Using Hypermedia to Develop an Intelligent Tutorial/Diagnostic System for the Space Shuttle Main Engine Controller Lab

Daniel O'Reilly
Robert Williams
Kevin Yarbrough
Rocketdyne Division Rockwell Intl.
2227 Drake Ave. Suite 45
Huntsville Al. 35805

Abstract

This system is a tutorial/diagnostic system for training personnel in the use of the Space Shuttle Main Engine Controller (SSMEC) Simulation Lab. It also provides a diagnostic capable of isolating lab failures at least to the major lab component. The system was implemented using Hypercard, which is an implementation of hypermedia running on Apple Macintosh computers. Hypercard proved to be a viable platform for the development and use of sophisticated tutorial systems and moderately capable diagnostic systems.

This tutorial/diagnostic system uses the basic Hypercard tools to provide the tutorial. The diagnostic part of the system uses a simple interpreter written in the Hypercard language (Hypertalk) to implement the backward chaining rule based logic commonly found in diagnostic systems using Prolog.

Some of the advantages of Hypercard in developing this type of system include sophisticated graphics capablility, the ability to include digitized pictures, animation capability, sound and voice capability, and its ability as a hypermedia tool. The major disadvantage is the slow execution time for evaluation of rules (due to the interpretive processing of the language). Other disadvantages include the limitation on the size of the cards, that color is not supported, that it does not support grey scale graphics, and its lack of selectable fonts for text fields.

Introduction

The lab for which the tutorial/diagnostic was developed provides an integrated test environment for verifying the software for the Space Shuttle Main Engine Controller (SSMEC). It includes real and simulated engine hardware components. The lab software controls the hardware through several computers to allow the test engineer to force off-nominal conditions and record the reactions of the SSMEC. The central theme followed in developing the lab was that all actions and results for the tests to be conducted should be contained in a single test procedure. Additionally, the entire process should be automated to the point that the user could conduct a series of tests by entering one command to the VAX. Finally, all actions taken by the user, the lab components, and the results must be logged such that one could exactly repeat a

467

test at a later date. Under these provisions many of the machinations of the lab remain invisible to the user. This tutorial and diagnostic was designed to help lab users to understand the lab and isolate problems in the lab.

The Tutorial/Diagnostic

The prototype tutorial/diagnostic system was initially implemented using Turbo Prolog on an IBM PC and later implemented using Hypercard on a Macintosh (Hypercard is Apple's implementation of hypermedia). In ease of development, particularly in the tutorial portion, Hypercard proved to be easier and faster to use than Turbo Prolog. The Hypercard version includes extensive graphics, some animation, and some sound.

The system addresses four major areas; the use of the tutorial/diagnostic, conducting tests in the lab, the hardware operation of the lab, and the diagnostic. The part illustrating lab operations has not yet been completed. Four buttons on the top card of the stack control entry into each of these areas. When the user selects one of the buttons, this top card is "pushed" so that is may later be "popped" in response to clicking a "return" button. Figure 1 illustrates the top level layout.

Using the System
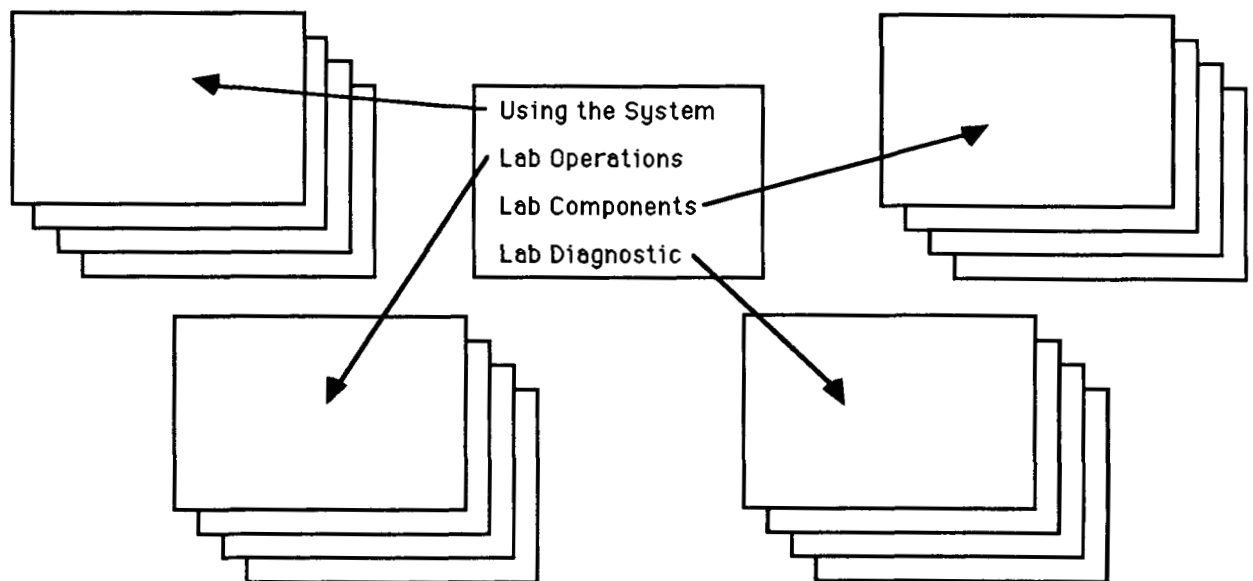Lab Operations
Lab Components
Lab Diagnostic

Figure 1
Top Level Layout

The first area instructs the user in using the tutorial/diagnostic. It attempts to illustrate rules the user can use to recognize buttons, navigate through the system, and get more information on a subject. To implement this area the developers used only the most basic capabilities of Hypercard, namely, graphics, text, buttons, push and pop, and linking cards. This area has no links to the other three areas to prevent the

468

first time user from getting "lost" in the stack. Instead, copies of cards from the other areas are used to illustrate the system.

The area providing the tutorial on lab components consists of a main path which includes a brief description of each of the major components. Each of the descriptions of major components provide two side paths; one to a more detailed operational description of the component, and the other to a detailed hardware description of the component. As in the first area, only the most basic Hypercard capabilities were used. Figure 2 illustrates the layout of this area.
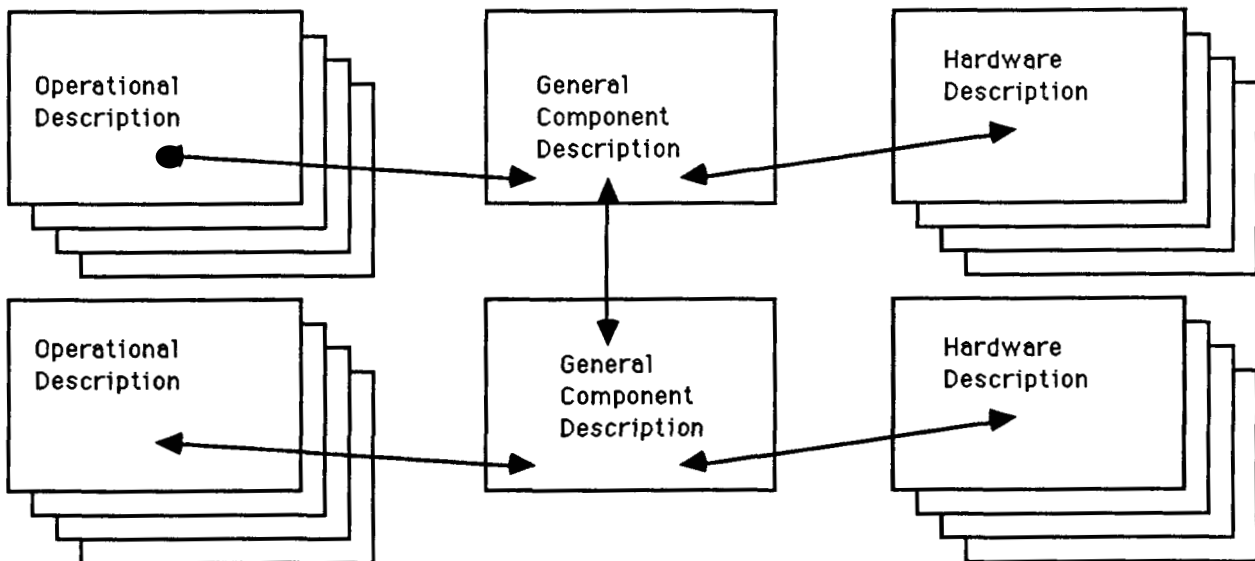

Figure 2
Lab Component Description Layout

Three basic Hypercard linking mechanisms were used to control navigation through the cards for these first three areas; direct linking of cards, go to next or previous card, and push and pop.

Direct linking of cards displays another card in response to a user action such as clicking on a button. The developers used this mechanism to go to subsets of cards from a card with multiple options. For instance, the top card of the stack gives the user four options via buttons. Clicking one of these buttons causes Hypercard to go to the first card of the subset for that option. This mechanism was also used for "help" functions where the user clicks on a button to acquire more information about a component or a test.

Go to next or previous card provides a mechanism for the user to move backward or forward to adjacent cards. This mechanism was used to allow the user to move freely among cards of a subset.

Push and pop pushes a card to the stack or pops a card from the stack. The developers used this mechanism to allow the user to

return from a subset to the card at which he chose the subset. For example, when the user clicks on the button to choose a subset, the current card is pushed. The cards of the subset each have a button for "return". When this button is clicked, the card on top of the stack (in this case, the card from which the selection was made) is popped. Since push and pop are an implementation of the familiar stack operators, this mechanism may be used to "nest" this return capability.

Through the use of these basic capabilities the developers built in an orderly navigation scheme through the cards for the tutorial parts of the system. One should note that these capabilities may also be used to add implicit logic to the system in that they can be used to implement trees. Figure 3 illustrate the basic linking methods.
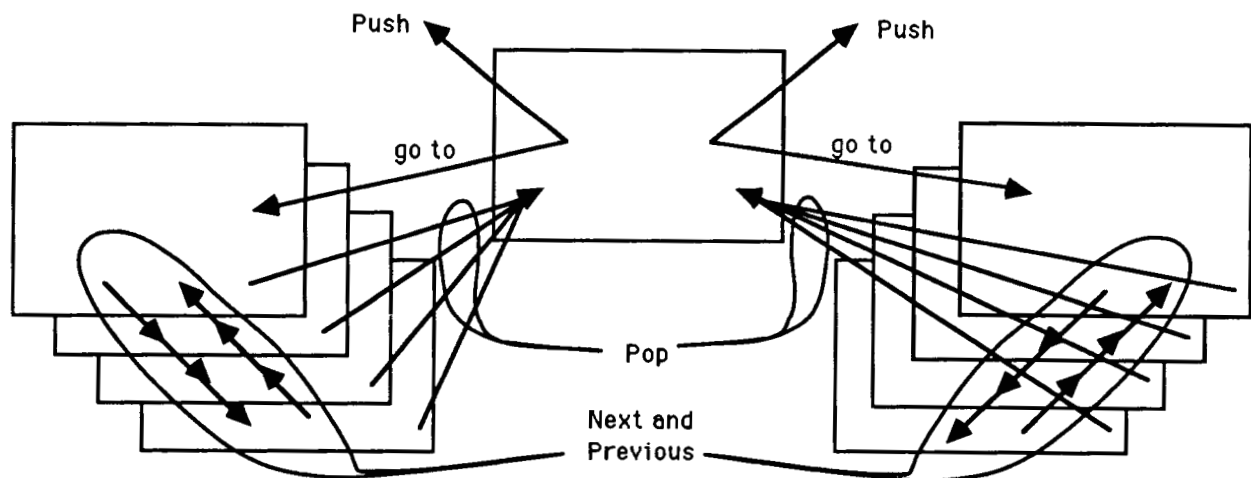

Figure 3
Basic Linking

The diagnostic comprises the fourth part of the system. This part allows the user to isolate bad components in the lab by answering a series of questions as to the symptoms exhibited at various points in the lab. Initial attempts at implementing this part included using a straight tree structure and then a modified decision tree. Ultimately, the Hypertalk capability of Hypercard was used to implement a Prolog-like rule based logic typical of many diagnostic systems.

In the first attempt, the straight tree structure quickly became too large to manage due to combinatorial expansion. In addition to becoming difficult to manage, it required the duplication of many of the symptom cards. Thus, for a system that involves any complex interrelationships or that is of any size, this method proves too cumbersome.

The next iteration attempted to simplify the tree by "modularizing" some intermediate tests and calling them from nodes

470

in the tree. To implement this scheme the push and pop operators provided a mechanism to return to the node in the tree which called the intermediate test in order to continue the diagnostic. However, this scheme proved lacking in that useful intermediate tests were difficult to define due to the interrelationships of the lab components.

The solution involved writing a script using Hypertalk to emulate the backward chaining rule based logic typically used in developing diagnostic systems with Prolog. This scheme proved to be relatively simple to design and use and has the added advantage of providing a shell that could be used for any diagnostic system.

The diagnostic uses four basic types of cards. These include the beginning card, test cards, conclusion cards, and symptom cards. Scripts at the stack level record the results of symptoms and tests, evaluate tests, and provide the navigation among the cards.

The beginning card serves as an introduction to the diagnostic. When the user selects "continue", the script for the "continue" button sends a message to the handler that initializes variables for the diagnostic session and pushes the first test onto the test stack. Figure 4 illustrates the layout of the beginning card.
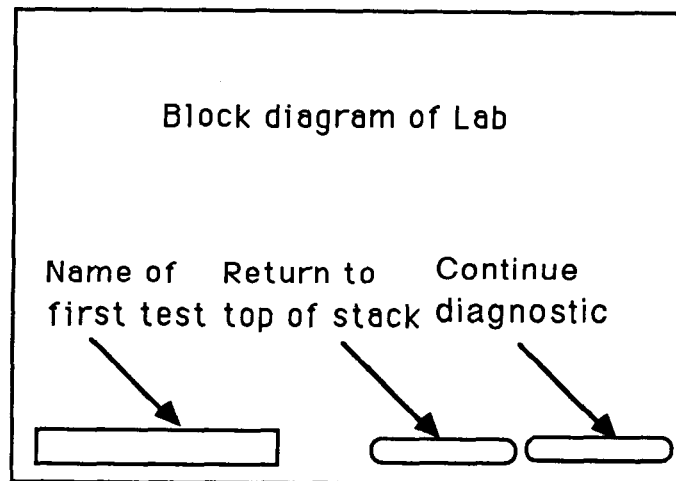

Figure 4
Layout of Beginning Card

Test cards actually define tests and may themselves be symptoms for other tests. These cards contain a description of the test in the language processed by the evaluation script. This language basically allows the knowledge engineer to describe a test in terms of boolean functions. The knowledge engineer enters this description into background field 1 which covers the upper 2/3 of the card. The outline of the test is basically an "if-then-else" statement where the conditions to be evaluated lie between the if and the then keywords. Parentheses may be used to control

evaluation. Since tests themselves evaluate to true or false, they may be used in the description of other tests. However, all tests must eventually reduce to a set of symptoms. If the user inadvertently generates an endless loop (for example: making test A dependent on test B which is dependent on test C which is dependent on test A) the diagnostic, at run time, will post an error in the message box. Test cards allow the user the options to continue the diagnostic or to return to the beginning card. If the user elects to continue the diagnostic, the script for that button sends a message to push the name of the test card on the test stack and then sends a message to evaluate the test at the top of the test stack. Figure 5 illustrates the layout of a test card.
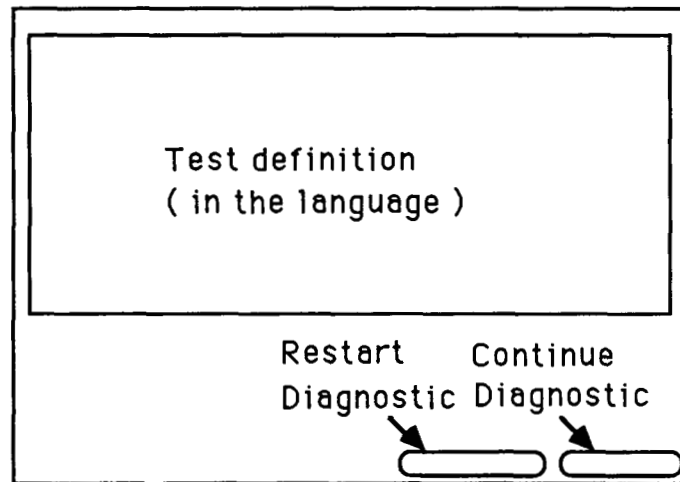

Figure 5
Layout of a Test Card

Symptom cards ask the user to enter the status of some test point in the lab. The user selects a button labelled "good" or a button labelled "bad" to indicate status at that point. Clicking the button sends a message to a handler at the stack level which records the symptom and its status. It then requests the evaluation script to evaluate the test. In addition to acquiring results, the symptom cards also allow the user to select, via buttons, more information on the lab components and the test being performed. Figure 6 illustrates the layout of a symptom card.

Conclusion cards provide the message identifying the bad component if one is found. Note that a card stating that no bad component could be identified is also a conclusion card. The knowledge engineer enters these card names in the "true" path of the test description on a test card. When the test proves true, this card is displayed. The conclusion card allows a user the choice of continuing the search for bad components or returning to the beginning card to begin a new session. Figure 7 illustrates the layout of a conclusion card.
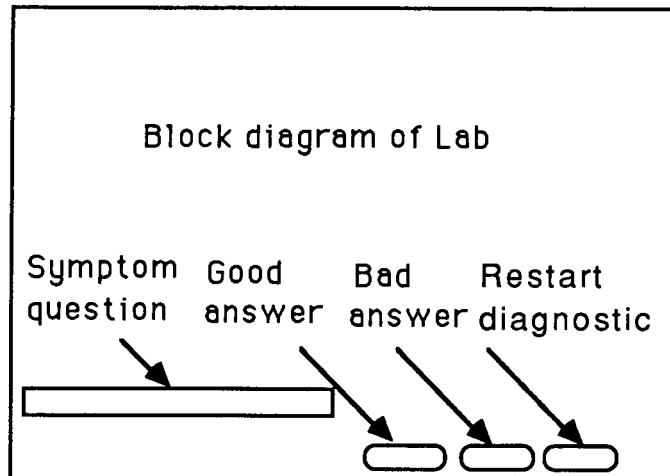
472

Block diagram of Lab

Symptom   Good   Bad   Restart
question answer   answer   diagnostic

Figure 6
Layout of a Symptom Card

Block Diagram of Lab

Conclusion   Restart   Continue
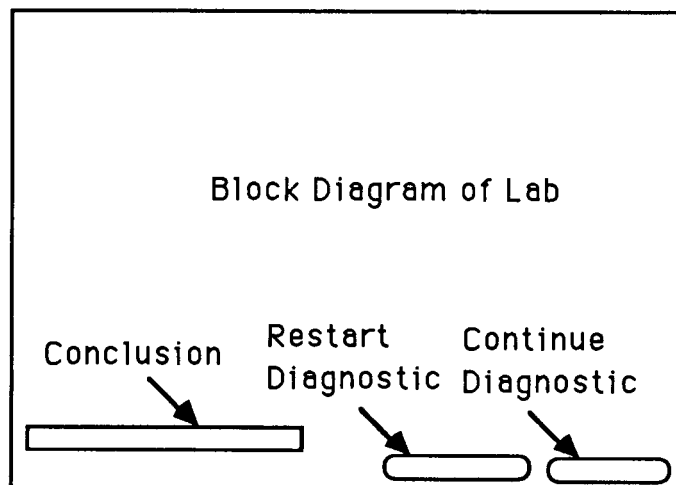   Diagnostic   Diagnostic

Figure 7
Layout of a Conclusion Card

The evaluation script evaluates the language on the test cards to determine the result and the action to take. This script consists of a simple parser to evaluate an "if-then-else" statement and provide branching to other cards as necessary or as dictated by the results of the "if". The following is the BNF for the test language.

```
<test_card_statement>      ::= 'if' <conditional> | 'ifopt'
                           <conditional>

<conditional>              ::= <expression> 'then' <action> 'else'
                           <action>

<expression>               ::= <simple_expression> |
                           <parenthetical_expression>
```

```
<parenthetical_expression>   ::= '(' <expression> ')'

<simple_expression>          ::= <status_card_id> <status> |
                                 <status_card_id> <status> <operator>
                                 <expression>

<status_card_id>             ::= <test_card_id> | <symptom_card_id>

<status>                     ::= 'good' | 'bad'

<operator>                   ::= 'or' | 'and'

<action>                     ::= <test_card_id> |
                                 <conclusion_card_id> | 'return'

<test_card_id>               ::= the name of a test card (there must
                                 be no intervening blanks)

<symptom_card_id>            ::=   the name of a symptom card (there
                                 must be no intervening blanks)

<conclusion_card_id>         ::= the name of a conclusion card
                                 (there must be no intervening blanks)
```

NOTE: white space must separate all tokens

The language offers two versions of the if statement: the 'if' and the 'ifopt'. The parser evaluates the 'if' version completely, regardless of whether the outcome can be true or false. The 'ifopt' version ceases evaluation as soon as the outcome can be determined. For example, in the statement 'if A and B and C', if A is false the entire statement evaluates false. Therefore, under the 'ifopt' version, the evaluation would cease and the 'else' path would be chosen.

The names of the cards must be used in the text describing the conditional part of the test. When the evaluation script encounters an identifier (card ID), it examines the status list to find out if the status of the test or symptom has already been determined. If so, it continues the evaluation. If not, it pushes the name of the card containing the current test and performs a "go to" the card name for which the status is unknown. If that happens to be a test card then that card is displayed and when the user chooses "continue" that card name is pushed on the test stack and a message is sent to evaluate the test on top of the test stack. If the card with the unknown status is a symptom card, that card is displayed and the user chooses "good" or "bad" for that symptom. The script for the "good" or "bad" buttons sends a message to the status message handler to record status and then sends a message to evaluate the test on top of the test stack. The card names direct navigation through the diagnostic and the evaluation script really makes no distinction between test cards and symptom cards.

474

Likewise, card names must be used for the "action" part of the test description. If the conditional part of a test description on a test card evaluates true the evaluation script performs a "go to" the name of the card following the "then" keyword. If the evaluation proves false the evaluation script performs a "go to" the card name following the "else" keyword. These cards may be test cards or conclusion cards. The "return" keyword presents an exception in that when it is encountered, the evaluation script sends a message to perform an evaluation of the test on top of the stack.

In the following example, all names in the conditional part of the if statement are test card names and both names in the action part of the test point to another test card. In this example the evaluation would continue at the test called "NextTest" regardless of the outcome of the tests named in the conditional part. The user could prevent this if any of the tests in the conditional part found a bad component by electing to restart the diagnostic.

```
if SSMECScaling good or
DPM good or
ADC good or
Hardware good or
GainDAC good or
OffsetDAC good or
HardwareSIASwitch then
NextTest else
NextTest
```

In the following example, the conditional part of the test description names only symptoms. In this example, the evaluation would branch to the conclusion card "SSMECScalingBad" if the results of the test were true and would return to evaluate the previous test if the result was false. In this example all of the conditional part of the test consists of symptoms. Also, notice that since the "ifopt" keyword was used, not all symptoms would necessarily be evaluated. For example, if the symptom "OffsetVDT" proved good, the "else" path would be taken since the entire conditional must be false. However, since the status of "GainVDT" had already been evaluated, its status would be recorded and that symptom card would never show up again during this session. Note also that if the evaluation reached "GainDPM" and "GainDPM" was good, the evaluation would terminate and take the "then" path since that was all that was required at that point to make the entire statement true.

```
ifopt GainVDT bad and
OffsetVDT bad and
LocalPotVDT bad and
( GainDPM good or
OffsetDPM good or
LocalPotDPM good ) then
```

```
SSMECScalingBad else
return
```

The fact that card names are used to direct the evaluation allows the system to function as the basis for any diagnostic type application. The evaluation script really knows nothing of the target system, but merely evaluates the boolean expression on the test cards and uses the card names to direct its action. Thus, this system could provide a shell for the development of prototypes for other diagnostic systems.

Conclusions

Excluding the two false starts, the diagnostic part of this system required about 50% of the effort that was spent on the Turbo Prolog version. The majority of this gain was due the the ease of implementing the tutorial and graphics parts. In these areas, the development gains afforded by Hypercard could range from 50-90% over a similar system developed using Turbo Prolog, depending on the amount of graphics used. Also, Hypercard's ability to include digitized photographs on the cards represents a significant advantage in developing tutorial systems. Another significant advantage of using Hypercard, at least for developing tutorial systems, is that the developer need not be a programmer to develop a successful system.

There are, however, some shortcomings in Hypercard. The execution time for the evaluation of the rules in the Hypercard system is much slower than that in Turbo Prolog. This fact is masked somewhat by the difference in the speed of the graphics of the two systems. Other shortcomings include the limitation on the size of the cards, that color is not supported, that it does not support grey scale graphics, and its lack of selectable fonts for the text in fields.

Overall, Hypercard provides a useful tool for developing sophisticated tutorial systems and moderately sophisticated diagnostic systems. Its ability to easily combine graphics, text, digitized photographs, animation, and sound, as well as its ability to function as a hypermedia tool makes it very powerful for developing tutorial systems. These capabilities also offset some of its limitations when developing diagnostic systems where these functions would prove useful.

476