An Object-Oriented Architecture for Intelligent Tutoring Systems

Jeffrey Bonar, Robert Cunningham, and Jamie Schultz

Intelligent Tutoring Systems
Learning Research and Development Center
University of Pittsburgh
Pittsburgh, Pennsylvania 15260

Abstract

We describe an object-oriented architecture for intelligent tutoring systems. The architecture is oriented around objects that represent the various knowledge elements that are to be taught by the tutor. Each of these knowledge elements, called *bites*, inherits both a knowledge organization describing the kind of knowledge represented and tutoring components that provide the functionality to accomplish standard tutoring tasks like diagnosis, student modeling, and task selection. We illustrate the approach with several tutors implemented in our lab.

1. Introduction

We are developing a general intelligent tutoring system (ITS) shell using the object-oriented programming language LOOPS (see Stefik and Bobrow [1986] for a description of LOOPS). Called the *Bite-Sized Tutor*, it provides the curriculum independent part of an intelligent tutor and specifies an organization for the curriculum knowledge to be supplied by a domain expert. Our goal is an interface where the curriculum could be supplied by a domain expert who is not a programmer.

The Bite-Sized Tutor exploits the expert system approach currently being applied in many ITS projects. First a domain is analyzed and novices are observed while learning that domain. The results from those studies supply a "cognitive task analysis" of the domain and a "bug catalog" of common novice problems in the domain. ("Cognitive task analysis" is a phrase coined at LRDC. It implies an analysis beyond a behavioral "rational task analysis" and specifically includes attention to the underlying cognitive skills and representations involved in a performance). From this base, various knowledge engineering techniques are used to construct an ITS. Although, many projects are extending these ideas to build tutors based on more detailed theories of human learning and inference of cognitive states (diagnosis) (see, for example, Bonar and Cunningham [1986], Ohlsson and Langley [1984] and Van Lehn [1984]), there is an enormous potential for tutors based on task analyses and bug catalogs. In particular, such tutors can exploit the wealth of cognitive science research on performance in a variety of domains.

Widespread implementation of tutors based on task analyses and bug catalogs will require an ITS architecture and supporting development tools. In this paper we describe our first steps toward such an architecture.

We first discuss problems with current ITS architectures and an overview or our approach. We also discuss the rationale for using the object-oriented programming paradigm. The architecture itself is then detailed, focusing on structure and flow of control. We illustrate this architecture with several example from our ITS projects. We conclude with future research directions.

2. Rationale

Most current ITS implementations are complex and unwieldy. Similar information repeats in several places and information that ought to be closely related is spread apart. Furthermore, ITS components (e.g. student model, diagnoser), whose roles can be quite clearly delineated in an abstract description of the system, end up implemented with code diffused through many parts of the system. Overall, the systems are not modular. In particular, they do not allow for addition of new domain knowledge or new approaches to the pedagogical tasks. We want to emphasize that this discussion is not a criticism of any particular ITS implementation, but a general problem that appears in many ITS.

It might appear that these difficulties are simply a matter of prototype implementations, written without concern for detailed software engineering issues. While this is part of the problem, there is a more fundamental design flaw, related to the use of knowledge within the ITS. ITS are usually conceived as a series of semi-independent components like "explainer", "diagnoser", "tutor", and "user modeler". The problem is that each of these components need to share many diverse pieces of knowledge. The knowledge needed for different components is at least overlapping, and often closely related.

The WEST tutor [Brown and Burton, 1982] provides an example of these problems. As one of the most intellectually important "classic" ITS, it serves as a useful foil for this discussion. It can be viewed in two ways: by its "issues" (the fundamental items for which the system is prepared to instruct the student) and by its components (e.g. "expert", "differential modeler", "tutorial selector", etc.). In the actual Interlisp-D implementation of the tutor, the program is organized by components. This results in a system with unnecessary duplication and complexity in multiple, overlapping representation of issue knowledge. Besides obscuring knowledge organization, the current implementation of WEST makes it difficult to reuse and extend parts of the tutor. Given the many open research issues for ITS, this is a serious problem.

In general, we need a tool that enables the development of tutoring systems much more rapidly than now possible. We also need a tool that allows a subject domain expert or a teacher (who is not necessarily a programmer) to modify the tutor-student interaction and the domain knowledge without reimplementing the system at each step. Finally we need a tool to make it easier for those developing tutors to test their systems as they are designed.

3. An Object-Oriented ITS Architecture

We propose an architecture where every different kind of thing that the system can understand and talk to the user about (such things are often refered to as "issues", from the usage in the WEST tutor) is represented by a class (in the object-oriented programming sense) in the system. Everything the system knows is stored in a class. Of the different classes representing the domain, many will share common substructure. For such classes, the standard inheritance mechanisms of object-oriented programming are appropriately used. The critical point is that every thing the system will interact with the user about is a separate class. We call these domain knowledge classes Bites. They are all subclasses of the class *Bite*.

Given that we organize the system based on the things that the system knows about, where are we to put components of the tutor like the "diagnoser", "student model", and "task selector"? We provide these components in a generic form as high level classes. So, for example, there are classes that contain the functionality to implement a component like a diagnoser. In this case, the class *Diagnoser* will specify the local data needed to perform the diagnosis function as instance variables and algorithms to use that data as methods. The *Diagnoser* class specification does not specify any particular diagnosis to be done, only the general procedure and data required for doing a diagnosis.

The specific data needed for performing an actual diagnosis are provided when the general component classes (e.g. the *Diagnoser* class just discussed) are inherited by the *Bite* classes that actually need to use them. Similarly, the other standard ITS components are implemented as classes and inherited by the *Bites*. Consider an example where there were two kinds of diagnosers to accomplish two styles of diagnosis. This would be handled by having the general properties of diagnosers in a class *Diagnoser* with the specific properties contained in two subclasses *DiagnoserA* and *DiagnoserB*. *Bites* are specified to inherit their diagnosis capability from *DiagnoserA* or *DiagnoserB* as appropriate.

The proposed architecture solves the problems described in Section 2 by making the system highly modular. Each curriculum element is represented explicitly as a class. To the extent that curriculum elements share structure, that sharing is explicitly represented in the inheritance among the classes representing these elements. Similarly, each of the key tutoring components is represented as a class object. These component classes are used to provide tutoring functionality to the domain classes. Like the domain element classes, component classes use inheritance to represent shared structure.

3.1 The Curriculum Elements: Bites

The structure of the classes representing curriculum element bites is defined by inheritance from two kinds of classes. Tutoring component classes, such as the student model and the diagnoser, provide a framework in which data must be supplied by the implementer or curriculum designer. We plan to build a non-programming interface to facilitate defining these bites. Bites also inherit structure based on the kind of knowledge they represent. We have defined several classes of bites: Abstraction Hierarchy Bites, Definition Bites, I/O Bites, and Discovery Bites. In this section we discuss each in detail.

An abstraction hierarchy represents an ordering of concepts in the curriculum. In this hierarchy specific versions of a concept appear at the lowest level of the hierarchy and more abstract versions of that concept appear higher in the hierarchy. An example of this is shown in Figure 1. There we see the abstraction hierarchies for Ohm's Law and Kirchoff's Law from our electricity tutor. The two highlighted nodes show

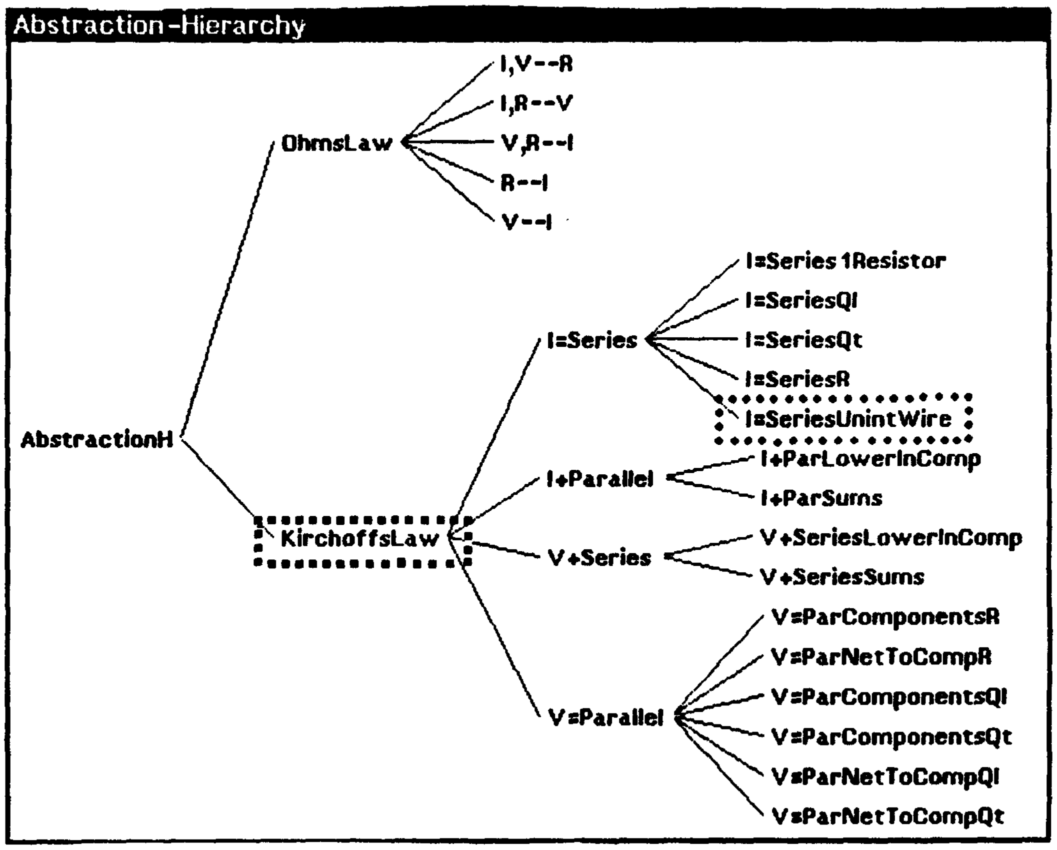


Figure 1. Abstraction Hierarchy from the Electricity Tutor

the relationship between the specific concept, "current is unchanged across an uninterupted wire", and the more abstract concept, "Kirchoff's Law". The I=SeriesUnintWire bite is a specific version of KirchoffsLaw bite and thus is shown at a lower position in the hierarchy.

Abstraction hierarchy bites play an important organizing role in the tutors. These bites exercise a range of simpler ideas in the curriculum. In electricity, for example, understanding Kirchoff's Law implies understanding a collection of more fundamental ideas: circuit geometry (e.g. parallel vs. series), resistor behavior, battery behavior, current, resistance, and voltage. Because of this organizing role, the problems generated from abstraction hierarchy bites are critical in diagnosis of student performance. Only abstraction hierarchy bites have sufficient perspective (i.e. connection to other bites representing fundamental ideas). to test the students performance in problems that integrate across several bites. Implementing this perspective is a current area of active research. Our intial work is presented in the section on tutoring components.

Definition Bites represent concepts that the student is to learn without being taught much background. Examples of this would be things like gravitational force used in our tutor for hydrostatics (Archimedes's Principle). Its important for the student to understand how gravity works when dealing with buoyancy, but it's not relevant why it works that way.

I/O Bites represent concepts that have a black-box behavior. The student needs to know that certain inputs produce certain outputs and the rule (formula) describing the behavior. The student does not need to know the justification for the behavior. The behavior of a resistor in an electric circuit is best represented in an I/O bite.

Several of our tutors combine the ideas of discovery microworlds with those of ITS to provide a directed discovery environment. These tutors provide a microworld which simulates some aspect of real life. This simplifies the discovery of concepts that are useful for understanding the "real" world by eliminating the deviations from the model that inevitably occur in "real" life. Because many students are lost in a purely discovery microworld, we want to provide the possibility of a more guided learning environment using ITS techniques. Ideally, the ITS allows the student to freely explore until it detects floundering, then it makes a suggestion.

Successfully exploration in computer microworlds requires the use certain scientific inquiry skills. Discovery Bites represent these skills. They enable the tutor to recognize when a student is floundering in his exploration and respond accordingly. An example of this type of bite is "vary only one variable while holding all else constant". This rule is necessary in the beginning of an exploration. [Shute and Bonar, 1986].

3.2 Tutoring Components: Diagnoser

There are three main tutoring components of the bite-sized tutoring architecture: the Diagnoser, the Student Model, and the Task Selector. We discuss each component in turn. The Diagnoser is invoked by some event that occurs during the tutoring session. What events invoke the Diagnoser is determined by the implementer of a specific tutor. In particular, we want to allow for different grain-sized observations of the student, ranging from a diagnosis only when a student completes a problem to a diagnosis based on the student's movement of the mouse every N milliseconds.

The *Diagnoser* class is best illustrated in our implementation of the Electricity tutor and the diagnosis associated with abstraction hierarchy bites. Consider what happens when a student responds to a problem constructed at some intermediate bite in the Kirchoff's Law abstraction hierarchy. That problem has been constructed from a number of component bites representing the fundamentals needed to understand the abstraction hierarchy bite. For example, a bite in the Kirchoff's Law abstraction hierarchy constructs problems based on component bites concerning resistors, current, circuit geometry, etc.

Once the system has a student response to a problem, the abstraction hierarchy bite begins a diagnosis. Using functionality provided by the *Diagnoser* class, the bite sends a message to each component bite asking if the domain knowledge in the component bite is relevant to the student's response, current tutoring goals, and the current tutoring mode.

If it is, the *Diagnoser* then checks to see if the student is misusing the concept taught by this bite. "Misuse" is defined by a specific diagnosis algorithm operating on the specific data of that bite. The *Diagnoser* then updates the student model accordingly. Note that the data for the student model are, of course, stored in the bites. When the *Diagnoser* has completed updating the bites, it invokes the *Task Selector* to choose what it should do next.

3.3 Tutoring Components: The Student Model

The *Student Model* maintains several components relevant to representing student performance. First, the Student Model contains a record of the events of the session. This is stored in a class variable of the *Bite* class so that all curriculum bites (which are instances of subclasses of *Bite*) have access to one copy of it. In addition, the *Student Model* specifies a series of instance variables that represent student performance on individual bites. We currently use a differential modeling scheme where we keep three seperate measures of the student's success with each bite. One is a measure over the entire tutoring session, one is a measure over the the last five events, and the last a measure of the last (or current) event. These measures are ratios of how many times the concept of each bite was used appropriately by the student divided by how many times it should have been used as determined by the *Diagnoser*.

3.4 Tutoring Components: The Task Selector

The basic flow of control of the tutor is based on *TutoringMode* objects stored in a stack located in a global object *TutoringSession*. *TutoringMode* instances set the local state for a series of instructional tasks. The *TutoringMode* has two instance variables useful to the Task Selector. One indicates criteria for the mode being satisfied, and one indicates some threshold for deciding that the student is floundering and currently unable to learn the current concept in the current mode.

Each mode object defines several messages. The *Initialization* message initializes the two instance variables mentioned above, based on the current student model. A *Process* message teaches the relevent bites in a manner consistent with the current mode (see below). A *Satisfaction* message will determine if the current mode is satisfied and what steps are to be taken when it is. It usually means popping the present mode instance off the *TutoringSession* stack and pushing a new mode instance on the stack. A *Threshold* message decides what actions to take when the student shows evidence of not being able to satisfy the mode object. This will usually initiate pushing some remedial mode object onto the stack.

The *Task Selector* first examines the stack. If it is empty the *Task Selector* creates a new instance of some default mode and sends the local *Initialization* message to the mode. The *Task Selector* then returns the control to the student. If the stack is not empty, the *Task Selector* sends the *Satisfaction* message. If the current mode is not satisfied, the *Threshold* message is then sent. Finally, if the threshold condition is not met the *Process* message is sent.

Tutoring modes describe the type of tutor-student interaction that is currently being used. We are implementing six of these modes:

Exploration -- The student is obtaining information from the microworld in order to refine and complete developing hypotheses.

Experimentation -- The student is performing some actions designed to confirm or differentiate hypotheses, whether explicitly stated or recognized by the tutor.

Elaboration -- The student is testing some previously confirmed hypothesis.

Didactic -- The tutor is driving the interaction by proposing problems for the student.

Demonstration -- The tutor takes over and demonstrates some concept explicitly.

Coaching - The tutor provides some hints that will help the student understand the bites in question.

4. Example Bite-Sized Intelligent Tutors

4.1 Bridge: An Intelligent Tutor for Programming

Bridge is a tutor that teaches computer programming. In Bridge, the student user is presented with problems which are of such complexity that they could be presented in the first ten weeks of an introductory programming course. Currently, the student passes through three phases while solving the problems.

In the first phase, the student constructs a set of step-by-step instructions using informal English phrases. In the next phase, the student matches these phrases to programming schemata we call "plans" [Soloway, et. al. 1982]. A program is built using a representation of these schema. In the final phase, the student matches the schemata to programming language constructs and uses these to build a programming language solution to the original problem. Currently the only language implemented in Bridge is Pascal, although other programming languages could be tutored using the same approach.

In the current Bridge implementation (see Bonar and Cunningham [1986]) the curriculum dependent bites are the programming plans and the plan specializations needed for each problem that Bridge can tutor. These plans fit into an abstraction hierarchy with the problem specific programming plans at the lowest level of the hierarchy. The *Diagnoser* determines whether a particular bite is not being used appropriately by comparing the student's current program with the requirements specified for that plan in the current phase. This information is represented by a requirements language. This language defines a group of operators which indicate various things about the plans, the correct order of their appearance, and their relationships to each other. Figure 2 shows an example of this language. This example shows some of the requirements of the Counter Variable Plan in Phase 1 for the Ending Value Averaging Problem.

Some of the requirements language operators we have found useful are:

Sequence -- this describes the order that the plans should appear in the program. Figure 2 shows three such sequence requirements. The '...' that seperates some plans indicates that zero or more plans can come between them in the student's solution.

Exists? -- This operator indicates that the plan mentioned must appear in the program. In figure 2, the Counter Variable Plan is required to be in the program.

AnyOf -- this is the equivalent of an OR operator. It is satisfied if any of it's arguments is satisfied.

All -- this is the equivalent of an AND operator. It is satisfied only if all of it's arguments are satisfied.

Not -- this is the usual NOT operator. It is satisfied only if it's argument is not satisfied.

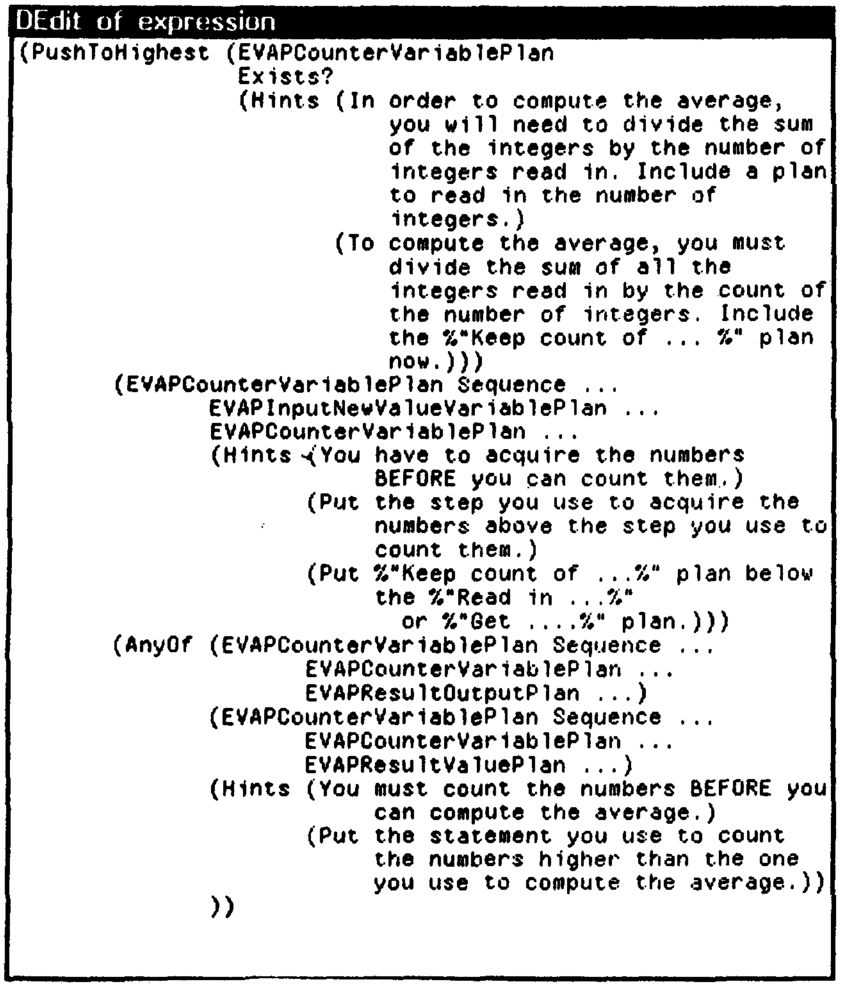
PushToHighest -- This manages several requirements at once and selects a hint dealing with the first unsatisfied plan.

4.2 The Invisible Hand: An Exploratory Microworld for Economics

Our economic microworld simulates an imaginary town which conforms to the laws of supply and demand. The student controls several variables, e.g. population, income per capita, interest rates, consumer preferance, number of suppliers, and weather. The student changes one or more of these variables and observes the resultant change on the other variables of the microworld. The student has several tools to aid this discovery, e.g. a notebook to record the changes observed in the variables and a graph package to observe relationships between variables. In addition to teaching an economics curriculum, the tutor teaches scientific discovery skills, so some of it's bites will be discovery bites (see Figure 3). The economics curriculum bites will teach about the patterns in the data collected by the students as they explore the microworld.

4.2 Ohm: A Tutor for Basic Electricity

The Electricity Tutor is designed to assist students with learning and developing basic problem solving skills in a learning-by-discovery environment. It teaches about voltage, current, and resistance in simple D.C. Circuits. It also teaches the process of discovery. The basic flow of the tutor is to pose problems that require a successively more



```
(PushToHighest (EVAPCounterVariablePlan
                Exists?
                (Hints (In order to compute the average,
                           you will need to divide the sum
                           of the integers by the number of
                           integers read in. Include a plan
                           to read in the number of
                           integers.)
                       (To compute the average, you must
                           divide the sum of all the
                           integers read in by the count of
                           the number of integers. Include
                           the %"Keep count of ... %" plan
                           now.)))
          (EVAPCounterVariablePlan Sequence ...
               EVAPInputNewValueVariablePlan ...
               EVAPCounterVariablePlan ...
               (Hints (You have to acquire the numbers
                           BEFORE you can count them.)
                      (Put the step you use to acquire the
                           numbers above the step you use to
                           count them.)
                      (Put %"Keep count of ...%" plan below
                           the %"Read in ...%"
                             or %"Get ....%" plan.)))
          (AnyOf (EVAPCounterVariablePlan Sequence ...
                      EVAPCounterVariablePlan ...
                      EVAPResultOutputPlan ...)
                 (EVAPCounterVariablePlan Sequence ...
                      EVAPCounterVariablePlan ...
                      EVAPResultValuePlan ...)
                 (Hints (You must count the numbers BEFORE you
                           can compute the average.)
                        (Put the statement you use to count
                           the numbers higher than the one
                           you use to compute the average.))
                 ))
```

Figure 2. Requirements Language from Bridge

sophisticated grasp on the basic concepts. Initally, students can solve problems with very weak models of the underlying phenomenon. As instruction proceeds, the problems require more accurate understanding. The problems focus on the application of Ohm's and Kirchhoff's Laws.

After successfully solving a problem the student is asked to fill in a skeleton sentence describing the principle they have used. This allows the tutor to determine if the student is understanding a bite that was successfully used to solve a problem. If the sentence is filled in correctly a new bite is given control. If the sentence is filled in incorrectly, the tutor develops another problem based on the current bite.

The tutor is constructed with Abstraction Hierarchy bites. We are currently expanding this to include other kinds of bites, providing us with the means to cover a wider range of electronics and more other kinds of tutoring tasks. Some of the bites to be integrated into the Electricity Tutor are AC circuits, capacitors, and inductors. Some of the tutoring tasks to be added are a "construction mode" where students are asked to build circuits and a metaphor mode where students can be shown certain phenomenon in a simpler metaphorical way.

4.4 Eureka: A Tutor for Hydrostatics Problems

Eureka employs an exploratory microworld environment that demonstrates the principles of buoyancy pertinent to Archimedes Principle [Klopfer, 1985]. The mode of learning is very similar to the economics tutor. The student explores the microworld, makes hypotheses when he or she discovers some relationship, and then tests those hypotheses with subsequent experiments. The student has the ability to change several variables in the "laboratory" environment: the mass of the block, the density of the liquid, the gravitational force, etc. He has the same tools available to aid his exploration that were described above in the economics tutor.

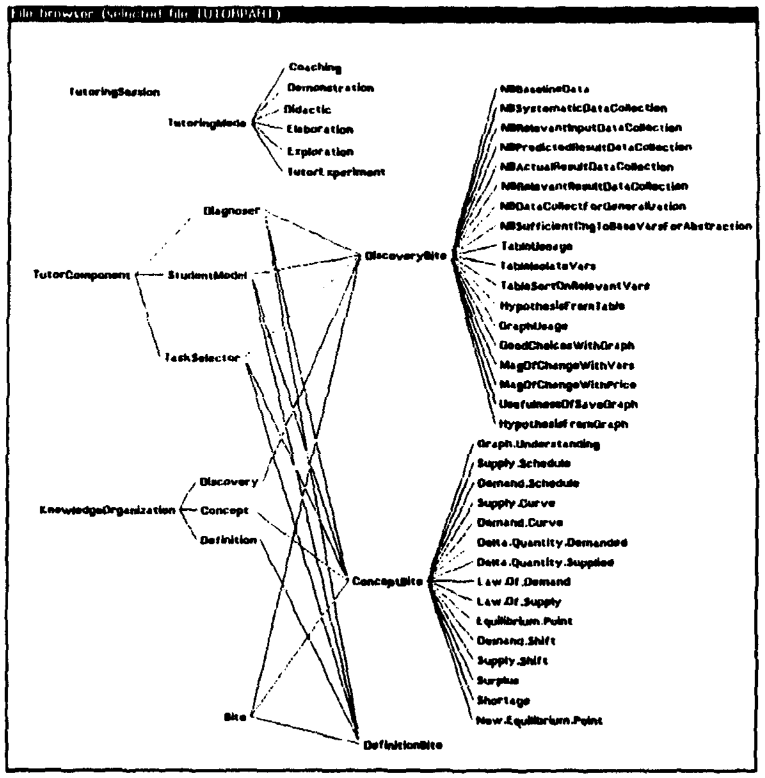


Figure 3. Bite-Sized Hierarchy from the Economics Tutor

Figure 4 shows the inheritance lattice for Eureka. This tutor is very similar in structure to the economics tutor. It uses discovery bites to tutor useful scientific skills. It has bites that teach about observeable patterns in the data collected during the session.

5. Concluding Remarks

We have illustrated a generic architecture for building intelligent tutoring systems. In particular, we have focused on techniques for domain independent representation of the knowledge to be taught. The key idea is to organize the tutor around objects that represent the knowledge to be taught, not around the various components of the tutor.

Although, each of the tutors discussed is implemented, very little code is actually shared between them. We are currently reimplementing several of the tutors to share code for all the basic components and knowledge organizations. We have also begun working with non-programming domain experts in designing an interface to let them design the tutor's curriculum. We are excited and encouraged as the experience and practical aspects of one tutor carry over into the implementation of the next.

6. Acknowledgements

This article discusses a series of intelligent tutors and educational computer environments developed at the Learning Research and Development Center. Many people at the center have contributed to the educational and computational aspects of the work discussed here. In particular, Valerie Shute developed the subject matter content and pedagogy for the Smithtown microeconomics intelligent microworld. Kalyani Raghavan has extended the microworld work of Valerie Shute and Peter Reimann in her development of the Electricity microworld. Those microworlds are experimental tools in a project to characterize and improve discovery learning. Joyce Ivill developed the non-exploratory parts of the Electricity tutor. Cindy Cosic, Leslie Wheeler, Mary Ann Quayle, Paul Resnick, and Gary Strohm have all worked on the tutors discussed in the article. Finally, Alan Lesgold, Stellan Ohlsson and Robert Glaser have all made important contributions to the ideas developed here.

This work was supported by the Office of Naval Research under contract numbers N00014-83-6-0148 and N00014-83-K0655 and the Air Force Human Resources Laboratory under contract number

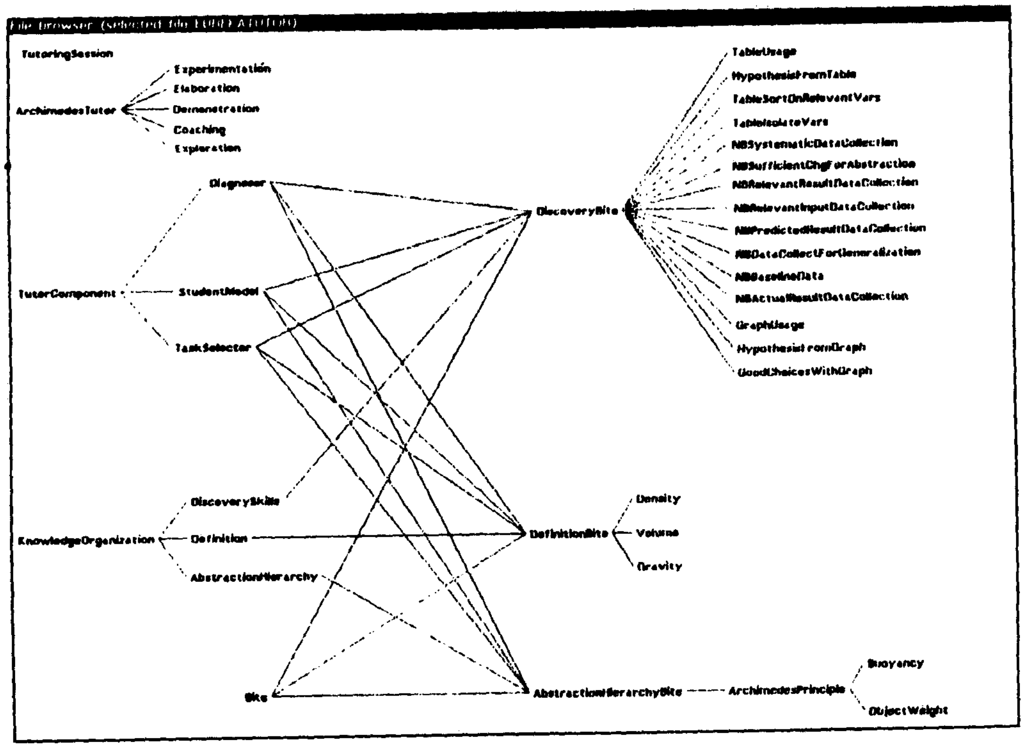


Figure 4. Bite-Sized Hierarchy from the Eureka Tutor

F41689-84-D-0002, Order 0004. Any opinions, findings, conclusions, or recommendations expressed in this report are those of the authors, and do not necessarily reflect the views of the U.S. Government.

7. References

Bonar, J. & Cunningham, R. (1986). Bridge: An Intelligent Tutor for Thinking About Programming. Learning Research and Development Center Technical Report.

Burton, R. R. & Brown, J. S. (1982). An investigation of computer coaching for informal learning activity. Appears in *Intelligent Tutoring Systems*, edited by Sleeman, D. and Brown, J.S., Academic Press.

Klopfer, L. E. (1985). Intelligent Tutoring Systems in Science Education: The Coming Generation of Computer-Based Instructional Programs. Appears in *Proceedings for the US-Japan Conference on Science Education*, Washington, D.C.

Shute, V. & Bonar, J. (1986). Intelligent Tutoring Systems for Scientific Inquiry Skills. *Proceedings of the 1987 Conference of the Cognitive Science Society*, Amherst, MA.

Soloway, E. M., Ehrlich, K., Bonar, J. G., & Greenspan, J. (1982). What Do Novices Know About Programming? Appears in *Directions in Human-Computer Interaction* edited by Shneiderman, Ben and Badre, Albert, Ablex Publishing Company.

Stefik, M. & Bobrow, D.G. (1986) Object Oriented Programming: Themes and Variations. *AI Magazine*, Winter 1986 (6:4), pp. 40-62.