Engineering Intelligent Tutoring Systems

Kimberly C. Warren Bradley A. Goodman
Artificial Intelligence Center[1]
The MITRE Corporation
202 Burlington Road
Bedford, MA 01730-1420
kim@linus.mitre.org
bgoodman@linus.mitre.org

ABSTRACT. We have defined an object-oriented software architecture for Intelligent Tutoring Systems (ITSs)[2] to facilitate the rapid development, testing, and fielding of ITSs. This software architecture partitions the functionality of the ITS into a collection of software components with well-defined interfaces and execution concept. The architecture was designed to isolate advanced technology components, partition domain dependencies, take advantage of the increased availability of commercial software packages, and reduce the risks involved in acquiring ITSs. A key component of the architecture, the *Executive*, is a publish and subscribe message handling component that coordinates all communication between ITS components.

We implemented critical components of the architecture as a simple hypermedia training system, the Macintosh Maintenance Training System (MMTS). The domain for the prototype training system is the maintenance of Apple Macintosh IIcx computers.

This project has shown that the use of a modular software architecture for the development of ITSs, and complex integrated artificial intelligence applications in general, has several important benefits. Its use allows for rapid development, incremental integration and testing of components, and a more maintainable, extensible, and reusable end-product. Even more evident was the benefit of an Executive component that facilitated the integration of commercial software packages with custom developed software in a well-defined manner.

INTRODUCTION

This paper describes the design and implementation of a generic architecture for Intelligent Tutoring Systems (ITS). The architecture uses object-oriented technology to provide the ITS developer with a solid, reusable framework for the development and testing of particular functional modules of an ITS and the ability to easily integrate COTS products with custom-developed software.

Prior to the selection and use of Commercial Off-the-Shelf (COTS) products, custom-developed functionality, and non-developmental software, software system designers benefit from a systematic definition of functions and subsequent interfaces that will be required of them, i.e., the definition of a software architecture. In the case of an ITS to be run on a single CPU with media-providing peripherals, a software architecture partitions the functionality of the ITS into a series of software components with well-defined interfaces and execution concept. The development of a robust software architecture can improve the overall software quality of a system.

1 This work was supported by the Computer-Based Training Specialty group as a MITRE technical overhead project.

2 © The MITRE Corporation, 1993

BACKGROUND

We identified the need for a reusable and extensible software architecture to be used across multiple ITS for various reasons. For example, a considerable savings in the development cost of multiple tutors will result if common software components are identified, developed once or assigned to an existing COTS or non-developmental software (NDS) product, and incorporated into all tutors. Large scale software reuse also means that the tutors can have a similar "feel" to the student and would be less expensive to maintain. The use of appropriate COTS products also provides the users/developers with emerging COTS interface look and feel standards. An extensible architecture attempts to achieve goals including making it possible to incorporate changes in technology, in such areas as multimedia interfaces, student modeling, expert modeling, and instructional strategy, without having to re-implement the entire system. We felt that building tutors on a generic framework of loosely-coupled software components had the best chance of achieving these goals. We developed such an architecture and specified it in terms of functional components, detailing the function, interfaces, and suggested behavior of each [1].

Other ITS researchers have defined what they call an "architecture" for an ITS to address issues of easing the cost associated with developing ITS. Most of this work relies on their particular definition of the term "architecture." Work done by John Anderson (with Boyle and Yost [3], Reiser and Farrell [4], and Corbett [5]) on the Geometry, Lisp and ACT tutors is likely the most advanced application of a reusable architecture to date. However, as Yazdani observed (in [6]), these tutors suffer from limited student modeling and instructional design strategies. These strategies are prescribed (even required) by the "architecture" and do not allow for modifications that would allow them to better deal with each individual domain. Also, none of these implementations used COTS, they relied upon custom developed software.

NASA/Johnson Space Center has followed a different path by making an attempt at developing a "generic architecture" for ITS in the form of a set of integrated tools [7]. Their general-purpose ITS design uses a COTS knowledge engineering tool, CLIPS, for encoding and manipulating production rules and a blackboard system component, custom-developed, for coordinating communication between modules. The developer of an ITS can build on top of the kernel provided by the NASA general architecture to reduce development time and to provide some consistency across ITS. However, their selection of specific tools constrain the ITS designers/developers to those methodologies supported by those tools. For example, the particular COTS knowledge engineering tool that they selected requires knowledge to be encoded in terms of production rules. Additional design and architecture changes would be required if the ITS designer required an alternative formalism, and/or alternative COTS tool, for the representation of expert, student, or instructional knowledge. Also, most of the ITS must be re-built for each new domain.

In contrast to other ITS architecture efforts, we proposed that general interfaces to the required functionality of an ITS could be specified and implemented. Components of an ITS such as an expert model, student model, and instructional environment could be specified in terms of their required functionality and that, indeed, when knowledge representation and implementation details are set aside, standard interfaces to the specified functionality emerge. Such encapsulation of functionality allows particular components to be "unplugged" and replaced by alternative components conforming to the same interfaces. This proves to be a very powerful capability in the development of ITS.

Although knowledge representation schema differ widely in ITS research into expert modeling, student modeling, and instructional design, the conditions under which such functional components are activated, i.e., their input, and the responses that are expected from them, i.e., their output remain similar across knowledge representation and manipulation methodologies. For example, expert modeling functionality receives input that includes student action/statements and will generate output that includes expert evaluations of those actions as part of its function as the expert problem solver in an ITS. In the case of student modeling, "higher" level measurement of the student's ability (e.g., "higher" than a simple recounting of student actions during problem solving) is expected to be available from the student modeling functional component during the lesson. For example, student modeling functionality usually receives student action/statements as input and generates some measurement of overall student ability as output. Knowledge representation, manipulation, and presentation differ widely in ITS research into instructional environments. However, the conditions under which an Instructional Environment operates and the responses that are expected from it, e.g., student action/statements and individualized responses/feedback to the student, respectively, remain similar across methodologies. These similarities allow general interfaces to these functional components in an ITS to be specified [2]. With these general interfaces, we have opened the door to the use of appropriate COTS products to implement the specified functionality.

APPROACH

The architecture of a digital system consists of the hardware and software components, their interfaces, and the execution control that underlies system processing [8]. A software architecture should partition the functionality of ITS into a collection of software components with well-defined interfaces and execution concept. A software architecture for an ITS should be designed (i.e., ITS functionality should be partitioned) to meet the following goals:

a. Technology insertion and isolation: The architecture should provide logical functional separation to allow the insertion of relevant new technologies as they become available and provide a way of isolating dependencies on existing technologies.

b. Ease of acquisition: The architecture should reduce the overall development and maintenance costs associated with designing and building multiple ITS.

c. Domain dependency isolation: The architecture should isolate domain dependencies of the ITS. While this isolation will not be complete, it should at least define and localize the effects of any domain changes on the remainder of the ITS.

d. Isolation of instructional design strategies: The architecture should allow differing instructional strategies to be implemented and tested, with minimal impact on the rest of the ITS.

In order to achieve the above goals, we specified and began implementing a generic ITS architecture in terms of an object-oriented design for the functional components of an ITS. The highest level objects in the ITS architecture correspond to the software components that support the individualized training environment. Each of these components is assumed to be an independent entity that can act on its own accord. As an object-oriented system, the software components communicate with each other by sending and receiving messages that either contain control information that affect components' operation (e.g., starting or stopping a simulation) or queries for data about some aspect of the course (e.g., state of a simulated system component.)

In one instantiation of our architecture, each ITS can consist of the ten software components shown in figure 1. (A *software component* is defined to be an independent functional entity that performs a logical set of functions. Each software component communicates with other software components via well-defined interfaces. One can come up with many different partitions of the functionality found in an ITS as the basis for an architecture definition; however, the resulting architecture should follow the guidelines and goals stated above for modularity of advanced technological (i.e., Artificially Intelligent), instructional strategic, COTS integration, and domain-dependent components. If this is the case, it should adapt to changes in implementation and domain gracefully, i.e., without requiring the re-coding of all components.)

The Executive is the core of the ITS architecture. It is the software component that implements the execution control that underlies system processing and fosters ease of COTS integration. The Executive provides one of the basic functionalities that is characteristic of object-oriented architectures, message passing between encapsulated processing entities.

The functionality of an ITS student interface, i.e., process student actions in order to generate individualized feedback, has been refined into multiple components in our architecture: Multimedia Interface, Presentation Controller, Explanation Generator, and Instructional Environment. This refinement was meant to leave the Instructional Environment to deal with high level instructional design strategy processing. With the availability of COTS tools for multimedia graphical user interface development and delivery, we isolated the Multimedia Interface to allow for the use of COTS. The Multimedia Interface provides the physical means by which the student interacts with the Instructional Environment. Specifically, the Multimedia Interface processes multimodal input from the student, via such devices as keyboard and mouse, and generates and displays multimedia presentations including text, graphics, sound, and video. Input in terms of input events (mouse clicks, key input) is passed on to the Presentation Controller for further refinement. Output requests in terms that the particular multimedia tools can interpret are generated by the Presentation Controller.

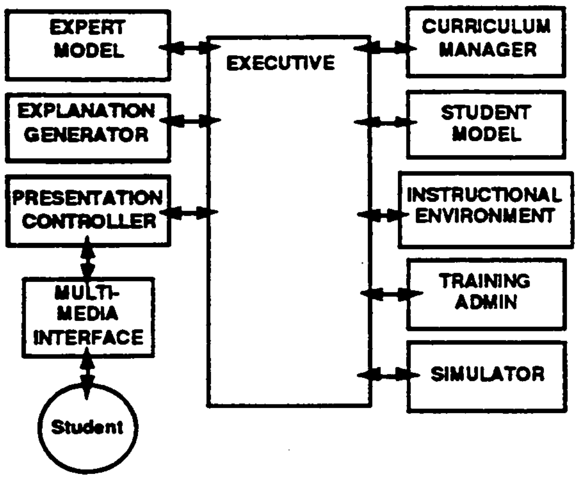


Figure 1 Components of the ITS Architecture

Depending upon the tools used to implement the Multimedia Interface, a particular "language" will be required to control presentation of information. The Presentation Controller organizes and sequences the information to be presented to the student. Given an Explanation Generator script containing several display directives such as "AT THE SAME TIME: Display video indices 1230 to 1259 and text 'This is how you ...' ," the Presentation Controller synchronizes the execution of requests for their physical display by the Multimedia Interface. The Presentation Controller is also responsible for the translation of the physical user interactions (e.g., mouse movement, keyboard input, etc.) supplied by the Multimedia Interface into actions on the problem domain elements for the Instructional Environment.

Considerable research and development is ongoing in the area of tailored explanations and multimedia explanation generation (see [9,10] for examples). For this reason, we isolated the functionality that generates the content and sequence of explanations given to the student in the Explanation Generation component. The Explanation Generator, in conjunction with the Instructional Environment, is the part of the training system responsible for generating individualized advice. The output of the Explanation Generator is a script denoting information to be displayed in terms of media items organized in terms of temporal and spatial relationships.

While the Instructional Environment and its related components control the presentation of material to the student within a given exercise, the Curriculum Manager governs problem selection with respect to the entire course or group of exercises. Like the other modules, the Curriculum Manager uses information about the student's current problem solving skills and knowledge to generate appropriate interactions with the student. This information is compiled and maintained by the Student Model. The Student Model gathers data from the student's problem solving actions. It should map these actions into a representation that summarizes and abstracts relevant student performance measurements.

A Simulator component was broken out of the standard ITS Expert Model component. This isolates as much of the domain dependencies associated with a tutor's domain simulation and allows for the use of COTS simulation and expert modeling tools. The Expert Model contains knowledge that includes the general methods or procedures that an expert would use to solve problems in the domain. It is possible that the expert problem solving knowledge used by one ITS, or the COTS tools used to implement it, could fairly easily carry over to additional domains. This is not the case for tutor domain simulations.

Finally, the Training Administrator collects and stores measurements of ITS usage and students' problem solving skills. The maintainers of the software may use standard COTS analysis tools, such as spreadsheets, to evaluate the usage information.

THE MACINTOSH MAINTENANCE TRAINING SYSTEM

We implemented critical components of the architecture as a simple training system, the Macintosh Maintenance Training System (MMTS). The domain for the prototype training system is the maintenance of Apple Macintosh[3] IIcx computers. The initial capability for the implementation described in this paper came from a multimedia explanation generation component for an intelligent help system developed by Bradley Goodman [11]. We encapsulated this system's functionality into separate software components in accordance with our architecture. We also augmented this system with student modeling capabilities that were lacking in the original Macintosh repair system. Our development approach made use of commercially-available software products and object-oriented technology. The implementation used the DOS/Microsoft Windows operating system running on an IBM PC-compatible hardware platform. We used a combination of C, C++ and Asymetrix Toolbook to develop our prototype.

The MMTS prototype is made up of four software components that provide the initial, critical functionality required by an ITS for Macintosh maintenance: Executive, Instructional Environment, Expert Model, and Student Model. The MMTS uses separate executables for the software components (implemented using different languages and COTS products) that make up the tutoring system. The following paragraphs describe the MMTS design and implementation in some detail.

The Executive has responsibility for the dynamic coordination of all message passing between software components in the object-oriented architecture. A *message* is an object that has attributes (fields) that may include: message class, contents, priority, sender, and time stamp.

In keeping with the strategy to use object-oriented techniques to maintain a loose coupling of components in the ITS, the Executive employs a subscription-based message handling approach wherein software components request message handler permission to publish messages on selected message classes. A message class is, for example, "student action" which specifies messages containing descriptions of student actions during problem solving with the tutor. Conversely, a software component can subscribe to the various message classes from which it wants to receive information.

Software components publish and subscribe to messages via I/O ports created for one or more message classes. The use of I/O ports for the sending and receiving of messages allow software components to easily suspend multiple message classes with a single call to suspend a port. Messages sent by a software component through an I/O port for publication are broadcast to all software components that subscribe to the message class. Message sending can be accomplished in any of the following ways:[4] asynchronous, synchronous, or synchronously timed. The flexibility allowed by these types of sends allows the ITS developer to control processing at many different levels. For example, if a particular component of the system wants to interrupt another component's processing, an asynchronous send to the relevant component(s) would be required. Message receipt can be accomplished in any one of the following ways:[5] unconditionally with wait, typed, timed, or typed and timed. These types of receipt allow the various components of the system to selectively process requests from other components.

3 Apple and Macintosh are trademarks of Apple Computer, Inc.

4 Currently, we are only handling the asynchronous passing of messages between processes under Microsoft Windows. We believe that, of the three types, this is the most complex to implement given a focus on incorporating COTS software into any implementation of the architecture.

5 Currently, we are only handling the unconditional receipt of messages from within the Executive.

The Executive was implemented in C as a Dynamic Link Library[6]. It functions as a layer of message publication and subscription handling on top of Microsoft Windows. This layer maintains tables of software components, messages that they subscribe to, and messages that they will publish. Software components are provided with the various send and receive capabilities described above, with the Executive handling all of the Microsoft Windows communication issues. Software components that wish to communicate need only link with the Executive and publish on and subscribe to appropriate messages. The availability of the Executive defining a standard interface between COTS and custom developed products simplified their integration to one of assuring communication with the Executive. Systems that employ multiple COTS products communicating in complex conversations require the developer to implement multiple interfaces between COTS and custom developed software, one per communicating pair of system components. When the Executive is used, however, the interface implementation is reduced to the implementation of one interface to the Executive for each COTS and custom developed component. Finally, the use of Windows allowed us to take advantage of non-preemptive multi-tasking to control the order in which messages were both sent to and received from applications linked to the Executive.

The graphical user interface of the MMTS, the Instructional Environment, is shown in figure 2. This interface was built using Multimedia Toolbook[7] . The scrolling menus on the left enumerate the actions that the student can take. The buttons down the middle of the screen provide the student with access to any of the three views of the Macintosh IIcx. The buttons on the right provide the student with textual, graphical, and video help during problem solving. The scrolling window at the bottom of the screen shows the plan that the student is creating during problem solving and allows him/her to select any of the actions in the plan for video display. This particular display shows the results of appropriate student actions in the MMTS as the listing of the English text in the "Actions to be Performed:" window.

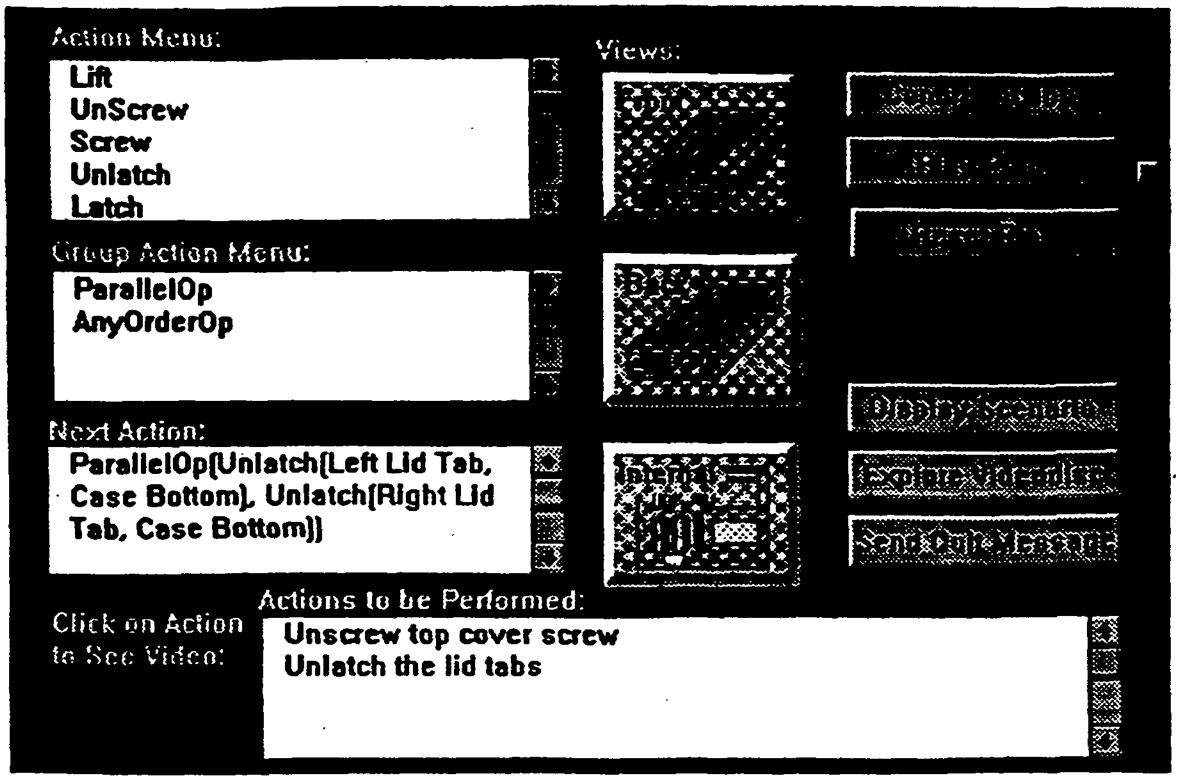
An Expert Model, written in C++, models the procedures required to do Macintosh repairs. These procedures were implemented as a finite state machine that encodes the rules for repair delineated in the Macintosh Maintenance Manuals. During the course of problem solving, the Expert Model evaluates student problem solving actions and responds to queries for appropriate (expert) actions in the context of the problem being solved.

A Student Model, written in C, models the student's problem solving by tracking the errors and their gravity during a problem solving session. During the course of problem solving, the Student Model may require the Instructional Environment to display additional, unsolicited advice if the students errors are of a type or frequency that the Student Model would find inappropriate.

The Instructional Environment coordinates instructional interaction with the student. For example, the Instructional Environment *publishes on* the "student action" class of messages. Conversely, the Expert Model and Student Model components *subscribe to* the "student action" class of messages. When a student completes a maintenance action on the Macintosh IIcx (e.g., clicking on the video board graphic and the menu action "pull up") via the Instructional Environment, the action is packaged into an instance of the "student action" message class and broadcast to all components that subscribe to the class via the Executive message handler. In this case, the Executive will route the message to both the Expert Model and the Student Model for evaluation and retention, respectively.

6 A DLL allows applications to access functions in a library without having to include the library code in the source code of the application. This also allows multiple applications to access a single copy of the library.

7 Toolbook is a registered trademark of the Asymetrix Corporation.



Next Action:
ParallelOp(Unlatch(Left Lid Tab, Case Bottom), Unlatch(Right Lid Tab, Case Bottom))

Click on Action to See Video:

Actions to be Performed:
Unscrew top cover screw
Unlatch the lid tabs

Figure 2 MMTS user interface

The Expert Model will evaluate the action with respect to an expert's problem solution. This component will then create and broadcast a message of class "student action evaluation" containing the evaluation which will be received by both the Instructional Environment and the Student Model. The Instructional Environment will process the evaluation with respect to its instructional design strategy (e.g., whether to give immediate feedback, the type of feedback, etc.) For example, the Instructional Environment may cause the creation of a multimedia display of a video board being removed with text overlayed describing the action.

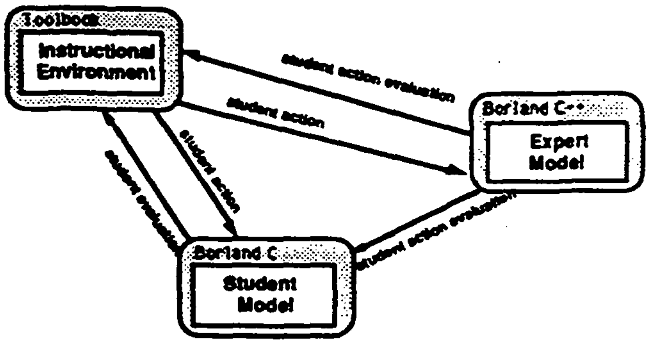


Figure 3 Sample Message Passing

The Student Model will store the action, as well as any Expert Model evaluation, and make any inferences about the state of the student's understanding that are possible. If it is the case that the student's understanding has dropped below a particular threshold, the Student Model will broadcast a "student evaluation" message which will be picked up by the Instructional Environment. At this time, the Instructional Environment may provide the student with unsolicited advice, depending upon the instructional strategy implemented. Figure 3 shows the message passing between software components that the Executive facilitates in this particular example.

CONCLUSIONS

The work done on this project to date provides evidence that the use of a modular software architecture for the development of ITS has several important benefits. Its use allowed for rapid development, incremental integration and testing of components, ease of COTS integration, and a more maintainable, extensible, and reusable end-product. In particular, the Executive component facilitated the integration of both COTS and custom developed software in a well-defined manner. Standard integration tasks require the specification and development of multiple interfaces, one per communicating pair of components. Software components, be they COTS or custom developed, that communicate via the Executive message handler require only a single interface; that is, they need only be integrated with the Executive. Acting as an abstraction of the message passing/process communication functionality provided by the operating system, the Executive is able to take quite a bit of the integration burden away from the developers of the system.

The use of COTS products allowed us to rapidly develop an initial training capability. The freedom to choose multiple COTS tools in combination with more standard programming languages gave us the ability to implement fairly complex algorithms in a short period of time. Many problems arise from the use of COTS products (alone) in a rapid prototyping situation due to the limitations of any one particular COTS product. Other problems arise from the range of different integration tasks that developers are required to perform when combining the functionality of multiple COTS products. The availability of the Executive defining a standard interface between COTS and non-COTS products simplified the integration to one of assuring communication with the Executive.

With the savings in development afforded by the use of COTS, the increased supply of powerful COTS software, and the increased demand for its use in government and private sector development efforts, issues of maintainability of software systems that utilize these products are brought to the forefront. Prior to the selection and use of COTS products, system designers benefit from a systematic definition of functions and subsequent interfaces that will be required of them. The architecture specified for ITS in this paper was designed to take advantage of the increased availability of COTS software. The partitioning of components took into account COTS trends. For example, the separation of the Multimedia Interface from the Instructional Environment allows developers/designers to take advantage of the multitude of available COTS drivers for graphical user interface providing hardware. The separation of the simulation functionality from the Expert Model provides one with the ability to either develop a simulation given a COTS tool or the use of an existing simulation of the training domain.

The Executive emerged from this activity as a tool for the integration of software components for any application, not limited to ITS, and a software engineering methodology enforcement mechanism to be used during design and development of a system. The use of the Executive enforces "good" software engineering practices by requiring all designers/developers to conform to a standard set of interfaces defined in advance of full-scale development. It was clearly efficient to allow each of the developers to ignore the implementation details of components for which he/she was not responsible. With the Executive, the interface and communication issues must be addressed and resolved "up-front".

The generic ITS architecture and its Executive are not magic. As a matter of fact, the hard ITS design and development work is yet to be done. This work includes the advanced technology design and development issues involved in creating the various "intelligent" modules of the training system. However, with the use of such an architecture for the implementation of ITS, and integrated COTS systems in general, we feel that the users and developers will benefit. The addition of techniques, products, and technologies has been anticipated. The re-use of complex components has been enabled.

Acknowledgments

The author would like to acknowledge Glenn Karcher for his extensive contributions to the design and development of the Executive functionality, George Huff and Steve Litvintchouk for their extensive Software Engineering advice, and Michael Sayko for his contributions to the design of the Intelligent Tutoring System architecture.

References

[1] Warren, K. C., Karcher, G. W., Sayko, M. S., Goodman, B. A., "The ITS Software Engineering Reference Guide," (draft) MITRE Working Paper, WP-92B0000085, 1992a.

[2] Warren, K. C., Huff, G. A., Michlowitz, E. M., "A Generic Architecture for Intelligent Tutoring Systems," MITRE Technical Paper, MTR 92B0000200, 1992b.

[3] Anderson, J., Boyle, C. F., and Yost, G., "The Geometry Tutor," in *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, 1985

[4] Reiser, B. J., Anderson, J. R., and Farrell, R. G., "Dynamic Student Modeling in an Intelligent Tutor for Lisp Programming", in *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, 1985

[5] Corbett, A. T., and Anderson, J. R., "Student Modeling and Mastery Learning in a Computer-Based Programming Tutor" in *Intelligent Tutoring Systems, Second International Conference, ITS '92*, Springer-Verlag, 1992.

[6] Yazdani, M., " Intelligent Tutoring Systems: An Overview," in *Intelligence and Education, Volume One*, eds. R. W. Lawler and M. Yazdani, Ablex Publishing, 1987.

[7] Computer Sciences Corporation, "Programmer's Guide to the General Architecture of Intelligent Computer-Aided Training Systems," Report CSC-A000154, December 1991.

[8] Horowitz, B. M., MITRE, "The Importance of Architecture in DOD Software," MITRE Technical Paper, M91-35, July 1991.

[9] Feiner, S. K. and McKeown, K. R., "Generating Coordinated Multimedia Explanations" in *Proceedings of the 6th International Conference on Artificial Intelligence Applications*, Santa Barbara, CA, 1990.

[10, 11] Goodman, B. A., "Multimedia Explanations for Intelligent Training Systems," presented at Conference on Intelligent Computer-Aided Training, Houston, TX, 1991.