator. The concept of an **instructional resource** is part of the pedagogy ontology; instructional resources like calculators, spreadsheets and simulators are legitimately part of the world of instructional design. But interface agents *manage* instructional resources, and so we have defined the **manages-resource** relation as holding between agents and instructional resources. Since instructional resources are not part of the system ontology, this relation is something of a "bridge" between separate yet related ontologies. (In fact, it is one of several such bridge relations.) In addition, the **available** predicate is defined to be **true** of an instructional resource when that resource is on-screen and accessible to the student.

This concludes our description of the major elements of the GIA system ontology. The terms and relations discussed above capture the key design commitments that use of GIA entails. In the next subsection we present a short annotated example that illustrates how our agent communication framework supports a simple ITS function.

3.3 Example of testbed operation

Logging the student in is one of the first things that must occur during an ITS session. The system needs to know who the student is, so that it can load any pre-existing records of the student's use of the system. In this section, we present an annotated example illustrating how students are logged in by the GIA testbed software. The example depends on a number of definitions in the CKB, built using terms in the system ontology. We present these in the next subsection, then proceed with the operational example.

3.3.1 Supporting definitions

First, we must define the agents. Four agents play a role in this example: the kernel executive, the screen manager, the training administrator, and the **student login agent** (an interface agent).

```
(facilitator kernel-executive)
(facilitator screen-manager)
(kernel-agent training-administrator)
(interface-agent student-login-agent)
```

Next, the following is the portion of the event hierarchy relevant to the example (we use the **isa** relation to express subsumption relationships):

```
(isa event student-connection)
(isa event student-login-event)
(isa event training-administration-event)
(isa student-login-event student-login)
(isa student-login-event student-logout)
(isa training-administration-event login-completed)
```

Events of type **student-connection** (including subtypes of this event, if any) are reported by the kernel executive when it receives and establishes a network connection from a remote screen manager:

```
(can-report kernel-executive student-connection)
```

Events of type **student-login-event** (and its subtypes **student-login** and **student-logout**) are reported by the student login agent:

```
(can-report student-login-agent student-login-event)
```

Events of type **training-administration-event** are reported by the training administrator:

```
(can-report training-administrator training-administration-event)
```

It is important to understand that terms such as **student-connection**, **student-login**, etc. represent *types*, not *instances*. When an agent generates an event, it creates a new constant symbol to represent that event, and adds a one-place predicate of the form (E <const>) to its VKB, where E is the event type. We will have more to say about this in a moment.

The system ontology also defines various relations that may hold between events and other types of things. For example, the **connection-by** relation is defined to hold between entities of type **student-connection** and entities of type **student** (also a term in the system ontology). The kernel executive is able to answer queries about the unique internal identifier assigned to new remote connections:

```
(able kernel-executive (ask-about `(.the. ,?s (connection-by ,?e
,?s))))
```

(The semantics of the backquote operator ` and the comma-escaped variables ?s and ?e are defined in the KIF specification.)

The training administrator needs to be notified when remote students connect to the kernel so that it can initiate the login process:

```
(handles training-administrator student-connection)
```

Recall that in GIA, kernel agents cannot manipulate the interface directly. All interface-related operations are performed by interface agents. The student login agent is designed to present a dialogue box on the interface that prompts the student for his/her name and password. This dialogue box, called the **login-tool**, is defined as an instructional resource, and the student login agent is defined as its manager:

```
(instructional-resource login-tool)
(manages-resource student-login-agent login-tool)
```

When the student fills in the name and password fields on the login tool and clicks OK, the student login agent records a **student-login** event. The training administrator needs to be notified when this occurs:

```
(handles training-administrator student-login-event)
```

The system ontology defines several important relations involving student-login events. The **student-fname**, **student-mi**, **student-lname** and **student-password** relations are defined be-

tween student-login events and strings. These strings are interpreted as representing the student's first name, middle initial, last name and password, respectively.

These are all the definitions necessary to understand the example, which we turn to next.

3.3.2 The testbed in action

As discussed previously in section 3.2.2, when the GIA kernel is started, the Kernel Executive is created and initialized. It, in turn, creates and activates all the agents in its jurisdiction. (Its jurisdiction is defined in the CKB.) Agents, when activated, query the CKB to find out what their interests and abilities are, then proceed to transmit appropriate **subscribe** and **advertise** performatives to their facilitator.

Given the definitions shown in the previous subsection, the Training Administrator will transmit these messages (for simplicity, we will write KQML messages in a shortened form consisting only of the performative and the content):

```
(subscribe `(student-connection ,?e))
(subscribe `(student-login ,?e))
(subscribe `(student-logout ,?e))
```

The Kernel Executive is able to report **student-connection** events, so it will send itself an **advertise** message:

```
(advertise (tell `(student-connection ,?e)))
```

As **advertise** and **subscribe** messages reach the Kernel Executive, it updates its tables of interests and abilities. When it finds a match between a subscription and an advertisement, it sends a **monitor** message to the advertiser to enable event reporting. When the Kernel Executive receives the first **subscribe** above from the Training Administrator, it matches it against its own advertisement, so it sends itself a **monitor**:

```
(monitor `(student-connection ,?e))
```

Imagine that a connection from a remote screen manager is now received by the Kernel Executive. This is a **student-connection** event, so the Executive creates a unique constant symbol, say **!event3**, to refer to it, and places the following expression in its VKB:

```
(student-connection !event3)
```

The Executive also needs to keep track of the students that are active at any given time, so it creates another unique symbol, say **!student 5**, to refer to the new student, and puts the following expression in its VKB to associate the student with the connection event:

```
(connection-by !event3 !student5)
```

After completing the network connection, the Kernel Executive proceeds to transmit the contents of its local table of interests to the remote screen manager. During this process, the remote screen manager receives notification of the Training Administrator's interest in **student-login** events. Because the **student-login-agent** is defined as generating such events, it will have sent a corresponding **advertise** message to its facilitator (the Screen Manager). Thus the Screen Manager will be able to match the **advertise** and **subscribe** and will issue a **monitor** message to the **student-login-agent.**

On the kernel side, as a consequence of the earlier **monitor**, the Executive automatically issues a **tell** message to the Training Administrator:

```
(tell (student-connection !event3))
```

When the Training Administrator receives notification of **student-connection** events, it needs to initiate the login sequence. That is, it needs to obtain the student's name and password.

Since it cannot access the interface directly, it needs to issue a request that will have the desired effect. First, the Training Administrator needs to query the Kernel Executive for the identifier of the student associated with the reported event, so it issues an **ask-about** message to its facilitator:

```
(ask-about (.the. ?i (connection-by !event3 ?i)))
```

The Kernel Executive replies to this message as follows:

```
(reply !student5)
```

At this point, the Training Administrator can now issue a request to display the login prompt on !student5's display. In order to ensure that the message is routed to the correct remote screen manager, the Training Administrator specifies !student5 in the **:context** field of the KQML message:

```
(achieve :context !student5 :content (available login-tool))
```

The Kernel Executive transmits this message to the remote screen manager over the network link associated with !student5. At the interface side, the Screen Manager uses content-based routing to determine that the **student-login-agent** is able to achieve the above goal because it manages the **login-tool** resource. It thus passes the request on to that agent.

Interface agents respond to requests to achieve **(available <resource>)** by presenting the resource on the display. The **login-tool** resource consists of a dialogue box containing fields for the student's first name, middle initial, last name and password, with a button to click on when all fields have been filled in.

When the student fills in all the fields and clicks the acceptance button, the **student-login-agent** records a **student-login** event and places associated data in its VKB as follows:

```
(student-login !event19)
(student-fname !event19 "brant")
(student-mi !event19 "a")
(student-lname !event19 "cheikes")
(student-password !event19 "secret")
```

At this time, the earlier **monitor** received by the **student-login-agent** will cause the agent to issue a **tell** to the screen manager:

```
(tell (student-login !event19))
```

The screen manager will pass this message on to the remote facilitator (the Kernel Executive), which will in turn route it to the Training Administrator. At this point, the Training Administrator could use the **!event19** constant as a "handle" and use the **ask-about** KQML performative to issue requests for the student's name and password that are associated with the event identifier in the **student-login-agent**'s VKB. From there the Training Administrator could proceed to authenticate the student's identity.

3.3.3 Summary

This example shows how agents in the GIA testbed can use KQML messages to exchange information and perform a basic ITS function, that of obtaining the student's name and password for the purpose of logging him/her in. All terms and relations used in the content expressions are defined in the GIA system ontology. We do not claim that ours is the only proper way to use the agent communication framework to prompt for the student's name and password; it is, however, an approach that we have "formalized" by encoding its supporting terms and relations in the system ontology. This encoding constitutes a form of publication; it is incumbent

upon developers to implement their agents accordingly. If that is done, it should be easy to share and reuse both the ontology and the agent software.

4. Conclusions

Research on ITS technology is currently being hindered by long, costly development cycles. One reason for the length and cost of these cycles is that ITSs are not designed in modular ways that promote sharing and reuse of software components. This situation has at least two undesirable consequences for the field as a whole. First, because building prototypes of reasonable scale and robustness is so costly, it is rarely done, and so meaningful system evaluations are uncommon. Second, until ITSs can be developed and fielded more cheaply and more quickly than the subject matter changes, the technology will be perceived as infeasible and perhaps not worth sustained research funding.

4.1 Open problems

In this paper we have described an approach, based on the agent-communication framework defined by the ARPA Knowledge Sharing Effort, that we believe can help reduce the cost and time required to develop and deploy ITSs. This work is in its early stages, so we cannot yet quantify the extent to which ITS development cycles might be shortened.

We believe our approach is promising for at least three reasons. First, it encourages developers to design and build ITSs out of modular components that interact in formally-specified ways. This has not generally been considered important, but is the first step towards producing reusable software. One obvious way to reduce ITS development costs is to reuse parts of old ITSs when building new ITSs. This will not be possible unless developers abandon the "made to order" mentality and instead design their prototypes with possible reuse in mind.

Second, the agent communication language provides a standard syntax and semantics for information exchange between ITS components. Widespread adoption and use of this language can foster *cross-organizational sharing*, i.e., sharing of software between development teams at different institutions.

Third, ontology engineering is fundamental to the specification of agent communication and interaction. It promotes a design process in which developers formalize the entities, relations and functions needed to describe the system, the domain, the instructional process and the student's knowledge. The ontologies produced as a result of this process can be shared and reused, and the design decisions that they express or imply can be evaluated and compared. Due to the idiosyncratic nature of ITS development today, it is usually difficult or impossible to compare results produced by different research groups.

While promising, the approach we have outlined raises some problems of its own. First, we lack a development methodology that will assure that the services needed by any given agent will be provided by one or more other agents. Second, it is not clear what ought to happen if more than one agent can respond to a request. Third, in any reasonably expressive language, it may be possible to achieve a given communicative goal in a number of ways. What effect will this have on sharing and reuse? Fourth, expressive communication languages can lead to ambiguity of expression. How can we prevent this from hindering sharing and reuse?

4.2 Current status

Addressing these open problems with take time and practical experience. We have been implementing a testbed system intended to help us assess the strengths and weaknesses of the agent-based approach to ITS development. Our system is written in the Smalltalk language,

using ParcPlace Systems' VisualWorks 2.0 programming environment. We have found the object-oriented nature of Smalltalk to be particularly well-suited to the development of agent systems. So far we have implemented the basic architecture shown in Figures 1 and 2, with enough support for KQML message passing and content-based routing to run the example described in section 3.3. Our ontologies are encoded using the CODE4 knowledge management tool, developed at the University of Ottawa by Doug Skuce and Tim Lethbridge. We are currently in the process of adding features to our testbed and plan to use it as the foundation for one of our internal ITS prototyping projects.

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.

Cohen, P. R. and Levesque, H. J. (1995) Communicative actions for artificial agents. In *Proceedings of the International Conference on Multiagent Systems*, San Francisco, June.

Finin, T., Weber, J., Wiederhold, G., Genesereth, M., Fritzson, R., McKay, D., McGuire, J., Pelavin, R., Shapiro, S., and Beck, C. (1992) *Specification of the KQML Agent-Communication Language*. Technical Report EIT TR 92–04, Enterprise Integration Technologies, Palo Alto.

Genesereth, M. S. (1993) *An agent-based approach to software interoperability*. Technical Report Logic–91–6, Logic Group, Computer Science Department, Stanford University, February.

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.

Gruber, T. R. (1993) *A Translation Approach to Portable Ontology Specifications*. Technical Report KSL–92–71, Knowledge Systems Laboratory, Stanford University, April. To appear in *Knowledge Acquisition*, 1993.

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.

Industry Report (1994) An overview of employee training in America. *Training*, October. pp. 29–37.

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.

Lentini, M., Nardi, D., and Simonetta, A. (1995) Automatic generation of tutors for spreadsheet applications. In *Proceedings of AI-ED 95—World Conference on Artificial Intelligence in Education*, Washington, D.C., August 16–19, pp. 59–66.

Ritter, S. and Koedinger, K. R. (1995) Towards lightweight tutoring agents. In *Proceedings of AI-ED 95—World Conference on Artificial Intelligence in Education*, Washington, D.C., August 16–19, pp. 91–98.

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