DOCUMENT RESUME

ED 287 473 IR 012 870

AUTHOR Bonar, Jeffrey; And Others
TITLE An Object-Oriented Architecture for Intelligent Tutoring Systems. Technical Report No. LSP-3.
INSTITUTION Pittsburgh Univ., Pa. Learning Research and Development Center.
SPONS AGENCY Office of Naval Research, Arlington, Va.
REPORT NO UPITT/LRDC/ONR/LSP-3
PUB DATE 14 Aug 87
CONTRACT F41689-84-D-0002; N00014-83-6-0148; N00014-83-K0655
NOTE 18p.
PUB TYPE Reports - Research/Technical (143)

EDRS PRICE MF01/PC01 Plus Postage.
DESCRIPTORS Artificial Intelligence; Cognitive Processes; Computer Assisted Instruction; *Computer System Design; *Courseware; Diagrams; Higher Education; *Programed Tutoring
IDENTIFIERS *Knowledge Representation; University of Pittsburgh PA

ABSTRACT

This technical report describes a generic architecture for building intelligent tutoring systems which is developed around objects that represent the knowledge elements 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 the standard tutoring tasks of diagnosis, student modeling, and task selection. The goal of the bite-size tutor is an interface that allows the curriculum of a system to be supplied by a domain expert who is not a programming expert. Three bite-sized intelligent tutors have been implemented at the Learning Research and Development Center at the University of Pittsburgh: (1) Bridge: An Intelligent Tutor for Programming; (2) Smithtown: A Discovery World for Economics; and (3) Eureka: A Tutor for Hydrostatics Problems. Descriptions of these three tutors conclude the report, and 11 references are listed. (RP)

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UPITT/LRDC/ONR/LSP-3	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Learning Research & Development Center, Univ. of Pittsburgh 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Personnel & Training Research Programs Office of Naval Research (code 1142PT)
6c. ADDRESS (City, State, and ZIP Code) 3939 O'Hara Street Pittsburgh, PA 15260	7b. ADDRESS (City, State, and ZIP Code) 800 North Quincy Street Arlington, VA 22217-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-85-K-0655/P00004
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 61153N; PROJECT NO. RR04206; TASK NO. RR04206-00; WORK UNIT ACCESSION NO. NR442c524

11. TITLE (Include Security Classification)
An Object-Oriented Architecture for Intelligent Tutoring Systems (UNCLASSIFIED)

12. PERSONAL AUTHOR(S)
Jeffrey Bonar, Robert Cunningham, and Jamie Schultz

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	1987, August 14	

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	
05	09		Artificial intelligence; Cognitive psychology; Cognitive science; Instruction, Computer-based; Instruction, Intelligent computer-based; Training; Tutors; Tutors,intelligen

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report describes an object-oriented architecture for intelligent tutoring systems, oriented around objects that represent the knowledge elements 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 at the Learning Research and Development Center.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Susan M. Chipman	22b. TELEPHONE (Include Area Code) (202)696-4318
	22c. OFFICE SYMBOL ONR 1142pt

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

An Object-Oriented Architecture for Intelligent Tutoring Systems

Jeffrey Bonar, Robert Cunningham, Jamie Schultz

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

Technical Report No. LSP-3

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.

This work was supported by the Office of Naval Research, under Contract No. N00014-83-6-0148 and N00014-83-K0655 and the Air Force Human Resources Laboratory under Contract No. 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.

Introduction

Many projects in intelligent computer-based instruction depend upon detailed but general theories of human learning or diagnoses of the learner's cognitive state (see, for example, Bonar and Cunningham [1986], Ohlsson and Langley [1984] and Van Lehn [1984]). We feel there is also an enormous potential for systems based on theory-driven analyses of domain expertise. For particular fields, for example, cognitive science researchers have developed accounts of skilled performance and of impediments to skilled performance. Such accounts could be used to develop useful and interesting instructional systems. Widespread development of this kind of tutor would be faciliated by a general intelligent tutoring system architecture and tools to support development. In this paper we desribe our first steps toward such an architecture, called the Bite-Size Tutor.

The Bite-Size Tutor is a general intelligent tutoring system shell that provides the curriculum-independent part of an intelligent tutor and specifies an organization for the curriculum knowledge, to be supplied by a domain expert. With the Bite-Sized Tutor, we can exploit the expert system approach currently being applied in many intelligent tutoring system projects. In this approach, competent performance in a domain is analyzed. Novices learning that domain may be observed. The results of these analyses and observations are a "cognitive task analysis" of the domain and a "bug catalog" of common novice problems in the domain. "Cognitive task analyses" (Learning Research & Development Center, May 1986) are analyses beyond a behavioral "rational task analysis" that specifically attend to the underlying cognitive skills and representations involved in competent performance. "Bug catalogues" (Brown & Burton, 1978; VanLehn, 1982, 1983) describe systematic errors or misconceptions that are likely to impede learning or skilled performance. Taken together, the cognitive task analysis and the bug catalog constitute the "curriculum" of most intelligent tutors.

Our key goal with the Bite-Size Tutor is an interface that allows the curriculum of a system to be supplied by a domain expert who is not a programming expert. This is a crucial goal -- creating the curriculum for any kind of computer based instruction is a demanding and time consuming task, running into hundreds of hours of development for each hour of instruction. The toy domains, microscopic curriculums, and limited instructional activities of experimental intelligent tutors must be expanded to complete curriculums that meet recognized instructional needs with an array of techniques and activities for the student. Such an expansion requires extensive input by domain experts who are not programmers. We in the intelligent tutoring community must provide the tools to allow this approach.

In this article, we first discuss problems with current intelligent tutoring system architectures. We then propose our solution: an architecture that uses the structures of an object-oriented programming language as domain-independent modules for intelligent tutors. We discuss the detailed architecture of the Bite-Size Tutor, and illustrate this architecture with several examples from projects being developed at LRDC. Finally, we touch upon future research directions.

Problems with current intelligent tutoring systems

Close examination of most current intelligent tutoring system implementations shows them to be complex and unwieldy. (We want to emphasize that this discussion is not a criticism of any particular intelligent tutoring system, but a general problem that appears in many systems.) Pieces of programming code are repeated in several places; closely-related information is spread apart. 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 intelligent tutoring system. Intelligent tutoring systems are usually conceived as a series of semi-independent components like "explainer," "diagnoser," "tutor," and "user modeler." The problem is that these components need to share many diverse pieces of knowledge. The knowledge needed for different components typically overlaps considerably. Functional components whose roles are quite clearly delineated in an abstract description of the system may be implemented with code diffused through many parts of the system. The systems, therefore, are not modular. In particular, they do not allow for addition of new domain knowledge or new approaches to the pedagogical tasks.

The WEST tutor [Brown and Burton, 1982] provides an example of these problems. As one of the most intellectually important "classic" intelligent tutoring systems, it serves as a useful foil for this discussion. It can be viewed in two ways: in terms of its "issues" (the fundamental lessons the system is prepared to teach the student) and in terms of its components (e.g. "expert," "differential modeler," "tutorial selector.") 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 its multiple, overlapping representations of issue knowledge. Besides obscuring the organization of its knowledge, the current implementation of WEST makes it difficult to reuse and extend parts of the tutor. Given the many open research issues for intelligent tutoring systems, this is a serious problem.

In general, we need a tool that enables the development of tutoring systems much more rapidly than now possible. Ideally, such a tool will allow a subject domain expert or a teacher (who is not necessarily a programmer) to modify the domain knowledge and the tutor-student interaction

without reimplementing the system at each step. Finally, we need a tool to make it easier for those who develop tutors to test their systems as they are designed.

An Object-Oriented Intelligent Tutoring System Architecture

We propose to take advantage of the character of an object-oriented programming language to develop an architecture that is modular and is therefore both comprehensible and easily modified. Object-oriented programming allows the programmer to create a *toolkit* of *objects* that represent items of interest in the application area of the program. Objects represent items in the world by containing both data, the state of the object, and programs, operations that can change the state. Objects can also share structure, with one object defining the structure for several other objects. Such an object is called a *class*. Typically an object specializes a structure it inherits from above, and in turn defines the structure for a lower-level set of objects. An object communicates by sending a message to another object and requesting some action. An object responds to a message by running one of its programs, thereby changing its state or sending new messages to other objects.

Although objects, classes, inheritance, and messages are the crucial constructs of object-oriented programming, the notions of *toolkit* and *protocol* are central to understanding the power of the approach. A toolkit provides a set of objects designed to be specialized and protocols for using those objects. Objects in a toolkit provide a range of capabilities designed to be specialized to particular applications. A *protocol*, in the object-oriented sense, is a set of messages that are defined for a broad range of objects. For example, we could design a drawing system where objects corresponding to rectangles, circles, and characters all responded to the messages `draw`, `erase`, `move`, etc. Note that each kind of object is free to implement these messages differently. This is the essence of a protocol: a general set of capabilities for simplicity, with a mechanism for accommodating the complexity of actual differences.

The Bite-Size architecture is a toolkit for implementing intelligent tutors. In the Bite-Size architecture, everything the system knows is stored in objects. Some of these objects will correspond to "issues" in the sense used by Burton and Brown (1982): things that the system can understand and talk to the user about. Many of the different objects representing the domain will share common substructure. For such objects, the standard class 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 on the basis of the issues that the system recognizes, 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 objects. So, for example, there are objects that can implement a component like a diagnoser. The class Diagnoser will specify the local data needed to perform the diagnostic function and the algorithms to use that data. 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 intelligent tutoring system components are implemented as classes and inherited by the Bites. Consider an example where two kinds of diagnosers are 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 diagnostic capability from DiagnoserA or DiagnoserB as appropriate.

The proposed architecture solves the problems described above 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 function to the domain classes. Like the domain element classes, component classes use inheritance to represent shared structure.

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, Input/Output 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 general 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 Kirchhoff's Law from our electricity tutor. The two highlighted nodes show the relationship between the specific concept, "current is unchanged

across an uninterupted wire," and the more general concept, "Kirchhoff's Law." The "UninterruptedS" bite is a specific version of the "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 Kirchhoff'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 abstraction hierarchy bites generate are critical for the diagnosis of student performance. Only abstraction hierarchy bites have sufficient perspective (i.e. connection to other bites representing fundamental ideas) to test the student's performance in problems that integrate across several bites (Lesgold & Ivill, 1987). Implementing this perspective is a current area of active research. Our initial work is presented in the section on tutoring components.

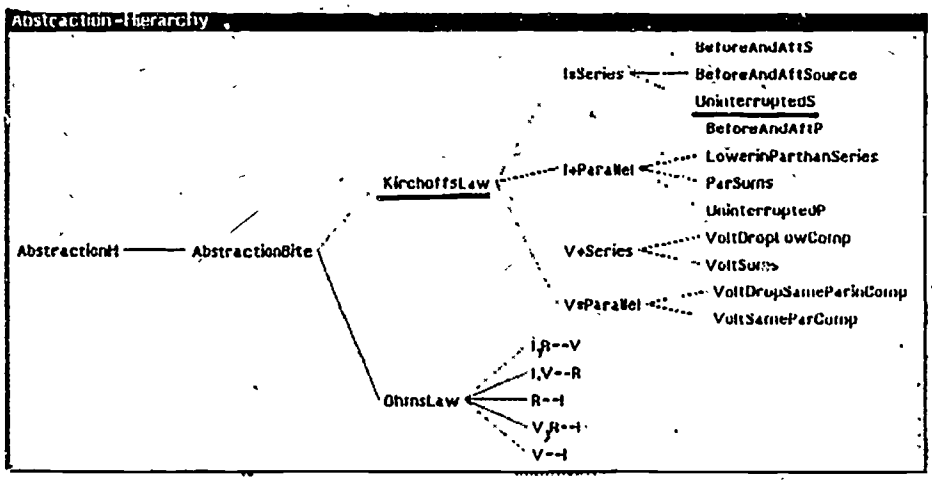

Figure 1. Abstraction Hierarchy from the Electricity Tutor

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

Input/Output Bites represent concepts that have a black-box behavior. The student needs to know that certain inputs produce certain outputs and needs to know 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.

Discovery Bites enable several of our tutors to combine the advantages of student-initiated learning in discovery worlds with support for students who lack the skills to learn efficiently from a pure discovery world. These tutors provide a simulation of some aspect of a domain. The intelligent tutoring system allows the student to explore the simulation freely until it decides the student is floundering, then it makes a suggestion. Discovery Bites represent the inquiry skills. An example of this type of bite is "vary only one variable while holding all else constant." [Shute and Bonar, 1986].

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. The implementor of a specific tutor determines what events invoke the Diagnoser. In particular, we want to allow for different grain-sized observations of the student, ranging from making a diagnosis only when a student completes a problem to making 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. Consider what happens when a student responds to a problem constructed at some intermediate bite in the Kirchhoff'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 Kirchhoff'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 capabilities 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.

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 separate 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 is 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.

Tutoring Components: The Task Selector

The basic flow of control of the tutor is based on Tutoring Mode objects stored in a stack located in a global object Tutoring Session. Tutoring Mode instances set the local state for a series of instructional tasks. The Tutoring Mode 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 relevant 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 if the Tutoring Session 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 discovery world in order to refine and complete developing hypotheses.

-Experimentation -- The student is performing some action 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.

Example Bite-Sized Intelligent Tutors

Bridge: An Intelligent Tutor for Programming

Bridge is a tutor that teaches computer programming. In Bridge, the student user is presented with problems of a complexity appropriate to the first ten weeks of an introductory programming course. Bridge coaches the student through three phases of problem-solving for each problem posed. In the first phase, the student constructs a set of step-by-step instructions by choosing and arranging informal English phrases. In the next phase, the student matches these phrases to visual representations of programming schemata we call "plans" [Soloway et al., 1982] and combines the representations to build a runnable program. In the final phase, the student uses the visual representation as a guide to building a correct programming language solution to the original problem. Currently Bridge tutors Pascal; other programming languages could be tutored using the same approach.

In the current Bridge implementation (Bonar & 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 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 that 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.

```
DEdit of expression
(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

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

-Sequence -- this describes the order in which the plans should appear in the program. Figure 2 shows three such sequence requirements. The '...' that separates 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 its arguments is satisfied.

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

-Not -- this is the usual NOT operator. It is satisfied only if its argument is not satisfied.

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

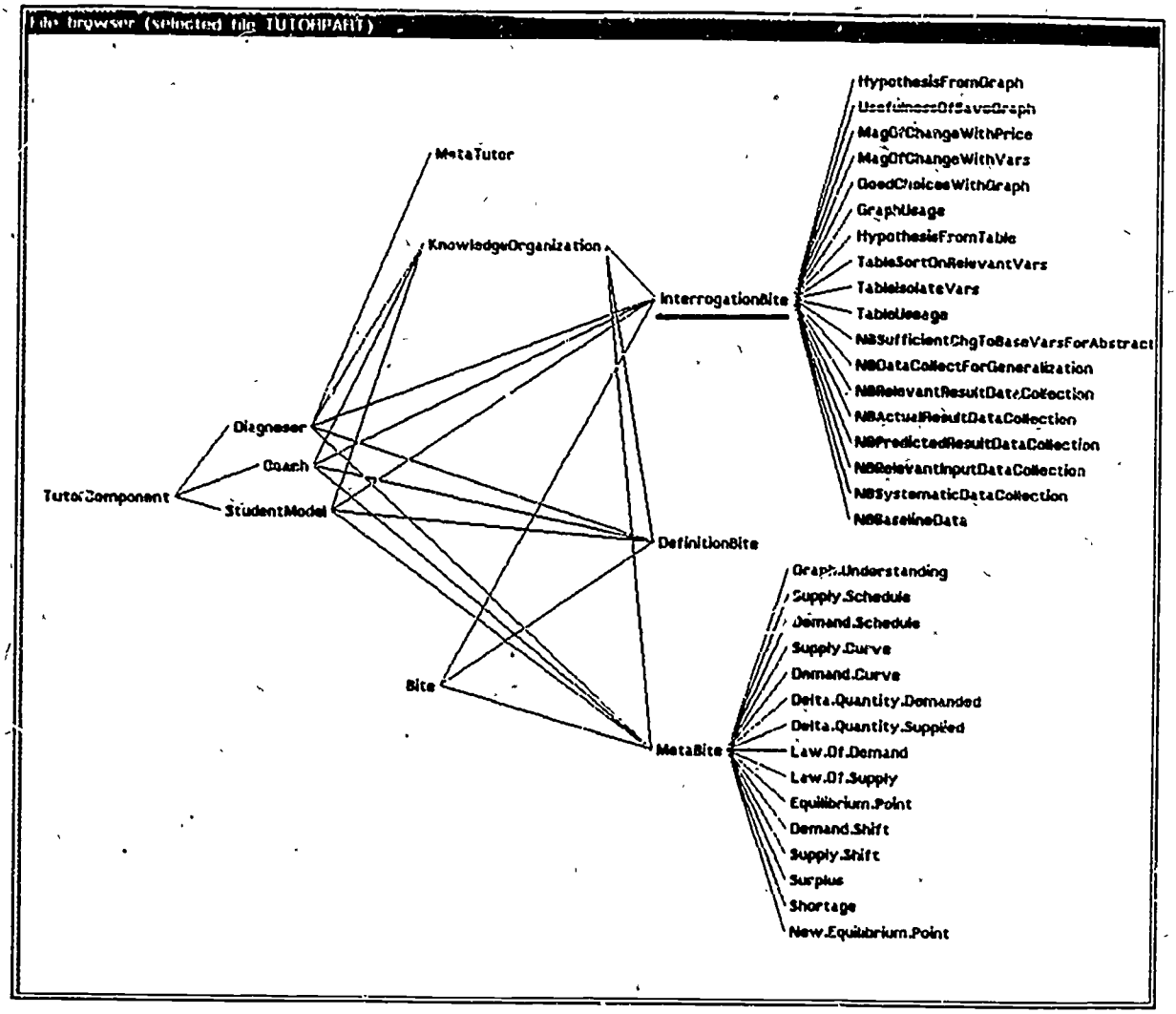

Figure 3. Bite Hierarchy for Smithtown. "InterrogationBite" is a class of Discovery Bites.

Smithtown: A Discovery World for Economics

The economics discovery world 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 preference, 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 world. The student has several tools to aid 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 its bites are 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.

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.

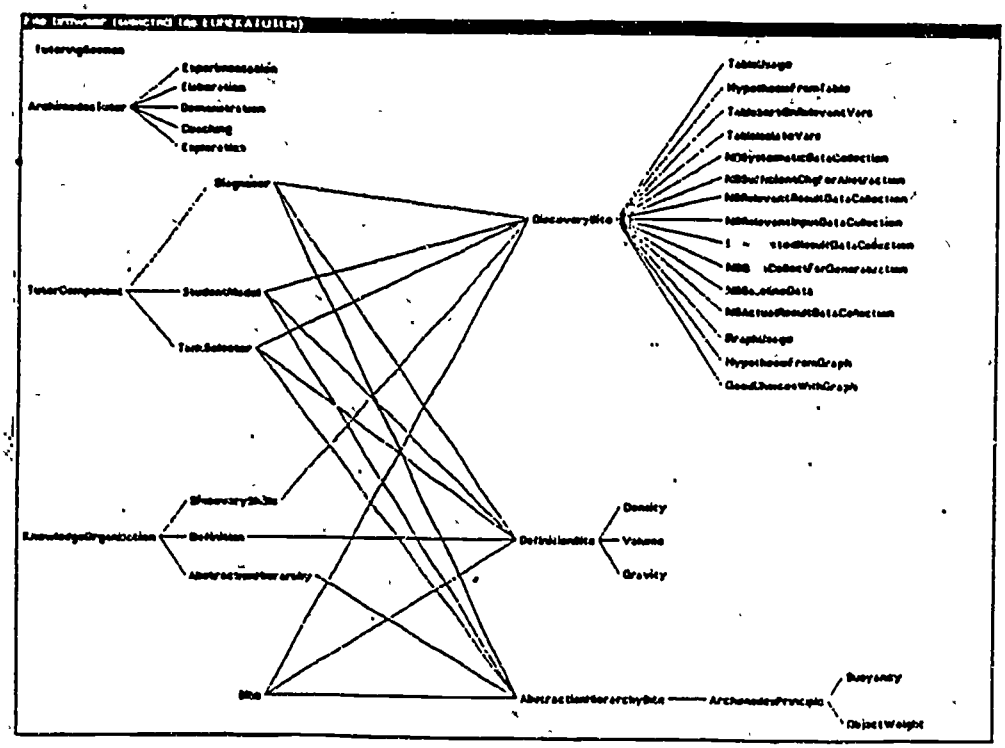

Figure 4. Bite-Sized Hierarchy from the Eureka Tutor

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 observable patterns in the data collected during the session.

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.

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 discovery world. The discovery worlds are experimental tools in a project of Robert Glaser's to characterize and improve discovery learning. Kalyani Raghavan has extended the discovery world work of Valerie Shute and Peter Reimann in the development of Voltaville and in further work with Smithtown. Joyce Ivill developed MHO, the Bite-Size electricity tutor, to which Andrew Bowen also made a substantial contribution. 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 and Stellan Ohlsson have made important contributions to the ideas developed here.

References

Bonar, J. G., & Cunningham, R. (1986). *Bridge: An intelligent tutor for thinking about programming.* Technical Report. University of Pittsburgh, Learning Research and Development Center .

Brown, J. S., & Burton, R. B. (1978). Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science, 4,* 379-426.

Burton, R. B. & Brown, J. S. (1982). An investigation of computer coaching for informal learning activity. In D. Sleeman & J.S. Brown (Eds.), *Intelligent Tutoring Systems,* pp. 79-98. London: Academic Press.

Klopfer, L. E. (1985). Intelligent tutoring systems in science education: The coming generation of computer-based instructional programs. In *Proceedings of the US-Japan Conference on Science Education,* Washington, D.C.

Learning Research & Development Center. (May, 1986). *Guide to cognitive task analysis.* Technical Report.
University of Pittsburgh, Learning Research & Development Center.

Lesgold, A. M., & Ivill, J. (1987). *Toward intelligent systems for testing.* Technical Report LSP-1. University of Pittsburgh: Learning Research and Development Center.

Shute, V. & Bonar, J. (1986). Intelligent tutoring systems for scientific inquiry skills. *Program of the Eighth Annual Conference of the Cognitive Science Society* (pp. 353-370). Hillsdale, NJ: Lawrence Erlbaum Associates.

Soloway, E. M., Ehrlich, K., Bonar, J. G., & Greenspan, J. (1982). What do novices know about programming? In B. Shneiderman and A. Badre (Eds.), *Directions in human-computer interaction.* Norwood, NJ: Ablex.

Stefik, M. & Bobrow, D.G. (1986) Object oriented programming: Themes and variations. *AI Magazine, 6,* 40-62.

VanLehn, K. (1981). *Bugs are not enough: Empirical studies of bugs, impasses and repairs in procedural skills.* Technical Report CIS-11 (SSL-81-2). Palo Alto, CA: Xerox Palo Alto Research Center. [Also, (1982), Journal of Mathematical Behavior, 3, 3-71.]

VanLehn, K. (1982). *Felicity conditions for human skill acquisition: Validating an AI-based theory.* Technical Report CIS-21. Palo Alto, CA: Xerox Palo Alto Research Center.