

AL-TP-1993-0012

AD-A264 320



DTIC
ELECTE
MAY 17 1993
S C D



ARMSTRONG
LABORATORY

AN OBJECT-ORIENTED TUTOR TO
TEACH TROUBLESHOOTING

William R. Murray

FMC Corporate Technology Center
1205 Coleman Avenue, Box 580
Santa Clara, CA 95052

HUMAN RESOURCES DIRECTORATE
TECHNICAL TRAINING RESEARCH DIVISION
7909 Lindbergh Drive
Brooks Air Force Base, TX 78235-5352

April 1993

Final Technical Paper for Period February 1991 - March 1992

Approved for public release; distribution is unlimited.

93 5 14 077

93-10897



62P8

AIR FORCE MATERIEL COMMAND
BROOKS AIR FORCE BASE, TEXAS

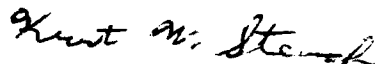
NOTICES

This technical paper is published as received and has not been edited by the technical editing staff of the Armstrong Laboratory.

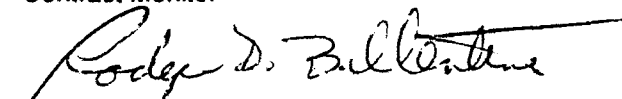
When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

The Office of Public Affairs has reviewed this paper, and it is releasable to the National Technical Information Service, where it will be available to the general public, including foreign nationals.

This paper has been reviewed and is approved for publication.



KURT W. STEUCK
Contract Monitor



RODGER D. BALLENTINE, Colonel, USAF
Chief, Technical Training Research Division

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (2704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE April 1993	3. REPORT TYPE AND DATES COVERED Final February 1991 - March 1992		
4. TITLE AND SUBTITLE An Object-Oriented Tutor to Teach Troubleshooting		5. FUNDING NUMBERS C - F33615-91-C-0004 PE - 62205F PR - 1121 TA - 09 WU - 77		
6. AUTHOR(S) William R. Murray		8. PERFORMING ORGANIZATION REPORT NUMBER		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) FMC Corporate Technology Center 1205 Coleman Avenue, Box 580 Santa Clara, CA 95052		10. SPONSORING / MONITORING AGENCY REPORT NUMBER AL-TP-1993-0012		
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Armstrong Laboratory Human Resources Directorate Technical Training Research Division 7909 Lindbergh Drive Brooks Air Force Base, TX 78235-5352		11. SUPPLEMENTARY NOTES Armstrong Laboratory Contract Monitor: Kurt W. Steuck, (210) 536-2034		
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.		12b. DISTRIBUTION CODE		
13. ABSTRACT (Maximum 200 words) This report describes a general approach to implementing troubleshooting tutors for complex hydraulic-electronic-mechanical systems in wide use throughout industry and the military. With this approach, user interfaces are rapidly constructed from scanned-in schematics, and animation is easily added with hypermedia tools to show the internal operation of the device. Object classes and methods are used to describe device structure and to implement the functionality of both device components and tutor components. The object-oriented approach facilitates reuse of these components and their portability across platforms, programming languages, and domains. It also supports the ability to generate instructional interactions, such as explanations, directly from device and component descriptions.				
14. SUBJECT TERMS Artificial intelligence Troubleshooting Computer-based training Training		15. NUMBER OF PAGES 66		
		16. PRICE CODE		
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	

CONTENTS

1. Introduction	1
2. Object-oriented design	4
2.1 Object-oriented programming.....	4
2.2 Benefits for this application.....	5
3. The subject matter representation	7
3.1 Device parts and their relationships	7
3.2 Device cycle of operation.....	8
4. The qualitative model of device operation	10
4.1 Modeling normal operation	10
4.2 Modeling faulted operation.....	12
5. The fault-space model of troubleshooting	13
5.1 Generating a fault-effect table.....	14
5.2 Using the table to track candidates and choose tests	15
6. Explanations and assessments from qualitative models.....	18
6.1 Generating explanations of device operation.....	18
6.2 Generating explanations of troubleshooting actions	18
6.3 Generating test questions and cases for assessment	19
7. A user interface that shows device parts and animation.....	20
7.1 Capturing schematics and adding animation and interactive capabilities.....	21
7.2 The graphics interface in the Lower Hoist Tutor	23
8. Interpreting student performance	25
8.1 A generic representation of target skills	26
8.2 Representing skill acquisition and tutor uncertainty.....	26
8.3 Representing generalized student capabilities	31
8.4 Using endorsements to make decisions	32
9. Controlling the tutor with a dynamic planner.....	32
9.1 The plan representation.....	33
9.2 Plan generation	35
9.3 Plan revision.....	37
9.4 Planning rules and plan critics.....	39
9.5 Integration of the planner and student model	39
10. Knowledge acquisition tools.....	42
10.1 Knowledge base debugging tools	42
10.2 Application of machine learning algorithms	43
11. Related work.....	43

Accession for	
NTIS	CRA&I
DTIC	TAB
Unannounced Justification	
By	
Distribution /	
Availability C	
Dist	Avail and Special
A-1	

12. Conclusion.....	44
Acknowledgements.....	45
References	46
Appendix I—Implementing semantic networks with objects.....	48
1.1 ISA and PART semantic networks.....	48
1.2 Implementation in an object-oriented language.....	52
Appendix II—An object-oriented implementation of the ESM.....	54

List of Figures

Figure 1	Schematic of lower hoist	3
Figure 2	A portion of the class hierarchy of parts in the Lower Hoist Tutor	7
Figure 3	Representation of device subcycles in Lower Hoist Tutor	9
Figure 4	Example of two connected parts showing port connections	11
Figure 5	Example where model normal operation will not terminate	11
Figure 6	A simple decomposition of troubleshooting skills	20
Figure 7	Overview of approach to building graphical device simulation	21
Figure 8	Implementation of Lower Hoist Tutor on MacIvory Platform	24
Figure 9	Decomposition of troubleshooting into subskills	27
Figure 10	Sample part hierarchy	31
Figure 11	Sample lesson plan showing plan representation	34
Figure 12	An ISA hierarchy	48
Figure 13	A PART-OF hierarchy	49
Figure 14	Object hierarchy to implement inheritance of ISA hierarchy	50
Figure 15	Linked instances of class concept hierarchy to represent class-subclass-member relationships in ISA hierarchy	51
Figure 16	Top level class hierarchy in object-oriented implementation	52

List of Tables

Table 1	Sample fault-effect table for the lower hoist domain	15
Table 2	One set of endorsement reliability classes	29

PREFACE

The mission of the Intelligent Training Branch of the Technical Training Research Division of the Human Resources Directorate of the Armstrong Laboratory (AL/HRTI) is to design, develop, and evaluate the application of artificial intelligence (AI) technologies to computer-assisted training systems. The current effort was undertaken as part of AL/HRTI's research on intelligent tutoring systems (ITS) and ITS development tools. The work was accomplished under workunit 1121-09-77, Machine Learning Techniques. The proposal for this research was solicited using a Broad Agency Announcement.

1. Introduction

This paper presents an object-oriented approach to implementing tutors that teach troubleshooting of complex machines. Such a tutor can be used within industry or the military as either a classroom aid or as a standalone instructional system. As a classroom aid it can be projected overhead to show the internal operation of a complicated system of hydraulic, electronic, and mechanical parts. As a standalone system it can provide students with practice on troubleshooting the system. Students can insert faults and see how the computer-based tutor solves them, or practice troubleshooting on faults the tutor has selected. The tutor can help the student with feedback and hints, taking over if necessary. Alternatively, the classroom instructor can provide assistance and the computer system can serve just as a practice environment.

The approach is illustrated with the implementation of the Lower Hoist Tutor. This tutor shows the internal operation of a complex hydraulic-electrical-mechanical system under both normal conditions and when one or more faults have been inserted. It can also demonstrate troubleshooting when a student has inserted a fault into the device. Conversely it can provide feedback on a student's troubleshooting when the tutor has inserted a fault into the lower hoist. The tutor can determine when troubleshooting actions are irrelevant or redundant, suggest the best action to perform next, and take over to diagnose the fault if the student reaches an impasse.

This lower hoist is one of the fifteen major assemblies of FMC's Mark-45 naval gun mount. A schematic of the lower hoist assembly is shown in Figure 1. It contains about 40 parts consisting of hydraulic valves, electrical solenoids and sensors, and mechanical gears and latches. The internal components of the Mark-45 and the lower hoist assembly is typical of many systems found within the military. Other similar systems include FMC's Mark-13 and Mark-26 launching systems and the hydraulic equipment used in FMC's harvesters. Another example of a similar domain is the helicopter blade fold domain used by Towne and Munro's IMTS [Towne and Munro, 89].

The approach described is generic to all assemblies of discrete-state parts where measurements are binary-valued. This includes logic gates that are high or low, as well as hydraulic valves and pipes that measure high pressure (also called PA, or pressure applied) or low hydraulic pressure (also called TANK). Parts can have any number of discrete

states. A hydraulic valve might have three states (left, center, and right) for example. Extensions can be easily made to the design if measurements can have multiple discrete outcomes. Some parts that have an infinite number of states can be modeled as being in one of several discrete states, in which the design still applies. The design does not apply to time-varying real-valued systems, such as electronic oscillators.

The tutors constructed are intended to teach troubleshooting based on a thorough understanding of device operation. A model of troubleshooting that is generic to the domains described above is part of the design. The model assumes that only one fault is inserted into the model at a time. The device model can qualitatively model any number of faults but the troubleshooting approach can handle only one. A more complicated model-based troubleshooting approach, such as that described in [De Kleer and Williams, 87], could have been incorporated but it would significantly add to the complexity of the tutor. It is also not clear whether or not debugging multiple unrelated faults is an important troubleshooting skill.

This design is described in enough detail that it should be readily implemented in most object-oriented languages and on most platforms, such as the Macintosh or IBM-PC, that support HYPERCARD-like graphics. The Lower Hoist Tutor, used as an example of the object-oriented approach to building a troubleshooting tutor, was implemented on the MacIvory. The LISP portion of the machine was used to implement the qualitative models of the device and of troubleshooting. The Macintosh portion of the machine was used to implement the graphics interface in SUPERCARD, which is essentially an improved version of HYPERCARD.¹

The first part of this report describes how such a system can be implemented, using the Lower Hoist Tutor as a working example of such a system. The second part (Sections 8, 9, and 10) describe extensions to the Lower Hoist Tutor that are not fully implemented.²

¹Several trademarks are mentioned in this report. Macintosh, Hypercard, and HyperTalk are registered trademarks of Apple Computer, Inc. Supercard and SuperTalk are trademarks of Silicon Beach Software, Inc. Spinnaker PLUS is a trademark of Spinnaker Software Corporation. ToolBook is a trademark of Asymetrix. MacIvory and Ivory are trademarks of Symbolics, Inc.

²Those parts of the design described as "extensions" have not been fully implemented. All other aspects of the design have been fully implemented in the lower hoist domain, unless statements are made to the contrary.

The extensions for planning and student modeling have been partially implemented but there is no implementation yet for the extensions for knowledge acquisition.

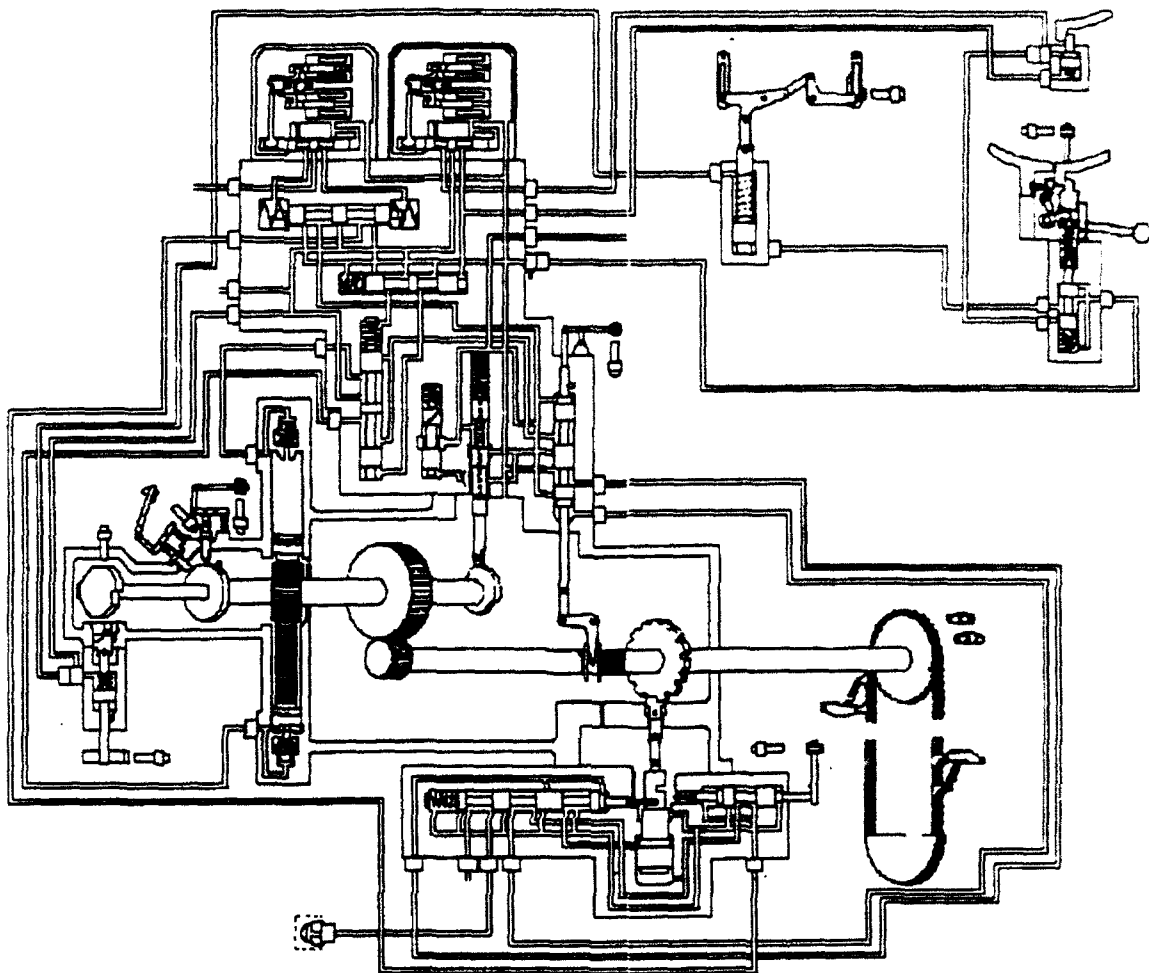


Figure 1. Schematic of lower hoist

These extensions to the Lower Hoist Tutor provide new capabilities to plan lessons and track student progress. By incorporating the ability to plan dynamically the tutor need not just be a simulation environment coupled with a purely reactive tutor. Instead, it can generate customized lesson plans and revise them during instruction as the tutorial situation changes. The plan and its revisions are designed to fit the student, so the accuracy of the tutor's model of the student is important. Although a numeric student can be used a non-numeric representation of uncertainty, called an *endorsement-based student model*, better supports the dynamic planner. These extensions—the planner and endorsement-based student model—are explained more fully in Sections 8 and 9. The key point is that they

provide the tutor with greater autonomy and an ability to pursue instructional goals. For example, a lower hoist tutor with these extensions could be deployed shipboard, where one use would be to provide refresher courses that are tailored to individual needs.

The rest of this report presents the Lower Hoist Tutor and a suggested approach to building similar tutors. Section 2 provides background on object-oriented programming and how it supports portability, reusability, and extensibility in the tutor design. Section 3 describes how knowledge about the device is represented. Section 4 explains how a qualitative model is used to predict normal and faulted operation of the device. A separate model of the troubleshooting process is described in Section 5. That model is used to demonstrate troubleshooting or evaluate student troubleshooting. Section 6 describes how domain-independent instructional activities can be implemented. In this way explanations and test questions can be generated automatically from device descriptions. Section 7 describes how the use of a HYPERCARD-based graphics interface greatly simplifies the building of a user interface that incorporates part schematics.

The proposed extensions to the Lower Hoist Tutor are described in Sections 8, 9, and 10. Section 8 describes the addition of an endorsement-based student model to track student progress. Section 9 adds a dynamic planner for control. Section 10 explores extensions for knowledge acquisition. Section 11 explains how the approach presented here builds on earlier work in intelligent tutoring systems. Section 12 summarizes the key features of this design and its contributions.

2. Object-oriented design

This section provides a brief review of object-oriented programming and explains how object-oriented programming supports portability, extensability, and code reuse of tutor and device components.

2.1 Object-oriented programming

Classes in an object-oriented programming language such as C++, CLOS (Common LISP Object System), or Ada, provide a user-defined kind of data structure. A class is similar to a *structure* that defines slots that must be present in that structure. One difference between a class and a structure is that the class can associate *methods* with objects of that class. For example, we can define either a data structure or a class to represent complex numbers. Both can specify that objects of that structure or class will have two slots, one for the real part and one for the imaginary part of the complex number. However, the class can also

define functions and procedures that apply only to objects that belong to that class. Such functions and procedures are the methods for that class. Methods for the complex number class could be defined to return polar coordinates. These methods would not apply to objects of other classes. In summary, classes define both a data structure and associated procedures and functions for the data structure.

Classes can be defined as specializations of other classes. Such classes *inherit* the slots and methods of their parent classes. This means that they have the same slots as the parent and the same methods, unless they redefine the slots or methods. Any redefined slots or methods typically override the inherited ones unless more complicated mechanisms for combining local and inherited methods are used. The more specialized class can also define new methods or slots that do not apply to the more generic class. One could define a class **window** that draws a window on a screen and then a more specialized class **labeled-window** that also adds a label to the window. It could do this by first calling the method for drawing the basic window and then calling its own method for adding the label.

2.2 Benefits for this application

Classes and methods provide the following general benefits:

1. *modularity*—methods that should not be called from outside of a class can be hidden from outside users. The implementor is free to change the internal implementation as long as the interface methods defined for the class are unaffected.
2. *code reusability*—objects that share functionality with other objects can inherit code and just change the methods that do not apply, or add new methods for what is missing, as in the window example above.
3. *extensability*—a user can create a new class that inherits slots and methods from some other class. The new class can define methods that call the inherited methods within methods of the same name. So instead of just replacing inherited methods the new methods can build on them.
4. *data abstraction*—a description of the classes and methods of a system provides a useful explanation of how the system operates without becoming overly immersed in implementation details.

The specific benefits for building intelligent tutoring systems such as the Lower Hoist Tutor are:

1. *modularity*—the implementation of methods defined for the device simulation or other parts of the tutor can be hidden from the user.
2. *code reusability*—a library of object classes can be built for the common hydraulic, mechanical, and electrical parts of a system. New assemblies of the system can be rapidly built up using instances of the library classes. Thus the classes used in the Lower Hoist Tutor could be reused for other assemblies of the Mark-45.
3. *extensability*—if a generated instructional explanation is not quite appropriate a method can be defined to take the output and alter it to be more appropriate for a particular application.
4. *data abstraction*—the description of the approach to building tutors such as the Lower Hoist Tutor will be given primarily at the level of classes and methods. Although the implementation of the Lower Hoist Tutor is in CLOS the same design applies to other object-oriented languages (e.g., C++).

The key advantage is code reusability. Once the first implementation has been created the next implementation of a similar system is much faster as classes with methods describing part behavior can be reused. This is a major advantage in complex systems such as the Mark-45. The assembly shown in Figure 1 is just one of 15 major assemblies. The six sheets of logic circuits that control the solenoids have not been shown. Each assembly similarly has multiple electrical and hydraulic schematics associated with it. Altogether the Mark-45 has over 23,000 individual parts. Many of the parts in systems such as the Mark-45, Mark-13, and Mark-26 share common components such as solenoids, sensors, and pilot valves. Thus much can be gained if we can reuse objects describing device components or tutor components.

Object classes and methods also facilitate the representation of semantic networks. These networks are used to represent device structure and operation, as described in the next section. There are some subtleties to the implementation that are suppressed in the discussion that follows. These details can be found in Appendix I.

3. The subject matter representation

This section describes the subject matter representation of the device in more detail. The lower hoist domain is used as an example to illustrate what these representations look like.

3.1 Device parts and their relationships

A class hierarchy of part types and the actual parts used in the device is required for the qualitative model. Methods that define the normal and faulted operation of part types are defined for the different classes. These methods can be reused for other assemblies of the same device or for other devices that incorporate similar components. Examples from the lower hoist domain should clarify these points.

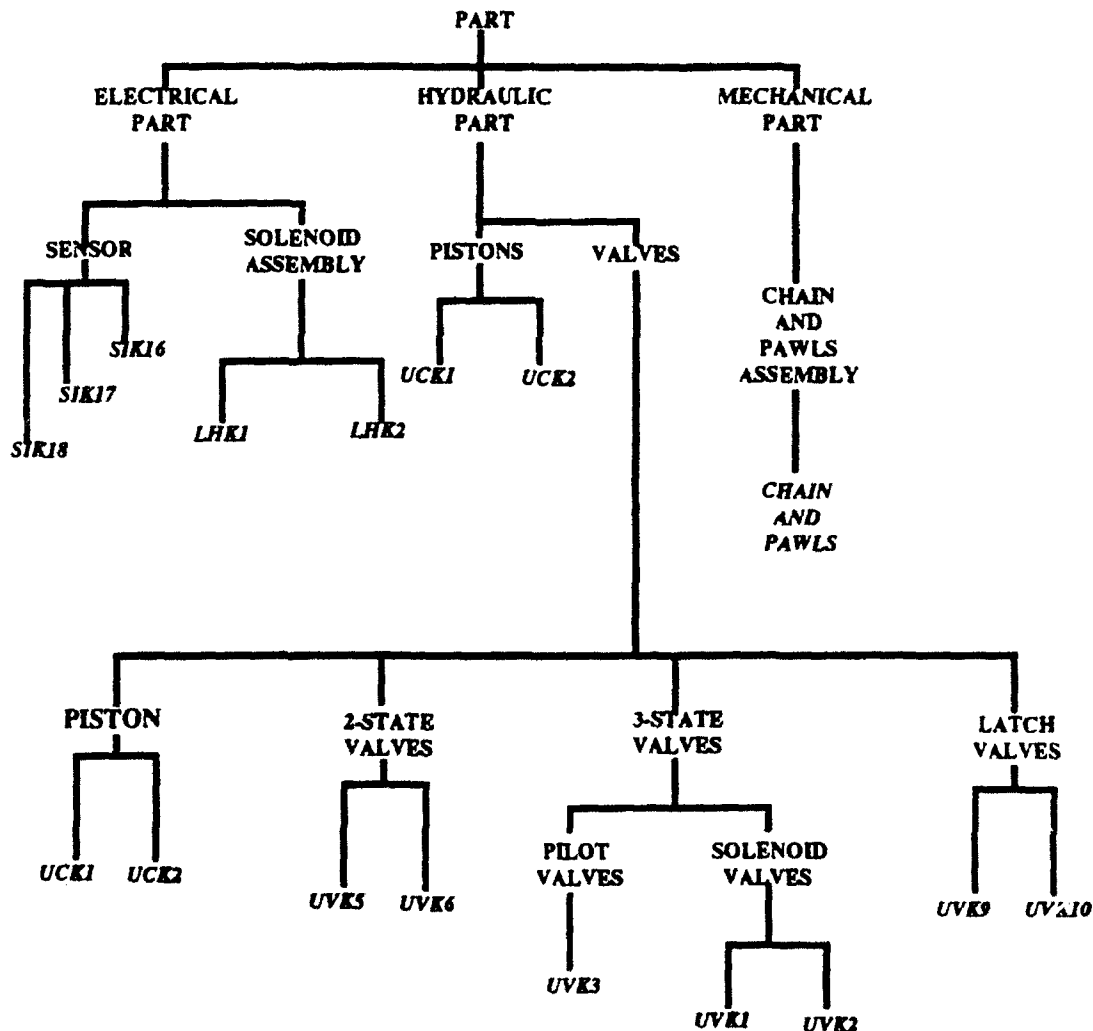


Figure 2. A portion of the class hierarchy of parts in the Lower Hoist Tutor

Figure 2 shows a portion of the class hierarchy of the parts used in the device model of the Lower Hoist Tutor. Methods for simulating the operation of each part type are defined for part classes when they cannot be inherited. For example, a **simulate-operation** method is defined for the class **3-state valves** that would apply to UVK1 and UVK2 if no **simulate-operation** method was defined for the class **solenoid valves**. However since solenoid valves are a special kind of three-state hydraulic valve, one in which control is achieved by electrical means rather than by hydraulic means, a new **simulate-operation** method has been defined for the class **solenoid valves**. This new method overrides the inherited method.

These methods are reusable in the following sense. Most of the component types in the lower hoist assembly are used in other assemblies of the lower hoist. To implement the functionality of another assembly that uses the same parts as the lower hoist only new instances for the parts need be created. The instance descriptions will have new slot values to describe part connections but no new classes need to be defined. It is as if we have created a software library describing the operation of solenoids, sensors, etc. for the lower hoist which applies to all assemblies. Similarly, there are six sheets of schematics for the electrical circuits that control the lower hoist solenoids. They all share common components such as logic gates, buffers, and solenoid drivers. Once the classes for one sheet have been defined they can apply to the remaining five sheets, significantly reducing the amount of work that must be done to implement the whole set.

In addition to methods to simulate part operation, additional methods are defined to reset parts when starting the simulation, to test the values of port outputs and inputs, and to insert faults into parts. These will be discussed more as we talk about the qualitative simulation.

3.2 Device cycle of operation

Typically a device such as the lower hoist can operate in different modes. For example, the lower hoist can operate to load or unload ammunition. The Lower Hoist Tutor currently only models the load cycle of operation but the discussion of how to represent cycles applies to other operating modes also.

Each cycle can be broken down into smaller units called *subcycles*. For example, in the lower hoist load cycle the first subcycle engages a drive coupling that connects two driveshafts in preparation for the second subcycle. In the second subcycle a rack piston moves up, rotates both drive shafts, and causes a chain and pawls assembly to raise

ammunition held between the pawls. The third subcycle drops the hydraulic pressure used in the first subcycle. The fourth subcycle disengages the drive coupling connecting the two drive shafts in preparation for the fifth subcycle. In the fifth subcycle the rack piston retracts. The last subcycle drops the hydraulic pressure used in the fourth subcycle.

These subcycles are all represented as instances of the class **device subcycle**, as shown below:

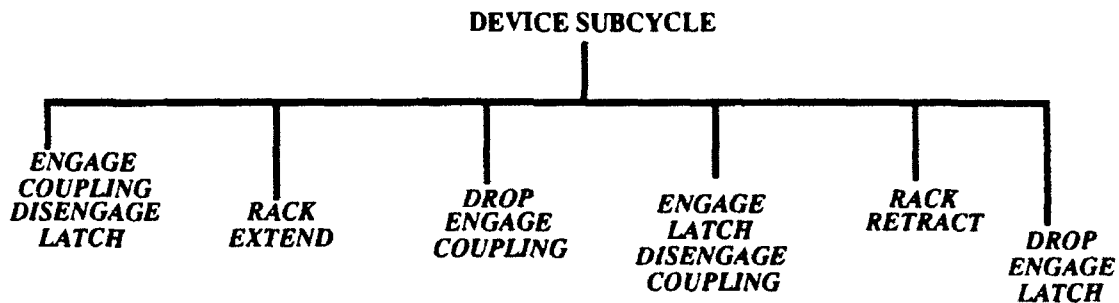


Figure 3. Representation of device subcycles in Lower Hoist Tutor

Each subcycle in turn can be broken down as individual part changes such as a solenoid energizing or a valve moving from one position to another. These part state changes are not represented explicitly as the Lower Hoist Tutor can generate them when necessary by running the qualitative model of the device, discussed in Section 4. Each subcycle begins with some external event—in this case solenoids energizing—and ends when the device reaches a quiescent state. These external events that start a subcycle are called its *initiation conditions* and are stored in a slot of the same name. These events occur only if certain preconditions are true. Typically these preconditions ensure that the device is in a safe condition for the subcycle to start and that previous subcycles have completed successfully, as earlier subcycles typically enable later ones. These preconditions are stored as the **preconditions** slot of a subcycle object. Another slot, called the **postconditions** slot, is used to **test to see** if a subcycle completed successfully.

As an example in the lower hoist domain, a precondition of the second subcycle is that the drive coupling be engaged and the latch retracted. If the drive coupling were not engaged then the rack piston's motion would not be coupled to the chain and pawls assembly. If the latch were still engaged then mechanical damage to the lower hoist would occur. But the subcycle prior to this second subcycle should preclude either possibility. The postconditions for the extend rack subcycle are that the rack is extended, as indicated by a

particular sensor. The subcycle starts when its initiating conditions are true. This happens when the lower coil of the first solenoid assembly is energized and its upper coil is deenergized, while simultaneously the lower coil of the second solenoid assembly is deenergized and its upper coil is energized.

4. The qualitative model of device operation

In this section we describe how both normal and faulted operation are modeled for the device. With this capability the tutor can demonstrate how a device operates with zero, one, or any number of faults inserted. Such a capability, when coupled with the graphic display described in Section 7, allows a student to visualize the operation of the device in a way that is not possible with the static viewgraphs commonly used for this purpose.

4.1 Modeling normal operation

Each part can be thought of as a black box with input ports, output ports, and a current state. We assume the number of states is discrete, but it can be a large number. Typically the part will have two states (e.g., a NAND gate with outputs 1 or 0) or three states (e.g., a pilot valve that is in either the center, right, or left positions). But a part could have additional states describing fault modes.

Associated with each part type are rules describing its normal operation. For the lower hoist there are rules describing how solenoids operate, how pilot valves operate, how a drive coupling operates, etc. These rules describe what the new state of a part will be given its current inputs and current state. For example, a NAND gate with inputs 1 and 1 will have output 0. A piston with PA (high pressure) at the top and TANK (low pressure) at the bottom will shift down, producing a mechanical output (down). The piston may be coupled to a sensor that is energized when its mechanical input is up and deenergized when it is down. The new outputs of a part can be either constant values or functions of the inputs. For example a multiplexor can connect outputs to inputs so that the output value of one port is the same as the input value of the port it is connected to. Writing these kinds of behavior rules for discrete-state parts is fairly straightforward.

Although the rules are associated with part types, specific part connections are associated with part instances. These describe how the input and output ports of each part are interconnected. In the figure below the two output ports of part 1 are connected to the two

input ports of part 2. There are no connections to the inputs of part 1 or the outputs of part 2, so the inputs to part 1 are set externally, by the tutor or student.

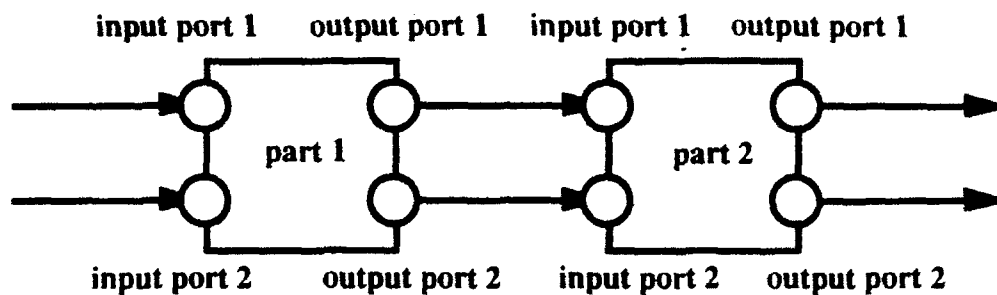


Figure 4. Example of two connected parts showing port connections

Modeling normal operation is straightforward. Given some external inputs to a part the new state and output of the affected part is predicted using the rules of normal operation. Next, those outputs that are different from before are propagated to the input ports of any connected parts. Every time a part has an input value change its new state and outputs are determined. Every time a part's outputs change from previous values they are propagated to all connected parts.

There is no guarantee that this process will terminate if there are cycles in the part connections. For example, consider an inverter connected to itself as shown in Figure 5. If its input is 1 then its output should be 0 and vice versa. But if the output is connected to the input the output cannot be predicted. The process described above will change the output to 1, then to 0, then to 1, and so on, never terminating. But this theoretical concern is not a problem when modeling properly designed physical devices such as the lower hoist.

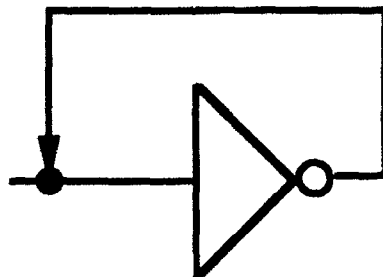


Figure 5. Example where model normal operation will not terminate

4.2 Modeling faulted operation

The algorithm that models normal operation can also model faulted operation. To do this the behavior rules that model how part types normally operate (the *simulate-operation* methods) must be changed. They must also model how parts operate when in various fault modes. For example, NAND gates can stick in a 1 or 0 position, and a three-state valve might not be able to enter the left position because it can no longer engage a slot. The methods describing how logic gates or three-state valves operate could be modified to include these fault modes.

A more economical way of modeling common faults is to use *generic faults* for discrete-state parts. This is the approach used in the Lower Hoist Tutor, which models the following three kinds of generic faults:

1. *sticky-state faults*—a part sticks in a position once it has entered that position.
2. *always-state faults*—a part starts in a position and never moves from that position.
3. *cannot-enter-state faults*—a part cannot enter a particular position.

Some faults produce equivalent behavior for certain parts and states. For example, a *sticky-state fault* where a valve sticks in the right position is equivalent to an *always state fault* if that valve is initially in the right position. But they differ if the valve starts off in the left position. In that case the device will cycle normally up to the point that the valve has entered the right position and would normally enter some other position. The *cannot-enter-state fault* also reduces to one of the other two kinds of faults if there are only two states the part can be in. But if a part can be in three or more states then this kind of fault is different. For example, a three-state pilot valve may not be able to shift left even though it can shift to the center or right position. Note that the *always-state fault* models some common faults for logic circuits, such as stuck-at-1 or stuck-at-0 outputs.

In the Lower Hoist Tutor these generic faults are modeled by altering the behavior rules for the different part types to account for the generic faults. The modifications are similar for all part rules as opposed to modeling specific fault modes for each different fault type. For example, to model a sticky-state fault a behavior rule would first check to see if the current state is the one a part sticks in. If it is the rule would not change the part state or outputs. Otherwise the normal rule of behavior would be followed. For the always-state rule the

modification is similar. However, the part is forced into the frozen state as soon as the fault is inserted—regardless of its current state, and then the outputs are propagated. For the cannot-enter-state fault the normal rules of behavior are ignored only when they specify the new state to be the prohibited one.

The generic fault methods could also have been programmed by providing a **simulate-faults** method that could call the **simulate-operation** method that would model only normal operation. Each part type would also need a **restore** method to restore the last state and outputs. The **simulate-faults** method would just call the **simulate-operation** method if no faults have been inserted into a part. If a sticky-state fault had been inserted the method would do nothing if the part was in its frozen state, otherwise it would call the **simulate-operation** method. For an always-state fault the method operates similarly but the part would be forced into the frozen state as soon as the fault was inserted. For the cannot-enter-state fault the **simulate-faults** method would just call **simulate-operation**. If the new state were the prohibited one then it would call **restore** to undo the effects of **simulate-operation** and it would not propagate the part's outputs. Otherwise the changes would not be undone and the changed outputs would be propagated.

The main advantage of this alternate approach is that the **simulate-faults** method could be defined just *once* for the class **part**. It then applies to *all* parts, regardless of their part types. So the **simulate-operation** methods, which differ for each part type, need not be changed. Furthermore, newly defined part types can also be modeled with faults provided they have a **simulate-operation** and **restore** method. And it would be easy to add new generic kinds of faults, for example, to model broken connections or inverted outputs, that are not currently present in the Lower Hoist Tutor.

5. The fault-space model of troubleshooting

In this section we present a general approach to diagnosis for systems of discrete-state parts that have discrete-valued outputs. Although the approach has been implemented in the Lower Hoist Tutor for only binary-valued outputs the approach can be generalized to systems with discrete-valued outputs. This diagnosis approach tracks faults that are consistent with information gathered and recommends optimal troubleshooting actions. It can also be used to evaluate student actions to detect troubleshooting tests that are suboptimal, irrelevant, or redundant.

Intuitively, the approach depends on generating a table that relates faults to symptoms. Using this table allows faults that are not consistent with the test results to be eliminated. The best test to perform next can be chosen by evaluating the possible consequences and cost of each test.

5.1 Generating a fault-effect table

The fault-effect table is generated by enumerating all possible combinations of fault types, parts, and states that are modeled. Each row of the table is generated by inserting a single fault into the device, cycling the device, and then recording the measurements that result. Not all fault types are distinct so the number of rows can be reduced by only representing unique fault-part-state combinations. These are the rows of the fault-effect table.

For example, assume we are modeling a system with only two gates A and B, two kinds of faults STUCK-AT-1 and STUCK-AT-0, and only three possible test points. Then the fault-effect table would look like this:

Fault	Test point 1	Test point 2	Test point 3
output A stuck at 0			
output A stuck at 1			
output B stuck at 0			
output B stuck at 1			

The cells of the table would be filled in by inserting the faults into the system and recording the values measured at the test points.

The Lower Hoist Tutor models nineteen separate parts. Each part can have any of the three generic kinds of faults affecting any one of its states. If all fault-part-state combinations were unique and each part had three states there would be $3 * 19 * 3$ or 171 rows in the table. But since many parts have only two states and since many fault-part-state combinations are equivalent only 92 rows are actually generated, about half the estimate above.

The columns of the fault-effect table are the tests or observations that can be made. Typically they include the results of pressure gauge tests and logic probes. They can also include other kinds of tests, such as auditory tests (e.g., listening to see if solenoids click), or visual observations (e.g., watching to see if an actuator moves) provided that the expected results can be predicted from the qualitative model.

Consider a small example from the lower hoist domain, where we model only a small number of fault-part-state combinations, as shown in Table 1. Consider the first row for the fault where the pilot valve UVK3 sticks in the right position. With that fault the coupling will engage, the latch will retract, and the rack piston will extend when the lower hoist is cycled. But the subcycle to engage the latch prior to retracting the piston will not complete since that requires that UVK3 shift to the left position and it is now stuck in the right position. That subcycle is initiated when LHK2-LC2 is deenergized, LHK2-LC1 is energized, and LHK1-LC1 is still energized. The solenoids will still be in that state when testing is performed. So the first three columns are OFF, ON, ON as shown. As the drive coupling is still engaged and the latch still retracted, drive coupling disengaged sensor SIK20 is OFF and latch retracted sensor SIK16 is ON, as shown in the last two columns of the first row. The other table entries are derived similarly, by inserting the faults shown, starting the lower hoist cycle, and then testing the solenoid coils and sensors at the point where the cycle stops because of the inserted fault.

Fault	LHK2-LC2	LHK2-LC1	LHK1-LC1	SIK20	SIK16
UVK3 sticks right	OFF	ON	ON	OFF	ON
UCK1 sticks up	OFF	ON	OFF	ON	OFF
UVK10 cannot enter left state	ON	OFF	OFF	OFF	OFF
SIK20 always energized	ON	OFF	OFF	ON	ON

Table 1. Sample fault-effect table for the lower hoist domain

5.2 Using the table to track candidates and choose tests

The fault-effect table can be used to evaluate hypotheses for consistency with measurements. For example, if all we know is that LHK2-LC2 is OFF and LHK2-LC1 is ON then only the first two faults are possible.³ If the student believed the problem was a fault in SIK20 the tutor can explain that different test results would be expected if that were the case. If SIK20 were faulted, then LHK2-LC2 should be energized (ON), but it is not. The Lower Hoist Tutor evaluates student fault hypotheses in this manner.

³Of course this assumes that only the faults that are modeled can occur. As the model will always differ from the physical system in some manner there will always be *some* faults that are not modeled.

The set of fault candidates can also be tracked with this approach. Each *fault candidate* is one of the unique part-fault-state combinations enumerated in the fault-table. It must be consistent with all test results that have been taken. This set can be determined by taking the intersection of all possible faults that are consistent with each test result. Alternatively this set can be initialized to all possible part-fault-state combinations and then successively reduced by removing those combinations that are inconsistent with new test results. The Lower Hoist Tutor uses this incremental approach.

The fault-effect table can also be used to evaluate troubleshooting actions to see if they provide new information. A test that does not provide new information has a value that can be inferred from prior test results. In the example above, with LHK2-LC2 OFF and LHK2-LC1 ON, it would make sense to check any of the other three possible measurements in the last three columns to distinguish between the first or second fault. But if all we knew was that LHK2-LC2 was energized it would not make sense to check either LHK2-LC1 or LHK1-LC1. Their results are the same for all possible faults consistent with LHK2-LC2 ON. Instead, the possible faults shown in the last two rows of Table 1 can be distinguished by testing SIK20 or SIK16. The Lower Hoist Tutor uses its fault-effect table in this way to comment on student troubleshooting actions whenever they are irrelevant or redundant.

For a larger fault-effect table such as that used by the Lower Hoist Tutor there can still be many potential tests to choose from, each of which provides new information. The Lower Hoist Tutor recommends the test that is most likely to split the space of candidate faults. For example, suppose the following parts could be faulted: UVK3, UVK4, UVK9, and UVK10. If any other part had a fault it would not be consistent with our test results. Suppose only SIK20 or SIK18 can be tested and the candidates consistent with the possible test results are shown in the table below:

possible test	possible faults if ON	possible faults if OFF
SIK20	UVK3	UVK4,UVK9,UVK10
SIK18	UVK9,UVK10	UVK3,UVK4

Then the Lower Hoist Tutor will prefer the second test as it is more likely to split the candidate space of faults.

In general, the Lower Hoist Tutor chooses the test to recommend by considering the different possible results for each test. It chooses the test that is most likely to split the candidate space of possibly faulted parts assuming that different possible test results are

equally likely. For the lower hoist domain all test results are binary valued. For each possible result the parts that could still be faulted are determined as described earlier. The closer this number is to one-half the number of fault candidates, which would be a ratio of 0.5, the better. The test to recommend is chosen by computing a penalty score based on this number (difference of this ratio to 0.5), rank ordering the tests, and then choosing the test with least penalty score.

An additional refinement takes into account the time required to perform tests. Cheaper tests that provide *any* information are preferred to more expensive tests, even if the more expensive tests would better split the space of candidate faults. Recall that tests that do not provide any information do not reduce the number of fault candidates, regardless of results, because the test is either irrelevant or the results could be predicted from an earlier test.

The fault-effect table approach that the Lower Hoist Tutor uses has been directly adopted from the approach used in the IMTS [Towne and Munro, 89] system and its PROFILE diagnostic engine [Towne *et al.*, 83]. PROFILE is a more sophisticated tool that uses an information-theoretic approach to selecting tests. That approach takes into account part malfunction probability, unlike the approach presented here. For example, smaller valves are more likely to have faults than larger ones, but the Lower Hoist Tutor does not take this into account. PROFILE also has a more sophisticated approach to weighing the cost of a test and its potential benefits. PROFILE assigns each test a utility measure for the amount of information gained by the test, and then divides this measure by the time required to perform the test. The result is a measure of information gained per unit time.

The Lower Hoist Tutor use an ATMS (assumption-based truth maintenance system) [DeKleer, 86] for caching the fault-effect table. An ATMS stores the assumptions required to make assertions true. The fault-effect table is stored as a disjunction of the faults that produce each possible test result. So, for example, in Table 1, the test result SIK20 ON is justified when either the fault "UCK1 sticks up" or "SIK20 always energized" is present. Thus from the ATMS we can directly retrieve the fault candidates that are consistent with a particular test result. Without the ATMS we would have to search the fault-effect table, so the ATMS serves as a cache. Fault candidates consistent with multiple test results are the intersection of those consistent with each individual test result. The tutor can evaluate how well a test splits the fault candidate space by comparing the fault candidates that remain for the different possible outcomes of a test (e.g., SIK20 ON and SIK20 OFF), assuming the

outcomes are equally likely. All this can be done without an ATMS; the ATMS just makes it faster.

6. Explanations and assessments from qualitative models

This section explains how textual explanations and test questions to assess student understanding can be *generated* from device descriptions, rather than being pre-stored. This approach has not yet been fully implemented in the Lower Hoist Tutor. Currently, the tutor can only generate simple explanations, such as "UVK3 shifts right" to explain part state changes. No test questions are currently generated although the tutor can generate troubleshooting cases by randomly choosing fault-part-state combinations for student practice.

6.1 Generating explanations of device operation

Explanations of part behavior could be defined as methods for each part type. Instead of producing an explanation like "LHK2 shifts right" an explanation specific to the **solenoid-assembly** part type class could be generated by a method for that class. This explanation might be:

"The solenoid assembly LHK2 shifts to the right position because its bottom coil (LHK2-LC2) is energized while its upper coil (LHK2-LC1) is deenergized. This change causes PA to be ported out of the left output port of LHK2 and into the left input port of the pilot valve UVK3."

These part type explanations could generate different levels of explanation. Section 8 discusses how an assessment of the student's understanding of each part type can be associated with part classes. Using this assessment, which is part of the student model, the tutor can generate detailed explanations for part types that the student has not yet demonstrated an understanding of. Less detailed explanations could be generated for those part types which the student appears to understand. Explanations would become more concise as the student learned more.

6.2 Generating explanations of troubleshooting actions

The Lower Hoist Tutor does not currently provide detailed explanations for how troubleshooting actions are chosen or how student hypotheses are evaluated. It could use a truth maintenance system (TMS) for the latter. For example, a student fault hypothesis (e.g., UVK3 sticks left) could be inserted into the qualitative model and then the device

cycled to determine the effects of the fault. If the results of the fault contradict the test results then the contradiction could explain why the student's hypothesis is inconsistent. Here is an example of the kind of explanation that could be generated:

"The fault could not lie in UVK3. If there were some problem with UVK3 then either the subcycle to engage the coupling and disengage the latch, or the subcycle to engage the latch and disengage the coupling would not complete. But these cycles have been completed since the lower hoist is currently in the subcycle to retract the rack piston.

We know that the lower hoist is in the subcycle to retract the rack piston since LHK1-LC1 is deenergized and LHK2-LC1 is energized."

Better explanations for choosing tests could appear in this form:

"At this point we know the fault first appears in the engage coupling and disengage latch subcycle. The only parts that change state in this subcycle are LHK2, UVK1, UVK3, UVK4, SIK20, UVK10, UCK2, and UVK9. Testing either SIK20 or SIK16 will help determine where the subcycle failed."

or in response to the choice of an irrelevant troubleshooting test:

"SIK17 indicates the state of the lower hoist rack piston. Since we have determined that the lower hoist stopped in the first subcycle, prior to extending the rack piston, this test is not relevant.

We know that the lower hoist is in the subcycle to engage the drive coupling and disengage the latch since LHK2-LC2 is energized, LHK2-LC1 is deenergized, and LHK1-LC1 is deenergized."

These kind of explanations are not currently generated by the Lower Hoist Tutor.

6.3 Generating test questions and cases for assessment

The skill of troubleshooting can be broken down into component skills, such as selecting tests and interpreting results. A highly simplified decomposition is shown in Figure 10, a more thorough decomposition is deferred until Section 8. The objects shown in the

semantic network of Figure 10 are implemented as objects of class **skill**. Methods could be defined both to carry out each skill for demonstration purposes, and to assess student capabilities for each skill.

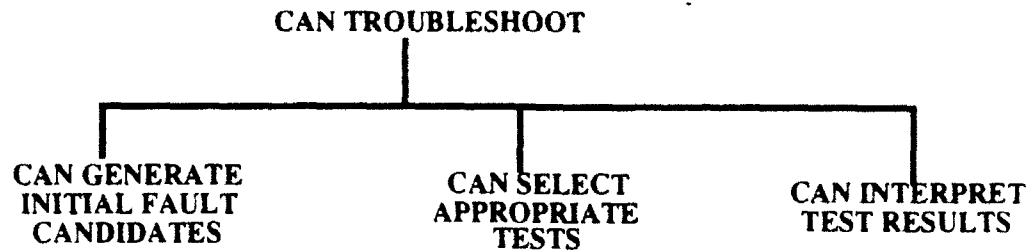


Figure 6. A simple decomposition of troubleshooting skills

For example, to assess the student's ability to generate the initial set of fault candidates the tutor would choose a fault and then present the symptoms to the student. Then the student would have to select each part that could be faulted by pointing to it. Using the fault effect table the tutor can determine if the selected parts are consistent and if the student has left out any parts that could be faulted.

Similarly, the tutor could ask the student what test to perform next for various situations where the set of possibly faulted parts is shown highlighted, and a list of previous test results is displayed.

Finally, the student could be told the results of a test and asked to interpret the results by indicating which parts could be faulted or not. Those parts that are initially part of the candidate set, before the test, would be highlighted. The student would then click on those parts that have been ruled out by the test.

7. A user interface that shows device parts and animation

The most important part of the Lower Hoist Tutor's user interface is its graphical device simulation. The graphical device simulation shows a device assembly in color. It allows the student to select parts with the mouse. The tutor can also highlight parts when discussing them. Most important of all the graphical device simulation can illustrate how parts change states through animation. The animation shows the sequence of part state changes directly, emphasizing the causality of the changes.

7.1 Capturing schematics and adding animation and interactive capabilities

Figure 7 shows an overview of the process of building a graphical device simulation. Schematics of the device assemblies are scanned in. The same assembly may appear in multiple schematics if they show how it operates in different modes, or in different stages (i.e., subcycles) of the same mode of operation. The scanned in schematics are cleaned up in a paint program that allows *bitmaps* (i.e., collections of pixels) to be edited. The cleanup is required to remove noise from the images or introduced by the scanning process.

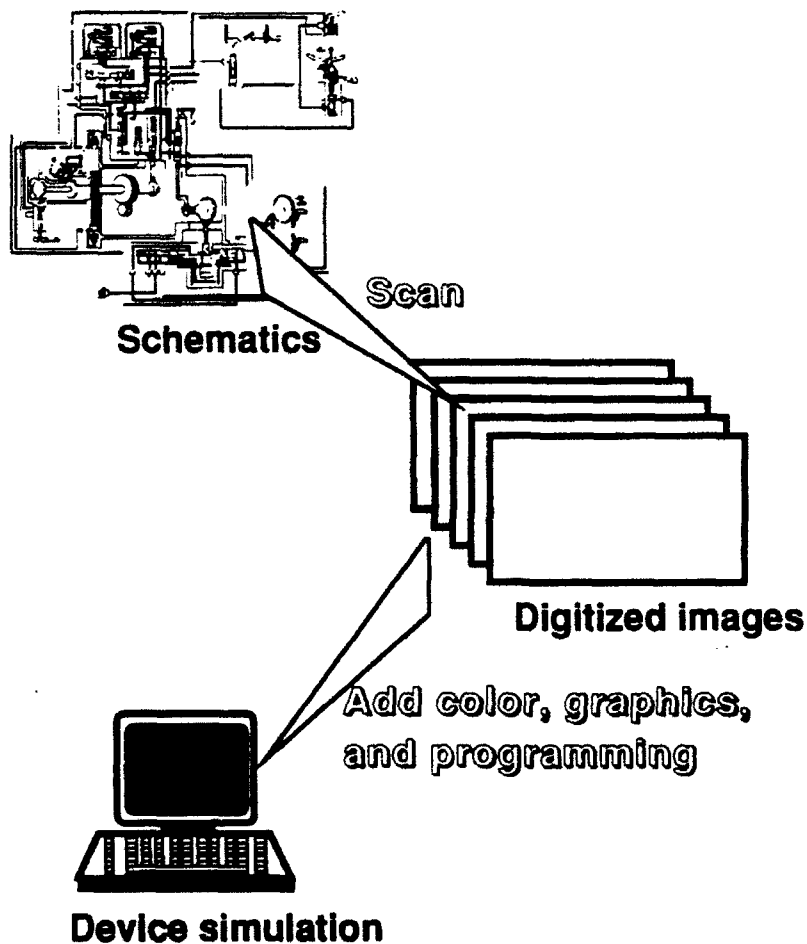


Figure 7. Overview of approach to building graphical device simulation

Animation can be introduced in one of two ways. The first, *flip-card animation*, shows parts only in different distinct states. For instance a pilot valve could appear in only three different states such as left, center, and right. A different bitmap is used for each state and only one is visible at a time. When the part changes state the bitmap for the old state is hidden at the same time the bitmap for the new state is revealed.

Hypercard and hypercard-based tools such as SUPERCARD, SPINNAKER PLUS, and TOOLBOOK are ideal for programming this kind of functionality. These tools provide an environment for the rapid prototyping of graphics applications. Visual effects can be introduced to fade gradually from one image to another and provide an illusion of part movement. Sound effects can also be added, such as metallic sounds as parts move and engage, and whooshing sounds as pipes fill or empty.

The other approach to animation is to provide a single graphic for a part and then change the position of the graphic to provide a continuous animation of the part moving from one state to another. In that case the part is visible all the time and only its location changes. In either approach all parts move with respect to a static background.

Schematics need not always be scanned in but this approach is often faster and more suitable for curriculum integration than drawing. It may be too difficult or time-consuming to draw complex mechanical and hydraulic assemblies where parts appear in multiple states. For electrical schematics it may be acceptable to just cut and paste images from a graphics library (e.g., of logic gates). In this case drawing may be acceptable. But even then subject matter experts frequently prefer that the tutor's graphics match exactly the graphics the students already work with. In this case scanned-in schematics are the best approach. Of course if schematics are already online for some other use, such as computer-aided design, then both scanning and drawing are unnecessary.

The Hypercard-based tools simplify adding color and interactivity to the schematics. Thin colored rectangles can be used to simulate hydraulic fluids in pipes. Their color can be set under program control. Following standard conventions red can be used to show PA (pressure applied, or high pressure hydraulic fluid) and yellow can be used to show TANK (low pressure hydraulic fluid). Programs can be written to group related rectangles together, to act as one pipe, and then to change the color of the simulated pipe as the pipe fills or empties. The Hypercard-based tools provide programming languages such as HyperTalk that simplify this task.

Mouse sensitive regions called *buttons* can be placed over the parts. These can be invisible to the student. When the student selects the button any arbitrary code can be called in the graphics programming language. For example, text describing the part's role can be retrieved and displayed. The buttons can also be used to answer tutor questions since the student can just select parts by pointing and clicking.

7.2 The graphics interface in the Lower Hoist Tutor

The Lower Hoist Tutor is built on a scanned-in schematic for the Mark-45 lower hoist assembly, following the process shown in Figure 7. A text window was added to upper left hand corner of the schematic for student text input and tutor text output. Hypertext is implemented in this window so that students can click on part names to retrieve part descriptions. These descriptions may refer to other parts that can also be clicked on to bring up their information. Parts can also be selected by mouse to access these descriptions.

Animation was added by writing routines that show parts in one state at a time. A routine to reset all parts to initial states at the start of a cycle was also added. The qualitative model of the device determines the sequence of part state changes. This model can be either implemented in the same environment or a separate environment. In the Lower Hoist Tutor the graphics environment is SuperCard and the symbolics programming environment is COMMON LISP and CLOS (the COMMON LISP OBJECT SYSTEM). SuperCard would not be appropriate for the qualitative model because of its slower speed and lack of support for user-defined object classes.

The MacIvory implementation of the Lower Hoist Tutor is shown in Figure 8. The MacIvory is a Macintosh with an Ivory co-processor that supports the LISP programming language and environment, and CLOS extensions for object-oriented programming. The Lower Hoist Tutor uses the Macintosh for all graphics. Graphics routines for flip-card animation are programmed in SuperTalk. The qualitative model of the device is programmed in LISP and CLOS. It executes on the Ivory processor.

Communication between SUPERCARD and LISP is achieved through external commands and external function calls (XCMDs and XFCNs). These are called from SUPERCARD and pass LISP forms to be evaluated to the IVORY processor. Results are received as strings by SUPERCARD.

The buttons on the bottom and right of the graphics interface shown in Figure 8 are listed below, grouped according to functionality:

1. control of the simulation—

- a. initialize—resets the qualitative model and graphics.*
- b. single step—shows the results of the next part state change*

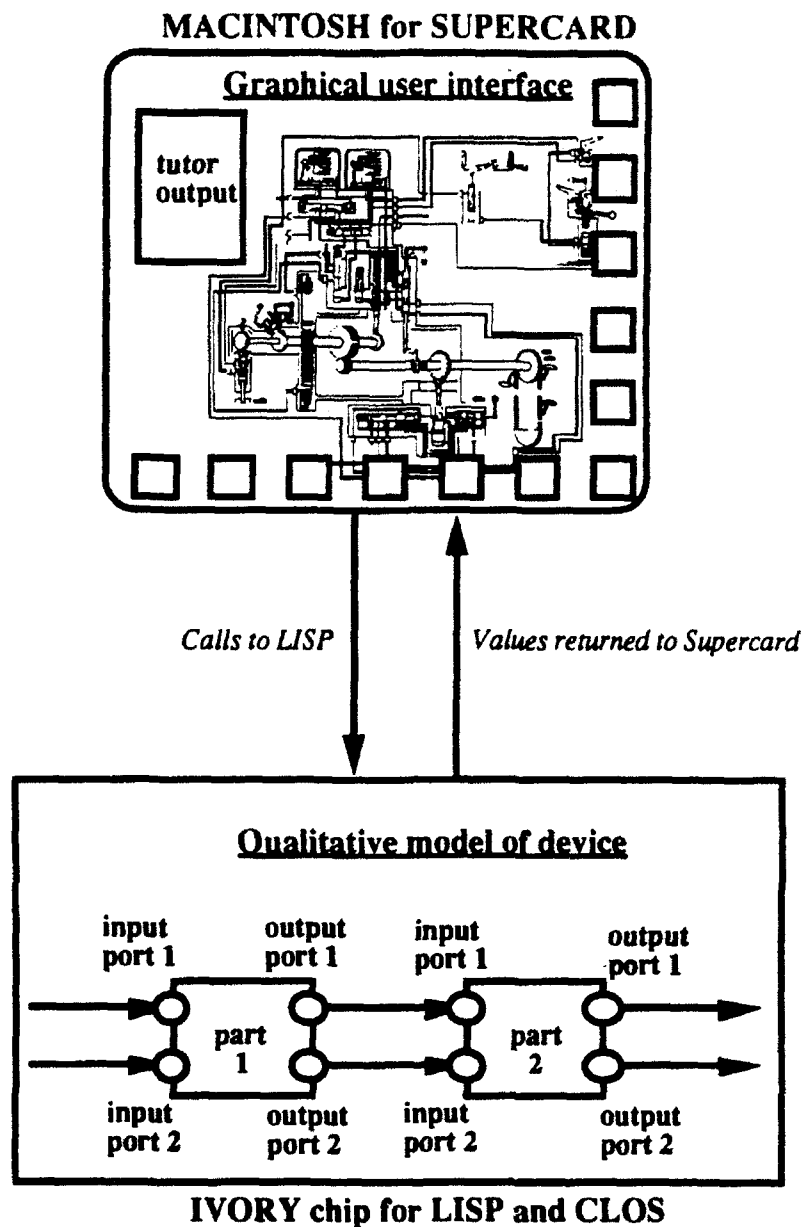


Figure 8. Implementation of Lower Hoist Tutor on MacIvory platform

- c. subcycle*—shows the remaining steps in the current subcycle.
- d. cycle*—completes the entire cycle.
- e. step text on/off*—controls whether or not textual explanations are generated to explain part state changes.
- f. skip steps on/off*—controls whether or not intermediate steps are shown for subcycles and cycles.

2. *simulation of troubleshooting actions*—

- a. *measure*—allows a port value to be measured
- b. *replace*—removes all faults, if any, from a part

3. *fault insertion*—

- a. *insert fault*—lets the student insert a fault in a part. The student selects the part, the kind of fault, and the state affected.
- b. *place random fault*—lets the tutor insert a fault by selecting a part, fault, and state randomly. The student does not know what fault was inserted.

4. *requests for help*—

- a. *suggest action*—the tutor explains what should be done next
- b. *list candidates*—the tutor highlights just those parts that could be faulted given the test results gathered so far.
- c. *reveal faults*—the tutor explains any faults previously inserted
- d. *explain fault consequences*—the tutor explains how the inserted fault or faults interfered with the normal lower hoist cycle.

These buttons invoke SuperTalk commands that send LISP forms to the qualitative model. The qualitative device model is updated and a description of the changes that occurred is returned. These are a series of SuperTalk commands such as:

```
shiftPart UVK3,right  
presentText "UVK3 shifts right."  
showPipe "UVK3 left output to UVK4 top input",PA
```

These commands describe part state changes to show, parts to be highlighted, and text to be placed in the tutor output window. These commands are executed one at a time to complete the operation of the button pushed.

8. Interpreting student performance

Now we begin discussion of extensions to the Lower Hoist Tutor. The extensions for planning and student modeling have been only partially implemented. Those for knowledge acquisition have not been started. With these extensions the tutor would follow its own goals, create plans, and modify these plans according to student progress. First we discuss the student model, then the planner. The student model can provide additional functionality

even without the planner. It could be used for automated problem selection or generation for example.

8.1 A generic representation of target skills

The skill to be acquired is broken down into subskills. One possible decomposition is shown in Figure 9. (Dashed lines are used to represent decomposition links and to emphasize that the graph shown is *not* a CLOS class hierarchy.) In this decomposition the key subskills are:

1. Understanding how the device operates
2. Identifying fault symptoms as deviations from normal behavior
3. Generating fault candidates that explain the symptoms
4. Choosing the most appropriate troubleshooting action given a set of fault candidates and a the history of previous test results
5. Interpreting test results to reduce the set of possibly faulted parts

The tutor's own troubleshooting capabilities would also need to be decomposed in the same way. Then it could demonstrate and generate test questions for each subskill. Each subskill would be an object of class **skill** with **demonstrate** and **test** methods defined for that class. This ability to assess each subskill would allow more accurate identification of problems the student might have.

The terminal nodes in Figure 9 represent skills that apply to individual domain objects. For example, the skill **can explain part roles** applies to all objects of class **part** and can be applied to any part. The skill **can predict changes to all parts in each subcycle** applies both to parts and subcycles. When assessing this skill for individual parts the tutor asks the student how the part changes state in each of the subcycles. When assessing this skill for individual subcycles it asks the student to indicate the sequence of part changes that occur in a subcycle. Some skills, such as **can interpret result to eliminate hypotheses**, apply to test results or troubleshooting tests.

8.2 Representing skill acquisition and tutor uncertainty

The degree to which skills have been acquired is represented in the student model. More precisely, the student model represents the *tutor's beliefs* about the student's skills. For instance, the tutor may not believe that the student has a skill that he has just not yet demonstrated.

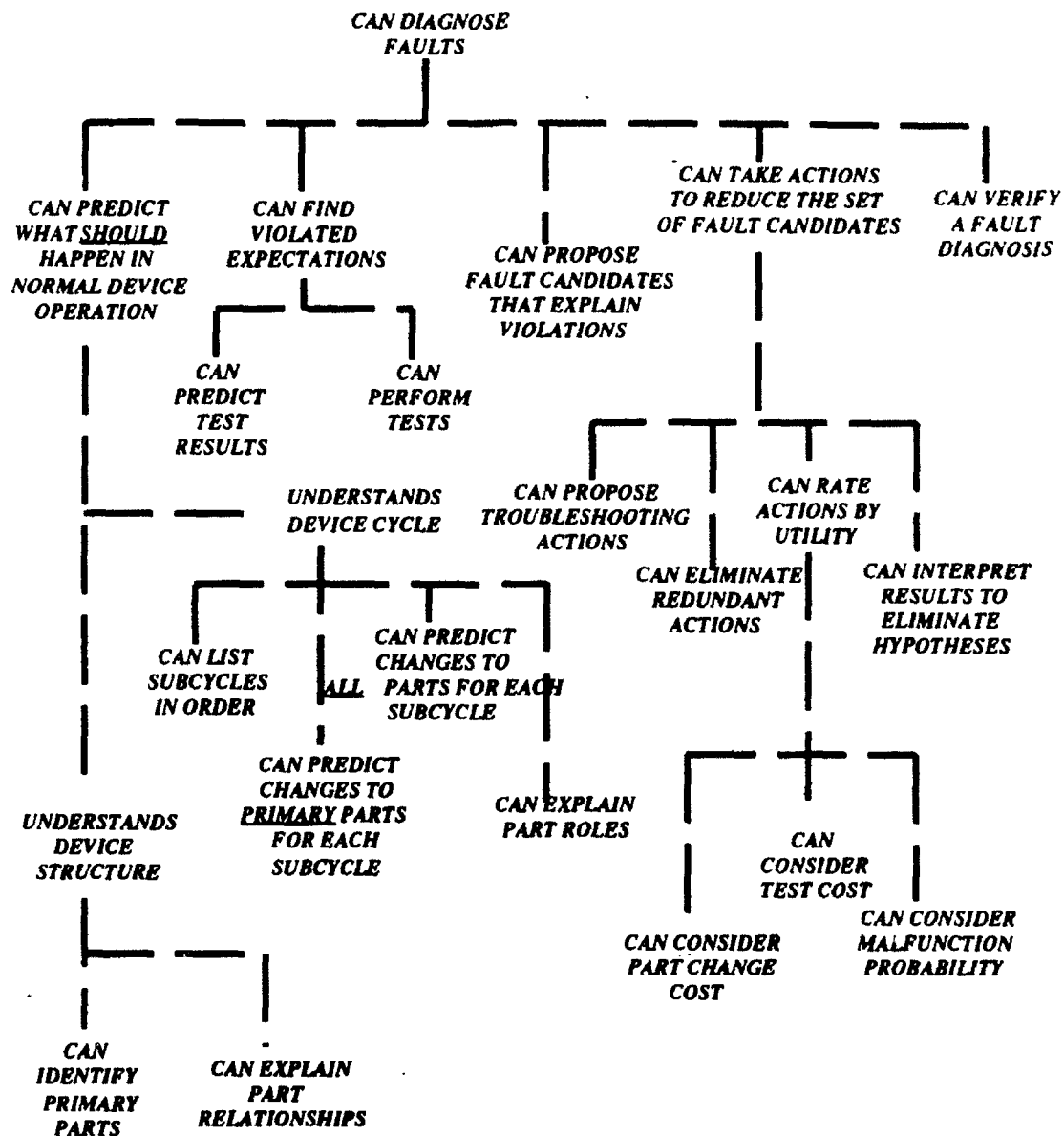


Figure 9. Decomposition of troubleshooting into subskills

A non-numeric representation of uncertainty is used to avoid the arbitrary use of numbers for the relative strength of one belief compared to another. Some problems with purely numeric representations are suggested by showing the kinds of questions that are difficult to address with them:

1. *How to select numbers*—What weight should incorrect answers to multiple-choice questions receive? True-false questions? Questions answered by pointing to parts? Correct answers compared to incorrect answers in general?

2. *How to interpret the numbers*—If the weight for a prerequisite falls below 5, assuming a scale from 1 to 10, should it be reviewed?

3. *How to debug incorrect results*—if a belief has certainty 3 on the same 10-point scale but the instructor believes that it should be much higher, how was that number derived?

A more detailed discussion of different numeric representations of uncertainty and problems associated with them is presented in [Murray, 91]. A more complete description of the non-numeric endorsement-based approach to student modeling, is also provided. We provide only a brief overview here.

This non-numeric approach uses pro and con arguments, called *endorsements*, to represent arguments for and against the tutor's beliefs in particular student capabilities. The advantage of this approach is that all conflicting arguments are retained, rather than being coalesced into one number, and the tutor can take into account the context of a decision when weighing different kinds of evidence.

Basic student performance data is called *assessment data* or *assessments*. Every piece of assessment data is interpreted as an argument for or against the student having some capability, either some general subskill, or a subskill applied to a specific domain object. For example, a T/F question answered incorrectly could be an argument against the student's ability to predict the operation of one particular hydraulic valve. Another multiple choice question about selecting troubleshooting tests, answered correctly, could be a pro argument for that troubleshooting skill in general. Each assessment is placed in a category called its *endorsement reliability class*.

The comparison method described in [Murray, 91] uses a table that places the endorsement reliability classes in a total ordering. Table 2 shows an example of one such ordering. Conflicting arguments are resolved by a lexicographic comparison of pro and con arguments that uses the table. Each belief is assigned a label BELIEVED-FALSE, BELIEVED-TRUE, UNCERTAIN, or UNKNOWN (i.e., no data) based on the comparison. In addition to the label the planner can examine the pro and con arguments responsible for a particular belief.

Class	Symbol	Description
Data trends	TR	Consistent trends in student performance
Negative student self-assessment	ST-	The student says he does not know something
Propagated disbelief	PR-	Argue that skill x cannot be known for class y as it is not known for class (or instance) z and y includes z
Tutor presentation	TU+	Argue that skill is known as tutor has covered it
Label trends	LT	Assign class X the same label as most of its children
Positive student self-assessment	ST+	The student says he knows something
Short-answer	S/A	The student answers a single short-answer question
Multiple-choice	M-C	The student answers a single multiple-choice question
True-false	T/F	The student answers a single true or false question
Inherited belief	IB+	Argue that class (or instance) y is known as its superior class x is known
Default belief	D	Default belief

Table 2. One set of endorsement reliability classes

Now we consider how conflicting arguments are compared in more detail. There is a default lexicographic comparison and a special case mechanism for overruling it. The default lexicographic comparison first sorts all pro and con arguments for a belief according to their endorsement reliability classes. Then it compares pairs of pro and con arguments, starting with the most compelling arguments for each. When each pair of arguments belongs to the same evidence reliability class then the next pair is considered. When there is a more reliable pro or con argument in a pair, then the belief being considered is labeled **BELIEVED-TRUE** or **BELIEVED-FALSE** (respectively) and this label is annotated with the endorsement reliability class of the winning argument. This annotation can be used as a measure of the strength of belief. Finally, if all pairs are balanced the label received is either **UNCERTAIN**, if there was at least one pair examined, or **UNKNOWN** if there was no data to examine.

In some cases it may be desirable to override this default comparison with special case rules. For example, the default comparison labels **BELIEVED-TRUE** a belief supported by a single multiple choice question answered correctly even if there are two true-false questions answered incorrectly, assuming the reliability classes of Table 2. A special purpose rule could instead label the belief **UNCERTAIN** or **BELIEVED-FALSE**.

In contrast to [Murray, 91], the ESM proposed here augments the lexicographic comparison process with rules that specify special exceptions to the lexicographic comparison process. Each rule has the following format:

(rule <name> <condition> <label>)

where the condition is any predicate and the label is any expression that evaluates to BELIEVED-TRUE, BELIEVED-FALSE, or UNCERTAIN. Each rule is tried in order and the first rule to assign a label to a belief is used. If none assigns a label then the lexicographic comparison routine is used as a default. The rule knowledge base initially starts with just that one rule:

(rule default-lexicographic-comparison T (lex-compare pro con))

where pro and con refer to the pro and con arguments and lex-compare is the lexicographic comparison routine.

Consider an example where we override default lexicographic comparison. Assume Table 2 provides the ordering of endorsement reliability classes and there are only two pieces of assessment data to interpret. The most recent is a true-false question answered incorrectly, thus a con argument of class T/F. The second assessment is a multiple-choice question answered correctly at some previous time. Thus it is a pro argument of class M-C. As it is higher in the table it is a more compelling argument and will be believed if the default rule is reached. But if we add the rule below:

(rule prefer-more-recent

(more-recent current-arg (predessor current-arg))

(if (eq (label current-arg) 'PRO)

'BELIEVED-TRUE

'BELIEVED-FALSE))

then the student model assigns the belief label according to the value of the most recent argument. As in this example it does not matter if there is an earlier argument, of higher endorsement reliability class, that opposes it. The rule's condition applies to the example above and its action is to assign a BELIEVED-FALSE label to the belief as the most recent argument has a CON label.

These rules are important in two respects. First, they add increased flexibility to the ESM beyond that originally discussed in [Murray, 91]. Secondly, they provide the basis for a

knowledge acquisition tool that tracks rules that lead to faulty conclusions. The tracking depends on explicit dependency links from rules to conclusions made with the rules.

The rule language presented above is very general. It requires knowledge of LISP and the accessor functions and global variables defined for the ESM. The ability to write these special-case rules increase the utility of the ESM. Eventually a more restricted data interpretation language might be possible that could be used directly by subject matter experts.

8.3 Representing generalized student capabilities

Another advantage of the endorsement-based student model (ESM) is its representation of the degree to which skills have been generalized. For example, consider the part hierarchy shown in Figure 10. Class instances are shown in *italics*, pointed to by dashed arrows, the rest are part classes.

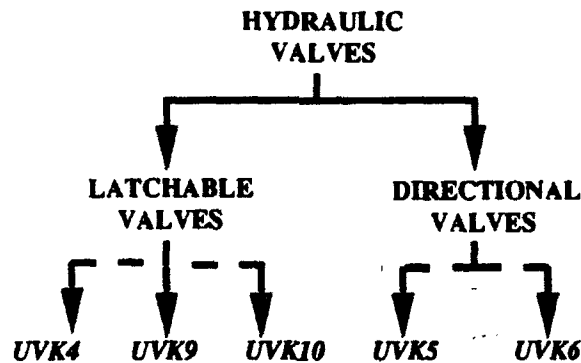


Figure 10. Sample part hierarchy

There can be many different skills that can be applied to each of these parts (e.g., predicting operation, listing possible faults, predicting inputs given outputs, etc.) and the ESM can represent the tutor's beliefs about the student's capabilities for each of these. Beliefs about superordinate nodes are beliefs about the student's capabilities with respect to one particular skill that can be applied to any of the subordinate terminal nodes. For example, if the ESM indicates that the **can predict operation** skill is believed true for the class latchable valves then the tutor believes the student can predict the operation of any of the parts UVK4, UVK9, or UVK10.

The ESM also infers new endorsements from a series of existing primitive assessments. These new endorsements are called *inferred endorsements*. There are two kinds of inferred endorsements: data trends and propagated endorsements. A data trend is inferred when

multiple data items support, or argue against, a skill for either a single object or class of objects. A propagated endorsement reflects inheritance semantics. For example, an argument that the student does not know the possible faults for parts of type directional valves is also an argument that he does not the possible faults for a particular directional valve, such as UVK5. These kind of inferred endorsements are discussed in more detail in [Murray, 91].

8.4 Using endorsements to make decisions

The planner can examine a belief's label and any part of its justification. The justification includes both the rule that assigned the label and the endorsements that argue for and against the belief.

This additional information allows a planner-controlled tutor to make context-sensitive decisions. For example, when reviewing previously learned information the tutor can accept the student's belief that he understands a topic or can perform a skill. This same evidence (student self-assessment) can be considered insufficient when teaching the same material for the very first time.

The ESM can be used in a non-planning tutor also. It can be used to select problems in areas where the tutor believes the student needs practice or is uncertain of the student's capabilities. Or it can be used to drive problem selection to precisely gauge a student's troubleshooting capabilities if the tutoring system was instead used solely as an assessment tool..

This completes discussion of the endorsement-based student model. Additional discussion of the implementation of the ESM using object classes and methods appears in Appendix II.

9. Controlling the tutor with a dynamic planner

This section describes how the Lower Hoist Tutor can be changed from a reactive learning environment to a autonomous goal-following tutor that generates and carries out its own plans. The enhanced tutor can still react flexibly to student requests and does not sacrifice the ability to provide opportunistic instruction or to let the student explore or practice with the device simulation.

9.1 The plan representation

The instructional plan for the Lower Hoist Tutor represents a single lesson. An earlier version of the Lower Hoist Tutor, implemented in conjunction with the Blackboard Instructional Planner, planned a sequence of lessons [Murray, 90b]. The approach of planning one lesson at a time is easier to implement. It should strike a better balance between not planning at all, like most intelligent tutoring systems, and planning an entire curriculum, like the Blackboard Instructional Planner.

The plan representation is a non-cyclic graph with three kinds of nodes: *objectives*, *activities*, and *procedures*. These are connected by two kinds of links: *achieves* links—indicating that one plan element is used to achieve a higher-level plan element—and *precedes* links—indicating that plan elements of the same type are placed in an precedence ordering.

Objectives, activities, and procedures are the same plan elements used in the Blackboard Instructional Planner and described in [Murray, 90b]. They are briefly reviewed here. *Objectives* represent instructional goals and provide a goal decomposition. *Activities* represent pedagogical activities such as introducing a new topic, allowing practice, or performing assessment. However, each activity could be performed in several different ways. So the explanation of a topic could be done with a demonstration, or textually, or through examples. Practice could be done with a set of stored problems, with generated problems, or by allowing the student to use the device simulation however he wishes. Each different way of performing an activity is a *procedure*. Procedures are executable routines that are different ways of carrying out activities. They can have parameters that allow them to be adapted to different tutorial situations. For example, a procedure to present troubleshooting cases could receive parameters specifying the level of difficulty of the cases and the kind of faults to present. Procedures also have a limited form of interruptibility, allowing students to ask questions or make requests between different parts of a procedure called *procedure steps*. These are logical divisions in a procedure. For example, each question asked in a questionnaire could be considered a procedure step.

Figure 11 shows a simple example of the plan representation. Actual plans would be more complex. The shaded circles are instructional objectives. The squares represent instructional activities. Procedures are shown as diamond-shaped boxes in the sample plan below. X-ed out circles indicate goals that are believed to be achieved already, either as the

result of tutor instruction or because the student has learned the information prior to using the tutor. The student model determines when goals are achieved.

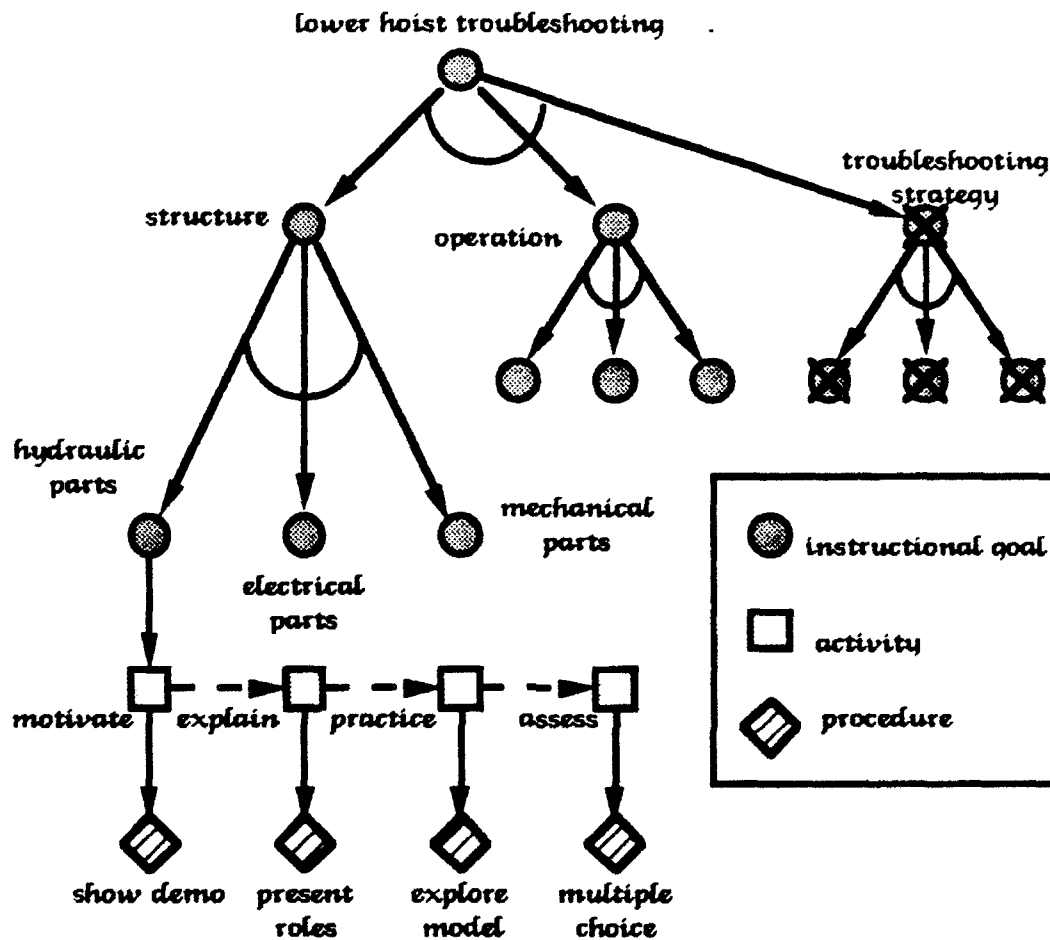


Figure 11. Sample lesson plan showing plan representation

The downward pointing arrows are *achieves* links. The arrows with arcs connecting them are *AND-achieves* links. This means that in order to achieve the higher level goal *all* the lower level goals must be achieved. The downward pointing arrows without arcs are *refines* links. This means that the activity and procedures selected achieve the goal. The dashed arrows indicate a sequence of steps to be performed.

Each plan element is *justified* from other plan elements or from the student model. Procedures are justified in terms of their activities. Activities are justified by the terminal goal they achieve. All terminal goals are labeled as either achieved or unachieved and these labels are justified by beliefs in the student model. Nonterminal goals are in turn justified from their constituent goals.

The planner is a top-down refinement planner with no backtracking. Plan critics are used to reorder parts of the plan and to add new plan elements that improve discourse quality. Plan critics also watch for problems with the plan. When a critic notices a problem parts of the plan may be replaced and attempted again. The next sections explain these processes of plan generation and refinement in more detail.

9.2 Plan generation

The planner is an incremental lesson planner that defers complete plan elaboration. The planner is *dynamic* as it responds to changes in the student model, to student questions and requests, and to changes in time remaining. It plans in the sense of choosing an intended sequence of goals and activities to carry them out. But unlike more sophisticated planners such as SIPE [Wilkins, 88], it does not perform resource allocation, develop nonlinear plans, deduce consequences of actions, or work in a goal-directed manner backwards from desired situations to determine action sequences. Instead the planner is intended only for intelligent tutoring systems where the most important resource is time and where there is some leeway in the length of a lesson, so dealing with time is not a primary issue. The planner proposed here is a simplification of the planner implemented in [Murray, 90b].

For a particular domain the instructor has pre-stored an *instructional goal tree* for the planner's use. This goal tree is a graph of prerequisite skills, that also imposes a sequence on skill acquisition. The goal tree need not fully specify this order, for example, it may only specify that the student learns device structure before learning how the device operates. The goal tree is not a complete plan as it does not specify instructional activities or procedures appropriate for a student. It also may include objectives that a particular student may have already achieved. No activities are planned for such objectives when the plan is customized to the student.

In addition to the goal tree the instructor must provide two kinds of plan libraries. The first is called an *activities library*. Recall that instructional activities are abstractions of instructional actions that assist in achieving instructional goals. They are abstractions because they do not commit to particular procedures to realize the activities, and typically there are multiple procedures suitable for each activity. The second kind of plan library that must be provided is a *procedures library*. This is a library of the procedures (executable routines) that can be used for activities. An example of an activity is ASSESS-SKILL. Procedures for that skill could include MULTIPLE-CHOICE-TEST or SHORT-ANSWER-TEST.

A key difference from the earlier planner and a further simplification lies in the way procedure parameters are initialized and adapted. Parameters in this simplified planner are initialized, monitored, and adapted by each instructional procedure, rather than the planner. This offloads some work from the planner and allows each procedure to apply specialized knowledge of its task for these purposes.

Procedures are still decomposed into *procedure steps* that provide limited interruptibility. Each procedure can be interrupted by questions or requests between procedure steps. The procedure can save state and then either resume where it was interrupted or be reset by the planner.

Planning primarily consists of selecting *activity plans* from the activities library and instructional procedures from the procedures library. An activity plan is a sequence of activities such as

1. motivate skill—explain how this skill fits in with other skills
2. demonstrate—show skill to student
3. explain skill—explain steps in procedure or component skills
4. practice skill—let student practice the skill
5. assess skill—see to what extent steps 1 - 4 succeeded in teaching the skill

Although a procedure typically consists of multiple procedure steps those steps are part of the procedure, i.e., there are no procedure plans. The procedures library only specifies individual procedures that can be used for individual instructional activities, such as any one of the steps above.

The first step in the plan generation and delivery process is to copy the instructional goal tree for the skill to be taught. Each skill that is believed known is marked as satisfied. For each terminal instructional objective a single activity plan is selected. Then for each activity in an activity plan a procedure is selected when that activity is ready to be executed. The planner is incremental because procedures are not selected *until* an activity is the next to be executed. A global switch could also allow the planner to be incremental at the activity level. In that case an activity plan would not be selected until an unachieved instructional objective was next in line to be achieved.

Thus plan generation consists largely of multiple selections. For each goal one of the fixed number of activity plans must be selected, taking into account the tutorial situation. For

each activity, one of the fixed number of procedures must be selected, taking into account the tutorial situation. Each choice can be viewed as classifying planning situations coupled with tutorial situations into one of several classes, where the classes correspond to the elements of a plan library. It is this repeated use of classification and selection that suggests that concept acquisition algorithms from machine learning may be applicable to knowledge acquisition, as discussed in Section 10.

Plan critics and special-purpose rules for making the selections discussed above are also part of the plan generation process. These refinements to the planning process will be discussed shortly in Section 9.4

9.3 Plan revision

Plan delivery consists of executing procedures that are already planned, interleaved with the selection of activity plans and procedures required to develop the plan sufficiently to continue. Every element of the plan is justified in terms of beliefs about the student and beliefs about the planning process itself. When new assessments are interpreted by the endorsement-based student model, beliefs about the student may change and a plan element may become no longer justified. When this happens a planner alert is recorded. Student questions and requests that interrupt delivery also generate planner alerts.

Planner alerts are special conditions that interrupt plan delivery to signal anomalous conditions. They alert the planner to some potential problem with the current plan indicating that it may no longer correctly accounts for the tutorial situation. The student model may have changed, the instructional objectives may have changed, or the time resources may have changed. The following are the kinds of planner alerts that the planner must be able to handle:

1. *Prior goal unsatisfied* — an earlier objective that had been believed satisfied is no longer satisfied.
2. *Pending goal satisfied* — a subsequent objective for which there are pending activities or objectives is already satisfied. Perhaps the student learned the material from the class or another student, or knew it all along and had just not demonstrated the skill until this point.
3. *Current goal satisfied* — the current instructional goal or one of its ancestors now appears to be achieved even though all the activities or procedures set up for it are not finished executing. The student learned faster than expected.

4. *Student question or request*— the student has asked a question or made a request.

5. *Time threshold approaching* — the normal time for a lesson has about expired.

The means by which the planner architecture handles these alerts is discussed next.

These planner alert types are used to select activity subplans so that the plan can be adapted or revised to fit the new tutorial situation. Note that the first three kinds of planner alerts directly relate the student model and instructional plan and can be implemented by use of the TMS dependencies between the two. When assertions in the student model become OUT (no longer believed) then those parts of the plan that depend on them also become OUT, causing a planner alert.

Most planner alerts are handled by splicing a new activity plan into the current plan to handle the alert. Such a plan for handling questions might be

1. transition-away—suspend the current activity and explain the shift of discourse topic.
2. answer-question—just answer the question directly.
3. transition-back—explain that the previous activity will now be resumed then continue where it was left off.

The activity plan itself must be further refined to include procedures before that plan patch can be executed. For example, there may be different ways of directly answering the question that vary in level of detail and use of demonstrations.

Another kind of planner alert monitors the planner itself. For example, if the planner attempts to reach the same objective twice a *meta-level planner alert* will be generated. Meta-level planner alerts warn of a problem with the planner's approach in addressing problems in the domain, rather than actual problems in the plan.

Each planner alert may have several activity plans that could be used to handle it. For example, the student question planner alert could have multiple activity plans, one to answer the question, one to defer it, and one to explain why the question cannot be answered. The choice of activity plan, both to handle planner alerts and to achieve instructional objectives, is made by planning rules, discussed below.

9.4 Planning rules and plan critics

The planner requires some means of selecting activity plans, some means of selecting procedures, and some means of selecting responses (activity plans) to planner alerts. *Planning rules* are used to specify how selections are to be performed. These rules select activity plans and procedures for their steps. The rules use the same format outlined in the previous section so that knowledge acquisition utilities can apply to both plan rules, either for generation or revision, and for data interpretation rules.

The planner also uses *plan critics*. Each plan critic can only add plan elements or resequence them, but never change plan elements or delete them. These restrictions are crucial to simplify tracing problems back to the faulty application of plan critics. The library of plan critics would include critics such as the following:

1. *ADD-INTRO* — adds an instruction and lesson overview for a student.
2. *ADD-REVIEW* — adds a summary of what was covered in this lesson.
3. *RELATE-MATERIAL* — adds material that relates detailed topics to the main topic to prevent the student from getting lost in details.

Most critics basically improve discourse flow or add activities to the instruction to enrich, support, or reinforce other activities. They could be used to detect and remedy other potential problems, such as a lesson plan requiring more time than provided.

This completes the description of the planner architecture. For a new domain the instructor must provide the goal tree, the activities library, and the procedures library. Selection rules must also be provided for plan generation and for handling planner alerts.

This planner is a simplification of the planner developed in the Blackboard Instructional Planner [Murray, 90b]. It builds on knowledge gained in the blackboard implementation of that planner. The rules of this simplified planner play a similar role to the knowledge sources of the earlier planner. But there is no blackboard agenda mechanism or scheduler and the control mechanism and data structures of the planner described here are simplified. The simplified planner should be easier to understand, more portable, and more efficient, as there is less computational overhead. However, the blackboard-based planner has a more flexible control structure that is more suitable for experimenting with more sophisticated approaches to planning.

9.5 Integration of the planner and student model

The student model and planner are connected via dependency links. Changes to the student model can cause instructional objectives to go OUT or come IN. In the first case (OUT)

they are no longer considered to be achieved and in the second case (IN) they are considered to be newly achieved or reacheived. Either case leads to planner alerts.

Pseudo-English code for the planner and student model extensions to the tutor are given below. Those places where knowledge acquisition can affect the results are marked by asterisks (see next section for the discussion of knowledge acquisition).

To generate a lesson plan

```
Generate planner alert - student model needs initialization
Copy the goal network
For each goal
  *If it is already achieved according to student model
    then mark it as satisfied
    else mark it as unsatisfied
If planning to the activity level
  then
    For each unsatisfied goal
      *Select an activity subplan for it
Apply all plan critics
Deliver the lesson plan
```

To deliver a lesson plan

```
While there are unsatisfied goals after the current goal
  Advance to the next unsatisfied goal
  If it has an activity subplan already selected for it
    then Select the next unexecuted activity step
    else *Choose an activity subplan for it
      Select the first activity
  end If
  *Choose a procedure for the selected activity step
  Repeat until the procedure is finished
    Execute the next procedure step
    If there are any new assessments
      then Update the student model
    If part of the plan is no longer justified, or
      there is a new question or request, or
```

```
        the time left has about run out
      then Generate a planner alert
    end Repeat
  end While
```

To handle a planner alert

```
*Select an activity subplan to handle it
Splice in the subplan
*As each activity step becomes current select a procedure
Resume execution of main plan
```

To update the student model

```
For each skill with new PRO or CON arguments
Collect all PRO and CON arguments for the belief
Order each set according to the reliability classes
*Apply special purpose rules
If none apply then use standard lexicographic comparison
```

To compare PRO and CON arguments lexicographically

```
If there are no PRO and CON arguments left to compare
  then
    If any arguments were examined
      then the result is UNKNOWN
      else the result is UNCERTAIN
    If there are no PRO arguments
      then the result is CON
      and the strength is given by the strongest argument
    If there are no CON arguments
      then the result is PRO
      and the strength is given by the strongest argument
    {Otherwise there is at least one more pair PRO & CON args}
    Compare the strongest PRO arg to the strongest CON arg:

    (Case 1) if the PRO argument is stronger
```

```

        then the result is PRO
          and the strength is given by that arg

    (Case 2) if the CON argument is stronger
        then the result is CON
          and the strength is given by that arg

    (Case 3) otherwise
        {arguments equal or incomparable}
        call this function recursively
          on remaining arguments and return results

```

10. Knowledge acquisition tools

In this section we consider tools for knowledge acquisition. First we consider knowledge base debugging tools and then the application of machine learning algorithms.

10.1 Knowledge base debugging tools

Two kinds of rules are used in the extensions proposed for the Lower Hoist Tutor. The first kind of rule is a data interpretation rule used by the endorsement-based student model to provide exceptions to a default lexicographic comparison. The second kind of rule is a planning rule used to select items from one of the planning libraries. It is used for both generation and revision of lesson plans.

Both kinds of rules have the same format and both explicitly record dependency links, despite their differing functionality. When data interpretation rules execute, these links tie data, student model beliefs, and the rules themselves. Similarly, when planning rules execute, these links tie student model beliefs, plan elements, and the planning rules used.

These dependency links support the building of a graphical editor that allows a hypertext-like examination of the support for plan elements or student model beliefs. Starting from a plan element the editor can directly retrieve the planning rules and student model beliefs that responsible for it. Student model beliefs lead to the data interpretation rules and assessments that underly them. By examining this derivation trace faulty rules can be more easily isolated.

This knowledge base debugging tool could also store previous sets of student data and beliefs derived from them, along with sets of beliefs and plans derived from the beliefs. These stored examples could be used to test new additions to the rule knowledge base for consistency with earlier rules. If a previous example no longer produced the same result this discrepancy can be shown. The discrepancy indicates an inconsistency between the new rule and earlier rules, or some situation not previously accounted for by the earlier rules.

10.2 Application of machine learning algorithms

The area of machine learning that is best understood is that of concept learning. Concept learning algorithms can be applied to classification tasks. Each class defines a concept that represents the distinguishing features of that class compared to the others.

To the extent that the planning problem can be reduced to a classification problem these algorithms can then be applied. To apply these algorithms to the planner design of Section 9, features of lesson plans and tutorial situations would need to be related to the selection of activity plans and procedures. If useful features can be selected that are predictive of the most appropriate activity plans and procedures then these algorithms can be applied.

Another approach, case-based reasoning, could replace the use of rules in the endorsement-based student model. With this approach the student model assigns a belief label by retrieving the most similar previous case of data interpretation of PRO and CON arguments and using that label. Initially the labels would have to be provided until a sufficient case library was acquired. The challenge with this approach would be to choose an appropriate case retrieval mechanism. For example, if the current case compares two PRO arguments of one evidence reliability class to a single CON argument of the next higher reliability class there may be multiple cases that could be considered. One might contain a similar situation that occurred but using two other evidence reliability classes. Another might use the same classes but compare two CON arguments to one PRO arguments. A third case might compare three PRO arguments to two CON arguments.

11. Related work

The Lower Hoist Tutor builds on several systems. It directly incorporates elements from the Blackboard Instructional Planner [Murray, 90b] and Endorsement-based Student Model [Murray, 91]. It changes the planner by simplifying it and directly linking it to the student

model through the idea of justifying plan elements. The student model has also changed: it incorporates special-case exception rules and is implemented with object classes and methods. This object-oriented implementation makes it easy to implement overlay [Carr and Goldstein, 77] student models by allowing one class, such as the **part** class, to inherit student model functionality.

The Lower Hoist Tutor has adapted several ideas originally implemented in IMTS [Towne and Munro, 89], especially with regard to its approach to fault diagnosis. The use of a fault-effect table was adopted from IMTS. The qualitative model of the device also adopts the approach used by IMTS of characterizing parts by ports, states, and behavior rules. However, unlike IMTS, the rules are implemented as methods in an object-oriented environment designed to facilitate portability and reusability. The graphics interface also uses Hypercard-based tools, that are unlike the graphics of IMTS.

STEAMER [Hollan *et al.*, 84] is another system that has greatly influenced the Lower Hoist Tutor. The tutor presented here continues in the object-oriented style that STEAMER used and also proposes instruction generated from device and part type descriptions, as STEAMER did. However, STEAMER was intended to teach operating procedures, whereas the Lower Hoist Tutor is intended to teach troubleshooting. Both require the teaching of a mental model of device operation.

12. Conclusion

The extensions discussed for the planner and student model raise various questions for future research. How useful is a tutor that can plan versus a purely opportunistic tutor? How useful is an endorsement-based student model compared to numeric student models? Is the truth maintenance system a good approach to coupling student models and planners?

The knowledge acquisition tools also raise another set of questions. To what extent can machine learning techniques be usefully applied to knowledge acquisition in intelligent tutoring systems? Can a similar mechanism apply to both data interpretation and planning rules? Can concept learning algorithms be applied to planning by reducing planning to classification?

The implementation of the Lower Hoist Tutor shows that much can be done with current tools, even though the research questions above are unanswered. It is now much easier to build intelligent tutoring systems that run on a variety of platforms and programming languages than it was a few years ago. Hypercard-based tools greatly simplify the engineering of the graphics. Typically graphics was a large part of the effort required to build a complete system. Now device schematics can be scanned in and interactive functionality added easily, along with animation. Object-oriented programming environments and standards (e.g., CLOS) also facilitate construction and portability of intelligent tutoring systems.

There are many potential practical applications of these systems. The graphic and qualitative device simulation can be used in isolation just as a classroom training aid. The graphics can be projected to a classroom or run on multiple workstations. The instructor can control the device simulation and use it to illustrate normal and faulted operation, and to explain how to troubleshoot various faults. The students could inject faults into the model and watch the instructor troubleshoot, or take turns troubleshooting faults that the instructor has inserted.

Complete instructional systems can also be fielded outside of the classroom where equipment is used. These systems can provide practice when needed, or deliver primary or remedial instruction on requested topics. They can be used for entry-level training and assessment of troubleshooting skill. They can be coupled with on-line documentation to provide additional help illustrated with dynamic device models.

As machinery in industry and the military continues to grow in complexity the need for classroom aids and instructional systems such as these will increase. And as long as there is a proliferation of delivery platforms, operating systems, and programming languages portability will be an important feature for such systems. Portability across domains and the ability to reuse device and tutor components will become increasingly important as larger instructional systems are built for devices that have tens of thousands of components and hundreds of schematics.

Acknowledgements

I would like to thank Dr. Kurt Steuck of the Armstrong Laboratory, Human Resource Directorate (formerly AFHRL/IDI), for technical guidance during this effort.

References

[Brown *et al.*, 75] Brown, J. S., Burton, R. R., and Bell, A. G. SOPHIE: a step towards a reactive learning environment. *International Journal Man-machine Studies*. Volume 7. Pages 675 - 696.

[Carr and Goldstein, 77] Carr, B., and Goldstein, I.P. Overlays: a Theory of Modeling for Computer-aided Instruction. Massachusetts Institute of Technology. AI Lab Memo 406.

[De Kleer, 86] De Kleer, J. An Assumption-based TMS. *Artificial Intelligence*, Volume 28, Number 2.

[De Kleer and Williams, 87] De Kleer, J., and Williams, B. Diagnosing Multiple Faults. *Artificial Intelligence*, Volume 32, Number 1. Pages 97 - 130.

[De Kleer *et al.*, 89] De Kleer, Forbus, and McAllester. Truth Maintenance Systems. Tutorial SA5, Eleventh International Joint Conference on Artificial Intelligence.

[Hollan *et al.*, 84] Hollan, J. D., Hutchins, E. L., and Weitzman, L. STEAMER: an interactive inspectable simulation-based training system. *AI Magazine*, Volume 5, Number 2. Pages 15 to 27.

[Murray, 90a] Murray, W.R. A Blackboard-based Dynamic Instructional Planner. FMC Technical Report No. R-6376. *Notes*: ONR Final Report. The conference paper below summarizes this report, which presents full details of the Blackboard Instructional Planner and an earlier version of the Lower Hoist Tutor.

[Murray, 90b] Murray, W.R. A Blackboard-based Dynamic Instructional Planner. *Proceedings Eighth National Conference on Artificial Intelligence*. July 29, 1990 - August 3, 1990. Pages 434 - 441. *Notes*: This paper is drawn from the final report listed above.

[Murray, 91] Murray, W.R. An Endorsement-based Approach to Student Modeling for Planner-controlled Tutors. *Proceedings 12th International Joint Conference on Artificial Intelligence*. August 24 - 30, 1991. Pages 1100 - 1106.

[Towne *et al.*, 83] Towne, D., Johnson, M., Corwin, W. A Performance-based technique for assessing equipment maintainability. Behavioral Technology Laboratories. Department of Psychology, University of Southern California. Technical report no. 102.

[Towne and Munro, 89] Towne, D., and Munro, A.. ONR Final Report: Tools for Simulation-based Training. Technical report no. 113. USC Behavioral Technology Laboratories.

[Wenger, 87] Wenger, E. Artificial Intelligence and Tutoring Systems. Morgan Kaufmann.

[Wilkins, 88] Wilkins, D. Practical Planning, Extending the Classical AI Planning Paradigm. Morgan Kaufmann.

Appendix I—Implementing semantic networks with objects

This appendix describes how ISA and PART-OF semantic networks can be implemented in object-oriented programming languages.

I.1 ISA and PART semantic networks

Semantic networks, also called conceptual hierarchies, can be used to represent knowledge about the device being taught. Two kinds of semantic networks are commonly used, both of which can be implemented by the classes and methods of an object-oriented programming language.

The first kind of semantic network is an ISA network. Nonterminal nodes represent conceptual classes. If one class is below another then it specializes it. The objects of the subclass are in a more specific category than those of the parent class. For example, in the network shown in Figure 12 the hydraulic valves consists of two classes: **latchable valves** and **directional valves**. The terminal nodes in this tree, the ones whose names appear in italics, are the names of parts in each subclass. This kind of semantic network is called an ISA hierarchy as each link can be considered an ISA link (i.e., UVK4 is a latchable valve, a latchable valve is a hydraulic valve, etc.).

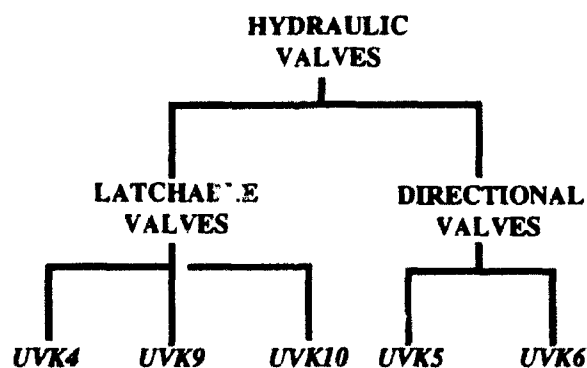


Figure 12—An ISA hierarchy

ISA hierarchies can be directly implemented in object-oriented languages. To avoid confusion we will use the terms *conceptual class* and *class member* when referring to a conceptual hierarchy, and we will use the terms *object class* (or just class if it is clear from the context) and *instance* when referring to class hierarchies in an object oriented language.

The first refers to an abstraction, the second to an implementation. Confusion can occur as ISA hierarchies can almost be directly implemented as object class hierarchies, but another kind of hierarchy (the PART-OF hierarchy, discussed below) cannot.

The ISA semantic network of the example can be directly implemented as an object hierarchy in an object-oriented language such as CLOS. For example, if properties are defined for the class hydraulic valves they will be inherited, i.e., also present, for all instances. This inheritance of slots in an object-oriented implementation mirrors the abstract notion of inheritance in a semantic network where class members inherit properties of their parent classes.

In a PART-OF semantic hierarchy the nonterminal nodes represent subassemblies (component structures) of their parent nodes. Terminal nodes represent parts that are not further subdivided. Figure 13 shows an example of a PART-OF hierarchy. Each link means "part of", e.g., UCK2 is a part of the latch valve assembly and the latch valve assembly is a part of the lower hoist assembly.

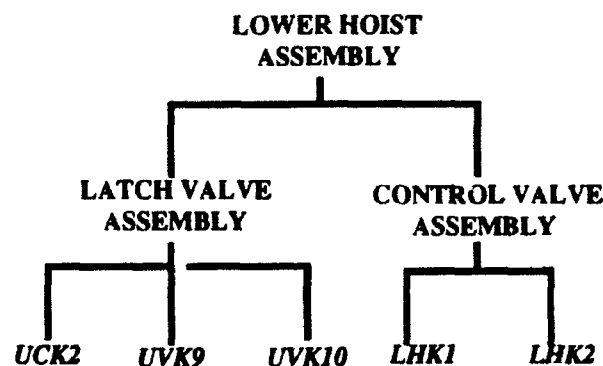


Figure 13—A PART-OF hierarchy

This hierarchy differs from the ISA semantic network because class members do *not* necessarily inherit properties from parent nodes. If we implemented the classes shown in Figure 13 as object classes in CLOS they *would* inherit these properties, so a direct implementation cannot be done.

Even with the earlier ISA hierarchy (see Figure 12) there is a problem with a direct implementation. We would like to be able to examine properties of the classes in the semantic network. These are not directly accessible in most object-oriented languages. Only the instances are readily accessible. For example, although there are instances for

each class member (e.g., UVK4 in Figure 12), there is no instance for the class **latchable valves**.⁴

One solution is to create a single object of class **concept hierarchy** that represents either a conceptual class or a class member for either kind of semantic network. Slots for each object will point to any parent class, subclasses, or instances. Another slot will indicate whether the object represents a class in a semantic network or a class member. Then the PART-OF hierarchy of Figure 13 can be implemented by creating instances of these objects and then setting the slots to mirror the network structure of Figure 13.

A similar approach can be used for the semantic network of Figure 12. But to implement inheritance of domain specific slots, such as the number of states a valve can be in, user defined classes are *additionally* provided that correspond to the classes in the concept hierarchy in Figure 12. These are shown in Figure 14.

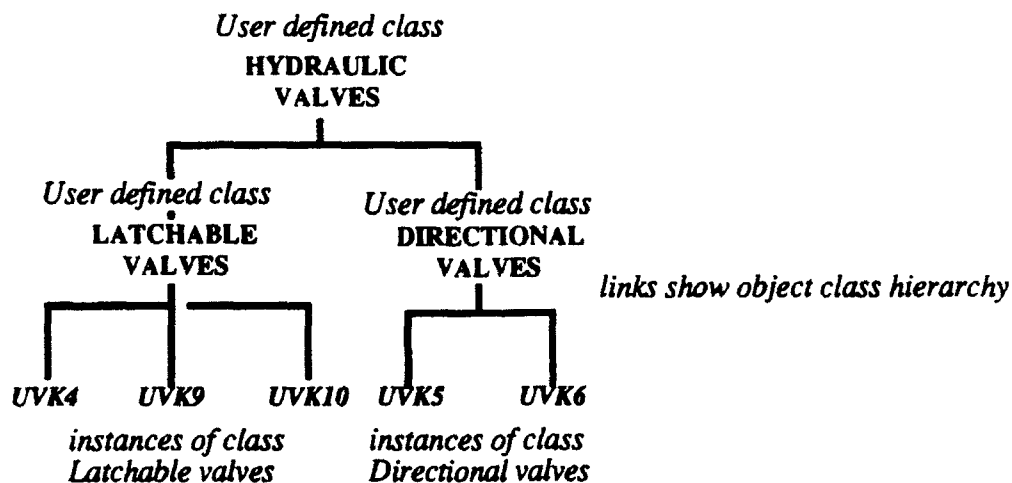


Figure 14. Object hierarchy to implement inheritance of ISA hierarchy

A second set of objects, all of class **concept hierarchy**, represent the class / subclass / member relationships as shown in Figure 15. To summarize, the first set of classes and instances in Figure 14 implement the inheritance of slots and methods in Figure 12. The second set of instances shown in Figure 15 represent the network structure and provide access to instances that represent the conceptual classes of Figure 12.

⁴Actually in CLOS one could access a CLOS object of class **standard-class** that corresponds to the user-defined class. But no such object would exist in C++.

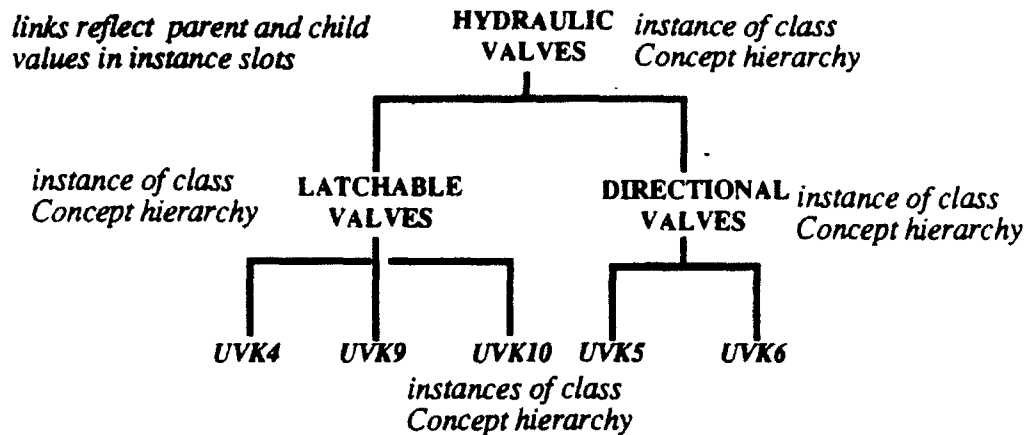


Figure 15. Linked instances of class **concept hierarchy** to represent class-subclass-member relationships in ISA hierarchy

Methods defined for all objects are inherited to individual instances in the first hierarchy. So, for example, methods that predict part state changes are defined in the first hierarchy. A **simulate-operation** method is defined for the class **latchable-valves** that applies to all of its instances. A different **simulate-operation** method is defined to apply to the class **directional valves** and all of its instances. Methods defined for *both* conceptual classes and class members are defined in the second hierarchy. So, for example, student modeling methods that represent the degree to which skills have been generalized are defined in this hierarchy. Such methods are defined for the class **concept hierarchy** and can apply to instances that represent either individual parts or classes of parts. In this way the tutor can represent the extent to which a skill has been learned for a class of parts.

In CLOS the second network (the objects of type **concept hierarchy**) can be constructed automatically as the first network is defined. The Lower Hoist Tutor defines an **after-init** method for the CLOS class **standard-class**. This method defines a new object instance of type **concept hierarchy** after each user-defined class is added. It is called an **after-init** method because it is called after each instance is initialized. For example, when the user defines the CLOS class **hydraulic-valves** this method will create an instance of class **concept hierarchy** to represent that conceptual class.

Links between instances of the **concept hierarchy** class are added automatically by a second method. This method is a second **after-init** method for objects of type **concept hierarchy**. It accesses the CLOS class object, an instance of class **standard-class**, that is being represented by the instance of class **concept hierarchy**. It can determine the

superclasses of that class from CLOS internal accessors. Next it can add links between the instances representing parent and child classes. So if the user-defined CLOS class **hydraulic-valves** has a single super class **part**, also user-defined, then the **concept hierarchy** instance for the first will be linked to the **concept hierarchy** instance of the second. As the user-defined classes are defined the **concept hierarchy** instances are created and linked together.

I.2 Implementation in an object-oriented language

Now we can provide an overview of how the tutor is implemented using objects. The top level of the object hierarchy is shown below:

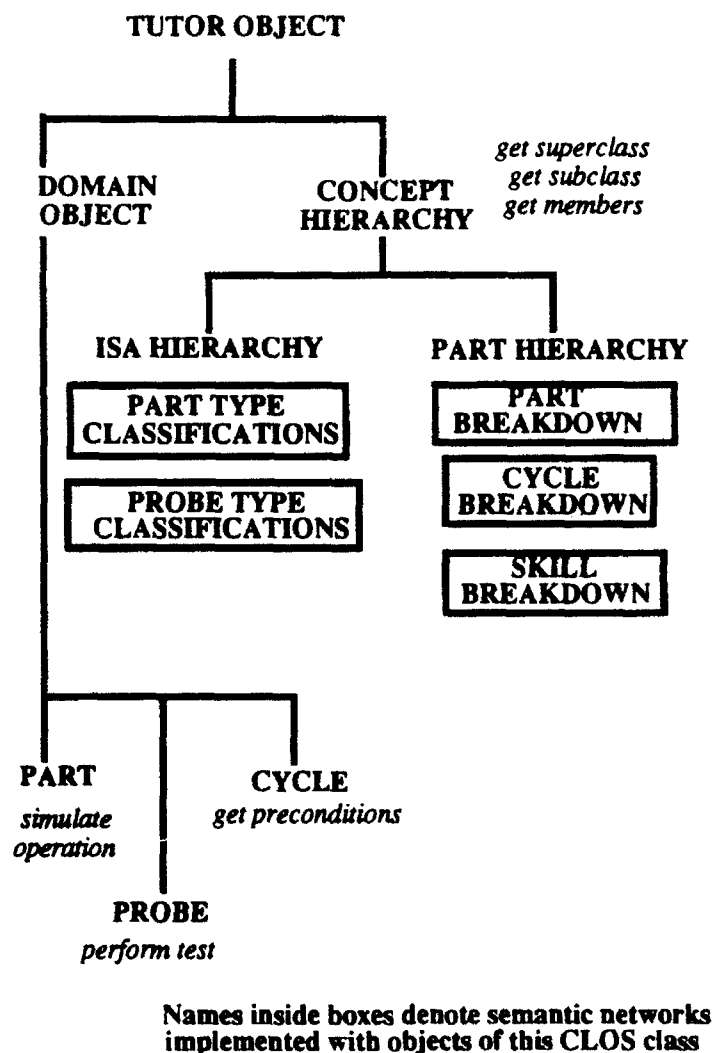


Figure 16. Top level class hierarchy in object-oriented implementation

All objects used in the tutor inherit from **tutor object**. Objects specific to a domain, such as **part names** and **structures**, or the names of specific troubleshooting tests (called **probes** here) are also part of the **domain object** class. Different cycles of the device corresponding to different operating modes are also part of this class. Objects of type **concept hierarchy** represent nodes in either a ISA or PART-OF semantic network.

Examples of class methods are shown in *italics*. For example, the **part** class has methods to simulate the operation of specific parts. Objects of type **concept hierarchy** have **superclass**, **subclass**, **class members**, and **type** (indicating whether class or member) methods.

Most of the qualitative model of device operation is implemented by methods defined on the **part** class or its subclasses (shown later). Most of the student model is implemented by methods defined on objects of type **concept hierarchy**. Methods to demonstrate, assess, or explain troubleshooting skills are defined on the class **skill**, a subclass of the **part hierarchy** class. Methods to explain the operation of parts or test the student's understanding of the operation of parts are defined as methods on the **part** class.

Appendix II—An object-oriented implementation of the ESM

The Lower Hoist Tutor has only a partial implementation of the ESM that is not yet coupled to the planner discussed in Section 9. Object classes are defined for different endorsement reliability classes and for endorsements themselves. Another object class is defined to allow endorsements to be associated with skills that apply to an object. This latter class is intended to be a *mixin*, that is, it is to be added to some other class, such as the class *part*, so instances of that class inherit both methods associated with that class (e.g., defining how the parts operate) and the ability to store endorsements associated with skills that apply to that object.

Propagated and inferred endorsements are linked to the endorsements or assessments they depend on through a truth maintenance system, the justification-based truth maintenance system (JTMS) of De Kleer [De Kleer *et al.*, 89]. This mechanism ensures that all inferred endorsements are updated automatically when necessary. The JTRE rule-based production language used in [Murray, 91b] is not used currently as all ESM functionality is implemented in LISP and CLOS. However, the JTRE is one possibility for implementing the rule language discussed in Section 8 for special-case rules and for storing dependencies between data interpretation rules and their conclusions. The special-case rule language has not been implemented yet.