Computers Math Applic. Vol. 20, No. 9/10, pp. 111–123, 1990
Printed in Great Britain.

0097-4943/90 $3.00 + 0.00

INTELLIGENT INTERACTION IN DIAGNOSTIC EXPERT SYSTEMS

Y. LIROV and S. RAVIKUMAR
AT&T Bell Laboratories, Holmdel, NJ 07733, U.S.A.

Abstract—Advisory systems help to improve quality in manufacturing. Such systems, however, both human and computerized, are less than perfect and frequently not welcome. Sharp separation between working and learning modes is the main reason for the apparent hostility of advisory systems. Intelligent interaction deploys computerized advisory capabilities by merging working and learning modes. We have developed a knowledge-based interactive graphic interface to a circuit pack diagnostic expert system. The graphic interface integrates both the domain knowledge (i.e. circuit pack) and the troubleshooting knowledge (i.e. diagnostic trees). Our interface dynamically changes the amount of detail presented to the user as well as the input choices that the user is allowed to make. These changes are made using knowledge-based models of the user and of the circuit pack troubleshooting domain. The resulting system, McR, instead of guiding the user by querying for input, monitors users actions, analyzes them and offers help when needed. McR is able both to advise "how-to-do-it" by reifying shallow knowledge from the deep knowledge, and to explain intelligently "how-does-it-work" by abstracting deep knowledge from the shallow knowledge. McR is used in conjunction with the STAREX expert system which is installed at AT&T factory.

1. INTRODUCTION

In this paper we discuss the development of advisory interfaces for electronic circuit pack troubleshooting. Advisory interface is a term used to describe collectively any training and reference material, on-line help and guidance, and any other user support means. A diagnostic expert system is an example of a computer-based advisor. Computerized advisory systems are hard to build and difficult to use. Since a consultation is a break in work continuity, new users try to skip around in a training sequence or dismiss training altogether [1]. Additionally, as Ref. [2] noted: "Studying advisory problems and developing advisory solutions for the leading edge of human–computer interaction is hampered by the fact that the leading edge must have already been codified and deployed before advisory problems can even exist."

Another basic difficulty in developing advisory interfaces is the lack of understanding of coaching and interacting. For example, people often seek advice by making claims about the possible answer to their (unannounced) query. Does it mean that in such cases a direct question is less efficient? Another important issue is the frequency of an advice, or how often should the advice be offered? A good parent does not teach the children at every occasion when something is not up to the standards of the parent. How detailed should be the advice? When is it more important to provide the declarative advice (i.e. how-does-it-work) and when—the procedural advice (i.e. how-to-do-it)?

In addition, it is important to note that the advisor is constantly under suspicion of giving the wrong advice. The advisor loses credibility after displaying even the smallest deficiencies. What are the techniques to restore credibility? And what are the answers to all these questions, when the advisory interface is a computer program.

1.1. Research and Development Topic

In this subsection we refine the above outlined research topic to a task: we specify what exactly we want a computer to do. Having developed an expert system for a manufacturing facility (STAREX, see Lirov [3]), we experiment with a possible solution to the most important question: how to deploy a less than perfect computerized advisor? Basically we are trying to improve the classical expert systems which patronize the user by issuing queries, explaining only when asked, and giving the result only at the end of the session. Such systems treat the user as an object for providing the details which are necessary to complete the reasoning done by the machine. These systems also maintain a sharp separation between the learning and working modes.

1.1.1. The task

Carroll and Aaronson [2] address the same question by simulating an intelligent advisory system for an interactive software design package (the "Wizard of Oz" technique). We build on their work in that we consider the same issue, but we restrict ourselves to a relatively narrow application domain. By considering a narrow application domain we expect to be able to go a step further from simulating an interface to actually writing its code and observing its behavior. We build an add-on software module, called McR, which merges working and learning modes.

1.1.2. The domain

We chose our application domain to be the electronic circuit pack diagnostics for two reasons: first, diagnostics is the widest expert systems application domain; and second, diagnostics of electronic circuit packs is the better understood diagnostic problem because of the availability of deep knowledge [3, 4]. We build on STAREX expert system in that we regard circuit pack diagnostic models to be available. Since STAREX is a deployed expert system, we expect our actual intelligent advisory system to be deployable at the factory floor level. We note here that to ease on our programming efforts, we have used a simplified STAREX version, which does not include truth maintenance. This issue is postponed for future research. On the other hand we reuse the software modules performing the optimization of the diagnostic sequences [5]. For the sake of better paper readability, we digress briefly to explain the basic concepts in electronic circuit pack diagnostics. For an overview of the topic the reader is referred to Ref. [4].

1.1.2.1. Diagnosis. Diagnosis of an electronic circuit usually refers to the process of determining the faulty component(s) that cause an undesired behavior (output) of the given circuit for some (correctly) given input. Diagnosis can be regarded as a problem of economic optimization: there is a value associated with every component in the pack, as well as a fiscal value associated with the process of assembly and soldering of the pack. Thus, when a pack is declared faulty, it is desirable to replace only the faulty components.

Moreover, the process of identification of the faulty components (the diagnostics) consists of a series of tests each of which has an associated cost, expressed in such parameters as test setup time, component destruction, etc. The successful troubleshooter must be able to select an appropriate strategy, conduct measurements and replace components. Morris and Rouse [6] report that humans are not good in judging failure rates, human performance degrades as systems become larger and more complex and/or in the face of time constraints, presentation of theory of operation does not improve performance and proceduralization improves performance.

Diagnosis is a difficult problem for the beginners. An unskilled troubleshooter has difficulties in making an indictment, doing it correctly and doing it sufficiently fast. As a result, a beginner causes buildup in work in process, unnecessary test-repair cycles, unresolved manufacturing problems (e.g. a faulty robot or a bad batch of components), and even sometimes overlooked design problems. Therefore, the construction of automated means for troubleshooting guidance is justified both as an intellectual and as an economic challenge. However, the diagnostics problem (i.e. the construction of a computer algorithm which is capable to perform diagnostics) is NP-hard [7, 8].

1.1.2.2. Troubleshooting strategies. Two basic approaches and their combinations are currently utilized in industry for conventional diagnostic software development: the fault dictionary approach and the guided probe approach. The fault dictionary approach requires simulation of the faulty behavior of the system and subsequent storage of the simulation results (as well as the fault assumptions) in the fault dictionary. A "misbehaved" output of the pack under test can therefore be looked up in the fault dictionary. The fault dictionaries are not being used widely since they are usually incomplete and ambiguous.

The guided probe approach requires only simulation of the correctly functioning circuit pack. The basic tool in utilizing this approach is the blame-shifting mechanism applied after each measurement. The upstream (from the test point view) components in the circuit are blamed only when the measurements do not agree with the expected results. The efficient sequencing of measurements becomes the main problem when implementing the guided probe approach.

The troubleshooter must employ some kind of strategy in searching for the source of the difficulty. Ref. [9] noted that poor troubleshooters mader fewer tests before accepting hypothesis as

correct, they had more incorrect hypothesis, and they pursued incorrect hypothesis longer than did the better troubleshooters. Glaser and Phillips [10] associated more than 20% of strategic shortcomings (e.g. insufficient testing) with faulty inferences (e.g. misinterpretation of a test). Additionally, poor troubleshooters tend to have incomplete lists of hypothesis and to be frequently overconfident about their completeness [11].

1.1.3. The view

McR behaves like a diary for the user, where the user enters the measurements and their results. The system collects this data and tries to reconstruct the reasoning of the user. If McR discovers a significant reasoning fault on the user's behalf, then it offers guidance. If the fault is a technical fault (e.g. using wrong measurement device), then the advice is a low-level, "how-to-do-it" advice. Otherwise, it is a "how-does-it-work" advice. Such advice can be about the test strategy (e.g. a diagnostic tree), or about the circuit pack (e.g. signal path). This combination of two approaches (the procedural and the declarative system explanations) is most likely to be the most effective means of troubleshooting advice [6], since the system merges the Socratic and the "learning by doing" methods of instruction.

1.1.4. Additional benefits

An important expert systems characteristic is its flexibility to acquire knowledge. Expert systems usually mimic the behavior of experts which combine knowledge about several domains. Experienced circuit pack troubleshooters, for example, have some understanding about the circuit pack, know how to use measurement equipment (e.g. oscilloscope), have knowledge about the manufacturing process and its weak spots (e.g. "that robot loses its calibration frequently and thus inserts wrong components", or "this transistor is often bad"), and know about generally good troubleshooting strategies (e.g. "divide and conquer"). Shallow knowledge bases contain rules which represent the combined expert knowledge. It is difficult to maintain such knowledge bases as it is difficult to comprehend all the interdependencies between the rules.

Deep knowledge bases, on the other hand, promote easier maintenance by segregating different kinds of knowledge in separate knowledge bases. The price for this convenience is the need for an integrated inference engine which may take a form of a meta-interpreter [3, 4]. Construction of user interfaces for such knowledge bases is complicated by the necessity to maintain all the knowledge bases simultaneously. Our system is able to acquire different kinds of knowledge (e.g. electronic signal path or troubleshooting tree) and immediately show the implications in the compiled (shallow) form. We demonstrate this flexibility using an integrated graphics interface.

From the methodological point of view, this experiment helps to understand better the taxonomy of problems related to artificially intelligent diagnostics. In particular, we show that the contemporary subdivision of the issue to seven subproblems [4] is rather superficial: the user interface problem is not a stand-alone problem, to be solved separately. Its solution includes solving all the typical problems for the knowledge-based applications, e.g. choosing knowledge representation, acquiring knowledge, etc.

1.2. The Approach

1.2.1. User model

Roughly, we develop intelligence of the advisory system by creating a user model. The model can be used as a reference with which the actual user performance can be compared. The task of building intelligent advisory system through a user model seems to be tractable for three reasons: first, we can inventory the errors of the analysts; second, we expect to develop shallow knowledge disassembly mechanisms to map from observations to user errors; and finally, we have developed in Ref. [3] a multi-source deep knowledge integration mechanism (reification), which we use to check the success of the disassembly of knowledge. Systematic shallow knowledge disassembly (extracting deep knowledge) is a new interesting problem having much in common with machine learning.

1.2.2. Reification and abstraction

Coding an intelligent advisory system is a large project requiring solutions to a variety of problems: designing knowledge representation, control strategy, building knowledge base, integrating multiple sources of deep knowledge about electronic circuitry, developing user model, acquiring knowledge and displaying advice in a variety of ways. Our previous report [3] addressed the first four problems, while in this paper we deal with the last three. We show that the algorithms for knowledge reification and abstraction are the cornerstones in the intelligent advisory systems. This view holds promise for further development because it provides better understanding of both reification and its inverse—abstraction. Furthermore, we introduce a new use for abstraction—we view it as a tool to test user behavior patterns.

1.2.3. Performance criteria

The performance of the proposed method can be evaluated by testing it for correct and timely user error classification. While correct classification depends on the user model and diagnostic algorithm, the timeliness of classification depends on how much of the relevant information has already been preprocessed. Being able to preprocess most of the information beforehand holds promise to be able to deliver timely and correct advice.

1.2.4. Programming techniques

The programming approach is similar to that of knowledge-based programming since we use knowledge to supplement the observed user actions to generate our interpretations. The difference, of course, is that the product of the system is not a program code, but an advice. We use object-oriented logic programming to develop graphic interface, and metaprogramming—for reification and abstraction.

1.2.5. The scope

Our method might scale up to bigger systems, if deep and differentiable shallow kinds of knowledge can be integrated, and a complete and finite inventory of human errors are available. Of course, certain types of intelligent help will always be missing because of the inherent brittleness of such systems. The system is honest about its limitations when it knows them: the system will tell that something is wrong but will not extend a temporary solution. We advocate such an approach to all advisory systems (machine and human) in order to maintain their credibility.

Our next question is: How well is the method understood? We confess that we do not understand the method very well. The reason is that the method depends on having complete knowledge about the system under diagnosis and about the human errors. We are unable to prove that we have completed the acquisition of either kind of knowledge, let alone the mapping between the human error inventory and electronic circuitry deep knowledge. Thus, paradoxically, the only way to complete knowledge acquisition is to write down a program with incomplete knowledge to deploy it, to provoke the difficulties, and to learn experimentally about the limitations of the method. As a result, we expect not only to expand the scope of the knowledge base but also to refine the knowledge about human analyst errors.

1.3. Design for Experimentation

Once we decided to use programming as the main exploration tool, we must verify that our program is going to support experimentation in addition to having the required performance level. To ensure "experimentability", we write the code in Prolog, in order to have all the inference mechanisms readily available for alteration. We will be able to observe the program's behavior both externally via graphics display and textual messages, and internally—via Prolog execution trace. We demonstrate its correct knowledge through a set of test cases corresponding to the user faults.

The program generally supports both "how-to-do-it" and "how-does-it-work" kinds of advice, it is not tuned to one particular kind of help. Performance of the program has two aspects: its functional and its time characteristics. Functionally, we expect to observe increased number of rules and improved troubleshooting procedures. Our understanding about interaction, together with a short review of some prevoius relevant work, is laid out in Section 2. An interface, implementing

our ideas of interaction, is developed by efficiently combining knowledge and metaknowledge in Section 3.

2. INTERACTIVE APPROACH

A computer program is considered interactive when it requires some user participation in order to conclude its processing. Diagnostic expert systems are obviously interactive systems both at the knowledge acquisition and at the diagnosis phases. The degree of interactivity, the amount of freedom that the user is allowed varies depending on the skill of the user. If the user is a programmer, then an editor is sufficient. Otherwise, more sophisticated tools are required. Some guidelines for constructing human-computer interfaces have been published by DOD [12–15], but they do not provide the necessary information for determining the effectiveness of specific interfaces and their sophistication. As a rule of thumb, the program sophistication level is directly proportional to the user's familiarity with the computer programming techniques.

SOPHIE (SOPHisticated Instructional Environment) is probably the best example of an intelligent interactive computer-based instructional system used for teaching electronics troubleshooting [16]. The system can be used in two modes: a team troubleshooting game and an interaction with an expert. In the game mode the players of one team inflict faults into a simulated electronics system and troubleshoot the simulated faults of the other team. SOPHIE is an example of an experimentally implemented *learning-by-doing* environment with a taxonomy of user errors and an advice feedback mechanism to the user. Unfortunately, the system lacks graphic interface, does not handle significantly complex electronics equipment, and does not evaluate troubleshooting strategies.

2.1. Graphic Interaction

Interactive graphics systems require both the inputs and outputs be specified graphically. Graphics editing of the knowledge base has been proposed recently [17–19] as interaction means with the knowledge base. An expertise transfer system (ETS) [20] has been proposed to generate the rules from the user-supplied elements and their attributes and corresponding ratings. The system identifies conflicts and ambiguities in the rules (but not omissions), and asks the expert for modifications. Since ETS does not use a model of the unit under test, numerous similar rules may need to be entered (e.g. pertaining to the same faults on different channels) and at the same time ETS may still not notice some of the missing rules.

SandKAST (Sandia Knowledge Acquisition System, Hill *et al.* [19]) uses a directed acyclic graph structure as a formalism for knowledge acquisition. In this graph, a node represents a state of the troubleshooting session with an associated set of possible faults. An arc represents a test to be applied in the context of the source node. The graph structure provides the means for viewing the knowledge acquisition and truobleshooting processes as well as for graphic editing of the knowledge base. SandKAST, although alleviates most of the problems that arise with knowledge base maintenance, runs only in the KEE environment on a Symbolics computer and relies heavily on KEEs graphics utilities and object orientation.

IMPULSE [18] permits the user to interact with the knowledge base via various multiple windows and graphic displays of the knowledge. Editing, however, is allowed only at the textual level and not graphically. The user of sophisticated graphics interfaces in computer aided design, simulation, or instruction systems has been also suggested in STEAMER [21], Omega [22], Pecan [23], Garden [24], Wlisp [25], PV [26], PAW [27], Thing Lab [28] and GUIDON-WATCH [17]. All of the above systems emphasize and make use of the abilities of the brain to detect spatial patterns and reason upon them [29].

2.2. Flexible Interaction

Few of the above mentioned systems, however, call attention to the questions of evaluating advisory strategies and adapting them, and all leave those questions open. Consequently, all of the above systems lack the flexibility of adjusting the user interface to the sophistication level of the user. As a result, the systems are either too complex at the initial stages or too detailed at

the subsequent stages. In any case the users becomes annoyed with the system and frequently stop using it.

The concept of an *adaptive* interface [30–32] is an extension of an interactive user interface idea. An interface system becomes an adaptive system in two ways: active and passive. The *passive* way allows users to modify the interface, so it is tailored to meet the specific users needs. Although resulting in a more suitable interface, the burden of its adapting is left to the user. An *active* interface modifies itself. Architecture of an active interface [33–35] addresses explicitly the issues of dialogue [36, 37] and user modeling [38, 39]. McR is an active user interface.

Crockford [40] identified the *user involvement* as the most important principle in the design of interactive programs. He characterized user involvement as having "more to do with taking part than in making decision". The user choices must affect the presentation. The user is viewed as a part of the program. Following this approach, our system maintains a *model of the user*. According to the user-model, the sytem defines the kind of data and rules to operate on. Such an approach allows the user to enter incomplete specifications of the problem and let the system make knowledge-based interpretations of user intentions. Thus, instead of a one-directional interface at a single level of complexity, the system *interface is flexible*.

Consider, for example, the arcade games. Depending on the number of accumulated points, the speed and the complexity of the game increases. But, contrary to arcade games, where the options given to the user to input into the system remain fixed, we require the user options to be dynamic and fit the user needs. The user model is constantly reevaluated by *challenging* the user at every step. Such a dynamic approach not only keeps the program to be updated about the level of the user's proficiency, but it also allows some doubt in the user's mind about the outcome of the interaction. Therefore, the user remains interested in the interaction. The final diagnosis then is the "happy end" which reinforces the user's interest in using the program.

Recently Fischer *et al.* [41] reported about the need for combining the advice-giving strategies. They distinguish between *active* strategies, where the system provides advice by interrupting the dialogue, and *passive* strategies, where the user must explicitly ask for advice. Passive strategies usually employ a *Socratic style* of interaction where the system poses questions and the user is expected to provide answers. Such systems often patronize the users, treating them as the information providing tools, needed only to conclude the reasoning by the computer. Active strategies often employ *learning-by-doing* environments, where user's actions are compared with the ideal actions and feedback is provided to improve the user's responses towards expert prototype. McR combines the strategies of interaction.

2.3. Adaptive Interface Architecture

Knowledge-based adaptive interfaces include four kinds of knowledge [36, 42, 43]: (a) user model; (b) interaction and dialogue management; (c) knowledge of the task and (d) system characteristics. In the next section we describe user model and interaction management.

2.3.1. User model

Kass and Finin [44] claim that individualized user models are essential for good explanations when the users differ in their knowledge of the domain. Loosely speaking, they perceive the user model as a knowledge source containing explicit assumptions on all aspects of the user that may be relevant for the interaction of the system with the user. Any interactive system has a user model. However, most systems maintain an implict user model because of the assumptions about the user made during system design. Representing user model explicitly allows to maintain it dynamically during the program execution, and thus to achieve the desired level of system interface flexibility (cf. Coombs and Alty [45], who stored error patterns as a normative user models for users of Prolog).

The simplest technique for building user models involves *classifying* users as novices and updating their status as they demonstrate improvements [46]. A more discriminating technique, allowing more efficient teaching, involves *comparing* the user's performance with that of the expert. It is assumed that the user knows about the underlying concept, if its derivative is used correctly [47, 48]. On the other hand, by classifying the errors that are made by the user, the underlying deficiencies may be uncovered. The most sophisticated technique is the *stereotype* user modeling

Table 1. Typical analyst errors and corresponding user stereotypes

Errors	Interpretation
Unnecessary measurement	Lack of plan
Wrong measurement	Misunderstanding of schematic, lack of skill to use equipment
Wrong conclusion	Misunderstanding of schematic
Early conclusion	Laziness
Late conclusion	Lack of self-confidence
No conclusion	Lack of persistence
Lack of technological suggestions	"Ivory tower" complex, lack of understanding of manufacturing process

which involves describing the user by a set of characteristics [49]. Examples of such systems include a bibliographical system GRUNDY [39], and a real estate recommendation system [50]. McR uses a troubleshooter's model which is a combination of stereotyping and user error classification techniques.

The user model is useful during the troubleshooting session to select the way in which a troubleshooting advice is constructed and displayed. Once such an error-stereotype table (Table 1) is constructed, the troubleshooters can be ranked, depending on a linear combination of their scores in the table.

When a novice analyst is interacting with the program, the entire is displayed, including the component location, description of the test equipment used, measurement procedure and the meaning of the readouts. As the user becomes more proficient in the use of the system, such a detailed display becomes annoying. A part of the advice is now sufficient. At the next level of proficiency, it is enough to display just the location of the measured component instead of the complete advice. This information now can be supplemented by the display of the relevant part of the fault tree. Finally, for the expert, it is enough to display just the list of suspected components and a scaled down fault tree. We notice that, as the user sophistication increases, we may condense more and more information at the increasingly abstract levels.

We note here that in order to diagnose correctly the user's errors, it is not sufficient to observe the last user's action. The trace of the entire process that the user traversed in arriving to the current situation is needed in order to make a correct conclusion about user's errors [51–53]. For example, if a user, troubleshooting a signal path, consisting of the components C1, C2, C3, C4, C5 in that order, and having observed good input to C1 and bad output from C4, measures the output from C5, then the user most likely misunderstood the signal path. On the other hand, if the user measures the input to C2 then it is most likely that she does not have a good troubleshooting plan.

2.4. Logic Programming Implementation

Before continuing with the discussion about user modeling and interaction, we digress briefly to highlight several points on system implementation.

Our graphic interactive interface has been constructed by efficiently combining knowledge and metaknowledge. Aiello *et al.* [54] describe three approaches of embedding metaknowledge in a system. The most primitive approach is to "*hardwire*" metaknowledge by simply writing it as pieces of code in the system. Such an approach results in extending the implementation of the system with the procedures that actually instantiate the variables of metaknowledge. An example of such an approach is the early rule-based expert systems where each separate case has to be covered by a specific rule instance.

A second approach to combine knowledge and metaknowledge is the *metalanguage* approach as in ML [55, 56]. However, using this approach prevents the higher levels of metareasoning. A third approach allows the user to access both the object and the metalevel simultaneously, as in FOL [57]. This approach requires that both the language and metalanguage have the same form of expression. Another requirement in the third approach is that both levels have access to the inference engine, allowing for proving both theorems and metatheorems. The user is referred to Ref. [54] for further discussion about the ways of amalgamating knowledge and metaknowledge.

It is suffice to note here that McR implements the *third approach* using the metaprogramming techniques of Prolog. Accordingly, we may state that a subcircuit X is faulty as follows:

```
indict(X, Z):-
                    suspect(X),
                    generate_suspect(X,Suspects),
                    generate_ideas(Suspects,Ideas),
                    try_ideas(Ideas,Suspects,Z).
```

The predicate suspect/1is true if its argument X has good input and bad output (this fact may be acquired from the user). The predicates generate_suspects/2 and generate_ideas/2 are the knowledge base access predicates, where the first is used to subdivide the circuit X into a set of faulty subcircuits, and the second to derive the set of indictment methods associated with the subcircuits.

Here the *object-level fact* about the component Z being the reason for circuit X to fail is derived provided that the *metalevel condition of provability* holds between the set of relevant facts and the goal try-ideas (Ideas,Suspects,Z):

```
try_ideas([Idea|Rest],Suspects,BadGuy):-
                    do_or (Idea,Suspects,BadGuy),
                    try_ideas(Rest,BadGuy).

do_or (A,[X|Y],Z):-
                    T=..[X,A,Z],T,
                    do_or (A,Y,Z).
```

The meta-predicate try_ideas/3 creates and executes (do_or/3) the goals required to indict the suspected subcircuits, as long as there are ways to indict (ideas) and as long as all the subcircuits are not indicted.

Note that a circuit pack can be faulty for different reasons and hence we not not declare X indicted by the following *object-level* sentence:

```
indict(X,Z):- suspect(X),
              idea(Z).
```

The reader is referred to Ref. [3] for further information about the code of the circuit pack diagnostic expert system.

Our interface consists of two basic modules: the test tree manager and the user model manager. Test tree manager consists of the modules to reify the diagnostic tree and to handle the knowledge bases of advice and of signal path and pins in the circuit pack. It also manages the database of relations which define the mapping between the nodes of the test tree and the graphical objects representing the top view of the components [58]. The user model manager maintains a counter representing the level of the user's proficiency and the conditions for achieving the next level of proficiency. When the value of the counter exceeds a preset number of allowable errors, the system issues the corresponding advice.

2.5. Adaptive Troubleshooting Interface

Any user modeling system must provide the following important functions: user model representation scheme, interface between the user model and the rest of the expert system, and user model acquisition. Our computational paradigm for implementing intelligent user interfaces is a loop consisting of the following four activities: monitor, abstract, compare, and react. Therefore, McR consists of a monitor to acquire knowledge about the user, a fault identifier to map from user actions to user descriptions, and a proficiency table to map from user model to advice level. The monitor simply presents the user with the top view of the circuit pack and accepts from the user the information about the troubleshooting session. This information contains the location of the measurements, and their results.

The user fault identifier matches the above information to one of the diagnostic trees developed at the knowledge acquisition stage. When a significant discrepancy is identified, the user is offered

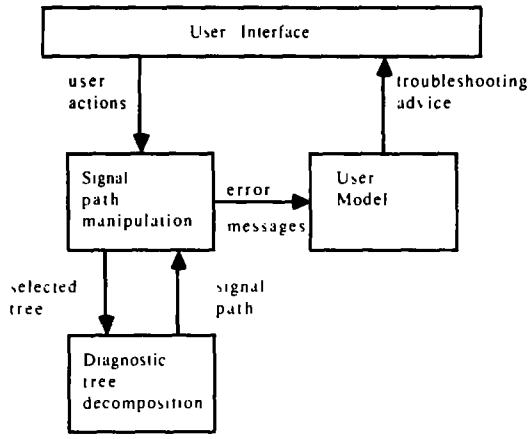

Fig. 1. Adaptive troubleshooting knowledge base schematic.

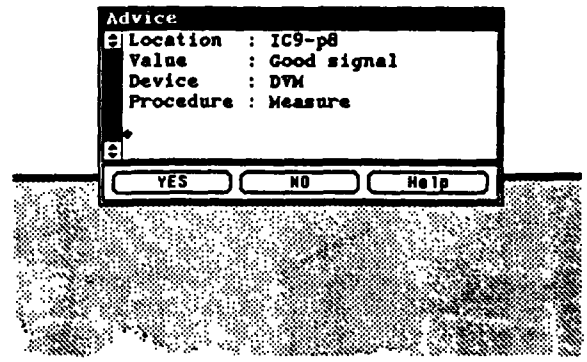

Fig. 2. Advice at elementary level.

help and the proficiency counter is decremented. Table 1 in Section 3.3 lists possible user faults. We differentiate between three basic kinds of troubleshooting errors: misunderstanding of schematic, misunderstanding of troubleshooting strategy and misinterpretation of the measurement. The proficiency level is matched also for the appropriate level of abstraction which should be used when presenting information to the user. Low proficiency level corresponds to high level of detail and vice versa. The schematic and measurement related errors can be identified by comparing the true signal path with the signal path abstracted from user measurement sequence observations. The abstractions rules are presented in Section 3.6. The troubleshooting strategy related errors can be identified by comparing the measurement sequence with the ideal troubleshooting tree.

Using a rule-based knowledge representation, we may subdivide our system into a hierarchy of rules (Fig. 1). At the first level are the rules which allow to decompose the knowledge base into a sequence of components corresponding to the path of signal flow in the unit under test. At the second level are the rules which use the user supplied measurements to manipulate the sequence of components obtained at the first level. Every manipulation triggers a rule in the third set of rules which describe the human troubleshooters behavior patterns. Every rule in the third set may issue a troubleshooting advice.

2.6. Diagnostic Tree Decomposition Rules

The diagnostic tree decomposition rules are used to generate an abstract signal path which will serve as a model for the actual signal path. This model serves the purpose of monitoring and analyzing in a convenient way the sequence of actions of the human troubleshooter.

Rule 2.6.1

If current diagnostic rule has subrules
then invoke Rule 2.6.1 for each of the subrules
and collect the current rule Id in the abstract signal path
and invoke Rules 2.6.2 and 2.6.3.

Rule 2.6.2

If current diagnostic rule is a replace rule
then tag the corresponding entry in the signal path as a replace entry.

Rule 2.6.3

If current diagnostic rule is a test rule
then tag the corresponding entry in the signal path as a test entry.

Note. The resulting signal path may contain non-unique elements which are ignored by Rules 2.6.2 and 2.6.3.

2.7. Abstract Signal Path Manipulation Rules

The abstract signal path manipulation rules maintain the current status of the model derived by the previous set of rules. The model is updated either by deleting its parts which become obsolete due to the results of test observations, or by replacing it by a new model due to a charge in the troubleshooting strategy.

Rule 2.7.1

If the outcome of the test is good
then delete from the current abstract signal path all upstream entries.

Rule 2.7.2

If the outcome of the test is bad
then delete from the current abstract signal path all downstream entries.

Rule 2.7.3

If current test point is the root of the diagnostic tree
then select new diagnostic tree

2.8. User Modeling Rules

The last set of rules actually selects the appropriate troubleshooting advice. These rules are used to decide whether to dispense a high level advice due to a minor troubleshooting strategy mistake, or to advice at a more detailed level. Such an advice is given because of a significant lack of circuit pack understanding on the part of the human troubleshooter discovered by one of the rules.

Rule 2.8.1

If current test point does not belong to abstract signal path
then display the signal path.

Rule 2.8.2

If current test point differs from the root of the current diagnostic tree
then display the diagnostic tree.

Rule 2.8.3

If activated Rules 2.8.1 or 2.8.2 more than three times
then disperse a conventional troubleshooting advice.

2.9. Intelligent Interaction Example

The following paragraphs illustrate the different kinds of advice the system can issue depending on the user. The most detailed advice which would be given to a notice user, is shown in Fig. 2. In this mode the system advices the user on *what-to-do* by telling her the location to be tested, the value to be expected, the measuring device to use and the procedure to be followed for making the measurement. This advice would be given to a user who does not have a troubleshooting strategy and whose proficiency counter value is low.

The next advanced level of advice, which would be given to a troubleshooter who has a basic understanding of the test set, various measuring devices, and the circuit pack in this mode, is shown in Fig. 3. Now the system displays the entire diagnostic tree which was constructed based on good troubleshooting strategies (e.g. "divide and conquer") for this particular pack. The system identifies the user as one who doesn't have a good troubleshooting plan and who might need a longer time to diagnose a fault. The tree displays only the location to be tested.

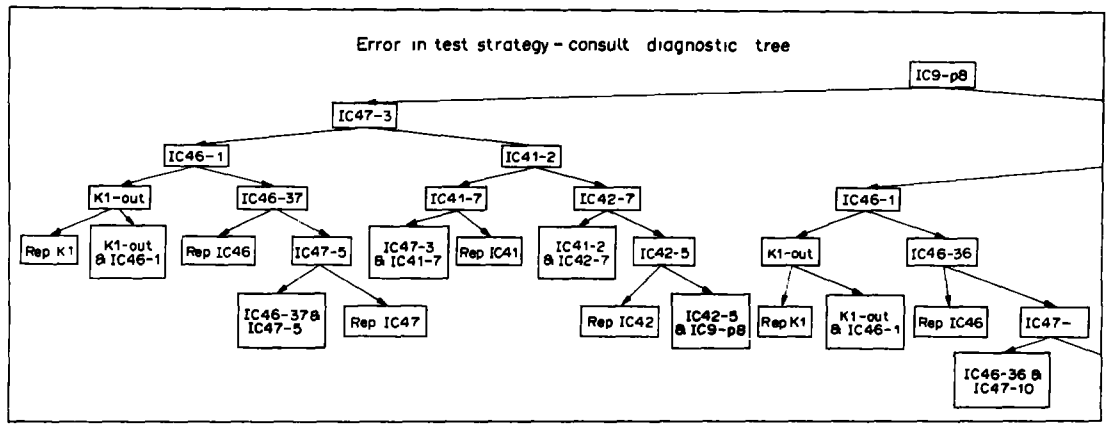

Fig. 3. A more advanced advice.

Finally, the most advanced level of advice is shown in Fig. 4. Such an advice would be given to an experienced troubleshooter. In this mode the system gives an advice on *how-does-it-work*, by displaying the signal path. The user is responsible for the troubleshooting strategy and also for completing any necessary details.

3. DISCUSSION

Since consultation interrupts working, advisory systems, both human and computerized, are frequently not welcome. Intelligent interaction deploys computerized advisory capabilities by merging working and learning modes. In this paper we explore the user modeling technique to implement intelligent interaction. If explanation—communicating knowledge to the user by the program—is viewed as a process of human knowledge acquisition, then a user model must be maintained by the program as a presumed human knowledge representation scheme.

Carroll and Aaronson [2] showed how it is possible both to frustrate and to help people by providing "intelligent" help. We are dealing with the question of how to deploy a less than perfect advisory capability without having a sound theoretical ground. We believe that a flexible interaction environment based on the human model, maintained by the computer program can alleviate some of the user frustrations. The flexibility of the interaction is achieved by allowing to query the advisory knowledge base at different levels of detail, depending on the level of user sophistication.

To obtain realistic experience and insight we chose to work in the area of engineering, mental, and economic significance: circuit-pack diagnostics. In this paper we describe the technical aspects of what we perceive to be an intelligent user interface to an advisory system of sufficiently economic impact. McR is a knowledge-based interactive graphic interface to a circuit pack diagnostic expert system.

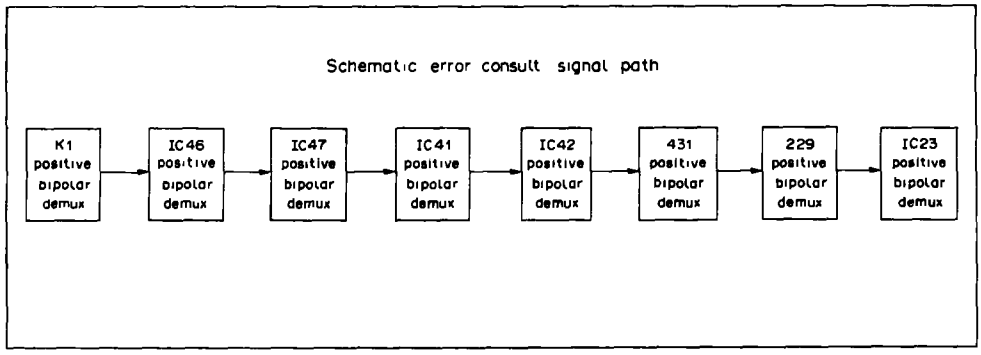

Fig. 4. The most advanced and the least detailed advice.

McR integrates three kinds of knowledge: domain knowledge (i.e. circuit pack), troubleshooting knowledge (i.e. diagnostic trees) and user knowledge (i.e. stereotype table). McR is an active interface which dynamically changes the amount of detail presented to the user as well as the input choices that the user is allowed to make. These changes are made using a knowledge-based model of the user and of the circuit pack troubleshooting domain. McR combines the strategies of interaction: it is able to advise both on "how-to-do-it" and on "how-does-it-work". While "how-to-do-it" advice is concerned with the measurement procedures [e.g. instrument (scope, etc.) and location (ic, pin)], the "how-does-it-work" deals with the deeper knowledge representing the flow of the signal and the sequence of measurements. We have demonstrated that the analysis of user actions requires disassembly of observations along the different kinds of knowledge. Thus we develop special knowledge abstraction algorithms along with reification algorithms. The resulting system, instead of guiding the user by querying for input, monitors users actions, and offers help when needed. McR is used in conjunction with the STAREX expert system which is currently installed at an AT&T factory.

Methodologically, we have shown a deep relationship between the problems of user interface construction, the problem of knowledge acquisition, and the problem of efficient diagnostic reasoning. All of these problems involve optimal selection of the sequences of measurements, and in fact, all of our implementations use the same software module for this purpose [5]. We have also proposed the basic intelligent interaction paradigm to be a loop, consisting of monitor, abstract, compare, and react activities (the resulting system, McR, demonstrates its software implementability and its name is a mnemonic to the paradigm). Additionally, we have demonstrated that metaprogramming techniques have great potential in implementing intelligent interaction systems. McR, in particular, has been developed entirely in Prolog—a logic programming language with an especially convenient environment to develop metainterpreters.

An important area for future research is that of improving a communication backchannel to allow the user to respond to the help in a less constrained way. Another possible way to improve our system is to develop an automated user activities monitoring system. We also foresee future intelligent advisors being able to perform "*what if*" analysis of user actions. A way to do such a task is to substitute the "correct" parts of the model with the assumed user actions and results, reify the knowledge base and compare the resulting tree with the ideal one. Additionally, we plan to incorporate the truth maintenance mechanisms in intelligent user interfaces. And finally, we foresee using intelligent user interfaces for knowledge acquisition.

Acknowledgements—We gratefully acknowledge the support of Walt Lawrence who made many invaluable comments and suggestions; On-Ching Yue for guidance and insight, and Leon Levy for patient reading and comments.

REFERENCES

1. J. Carroll and J. McKendree, Interface design issues for advice-giving expert systems. *Commun. ACM* **30**(1), 14–31 (1987).
2. J. Carroll and A. Aaronson, Learning by doing with simulated intelligent help. *Commun. ACM*, **31**(9), 1064–1079, (1988)
3. Y. Lirov, STAREX—simultaneous test and replace circuit pack troubleshooting expert system prototyping and implementation. *Engng Applic Artif. Intell.* **2**, 3–18 (1989).
4. Y. Lirov, Circuit pack diagnostic expert systems—a survey. *Computers Math. Applic.* **18**(4), 381–398 (1989).
5. Y. Lirov and O. Yue, Circuit pack troubleshooting via semantic control I: goal selection. *Proc. IEEE Wkshp Artificial Intelligence for Industrial Application*, Hitachi, Japan, pp. 118–122 (1988).
6. N. Morris and W. Rouse, Review and evaluation of empirical research in troubleshooting. *Human Factors* **27**(5), 503–530 (1985)
7. L. Hyafil and R. L. Rivest, Constructing optimal binary decision trees is NP-complete. *Inf. Process Lett.* **51**, 15–17 (1976)
8. O. H. Ibarra and S. Sahni, Polynomially complete fault detection problems. *IEEE Trans. Comput.* **C-24**(3), 242–250 (1976).
9. J. L. Saupe, Troubleshooting electronic equipment: an empirical approach to the identification of certain requirements of a maintenance occupation. Ph.D. Dissertation, University of Illinois (1954).
10. R. Glaser and J. Phillips, An analysis of proficiency for guided missile personnel: III, patterns of troubleshooting behavior. Technical Bulletin, 55-16, American Institute for Research, Washington D.C., Aug. (1954).
11. C. Gettys, C. Manning, T. Mehle and S. Fisher, Hypothesis generation: a final report of three years of research. *Technical Report* 15-10-80. Decision Process Laboratory, University of Oklahoma, Norman, Okla, Oct. (1980).
12. Human engineering criteria for military systems, equipment, and facilities. Report MIL-STD 1472C, Department of Defense, Washington, D.C. (1981).

13. W. Buxton, M. R. Lamb, D. Sherman and K. C. Smith, Towards a comprehensive user interface management system. *Comput. Graph.* **17**(3), 35–42 (1983).
14. S. L. Smith and Y. N. Mosier, Design guidelines for the user-system interface software; The Mitre Corporation Technical Report ESD-IR-48-190, Bedford, Mass. (1984).
15. A. F. Norcio and J. Stanley, Adaptive human–computer interfaces. Technical Report #NRL-9148, Naval Research Laboratory, Washington, D.C., Sept. (1988).
16. J. Brown, R. Burton and J. deKleer, Pedagogical, natural language, and knowledge engineering techniques in SOPHIE I, II and III. In *Intelligent Tutoring Systems* (Eds D. Sleeman and J. Brown), pp. 227–282. Academic Press, New York (1982).
17. M. H. Richer and W. J. Clancey, GUIDON-WATCH: a graphic interface for viewing a knowledge based system. Technical Report, STAN-CS-85-1068, Stanford University, Calif. Aug. (1985).
18. E. Schoen and R. G. Smith, IMPULSE: a display oriented editor for STROBE. *Proc. AAAI'86*, pp. 356–358 (1986).
19. F. N. Hill, J. D. Ward and A. L. Yates, SandKAST: An automated knowledge acquisition system. SAND-87-0364C, Sandia, Dec. (1987).
20. J. H. Boose, Personal construct theory and the transfer of human expertise. *Proc. AAAI'84* (1984).
21. A. Stevens, B. Roberts and L. Stead, The use of a sophisticated graphics interface in computer-assisted instruction. *IEEE Comput. Graph. Applic.* **Mar./Apr.**, 25–30 (1983).
22. M. L. Powell and M. A. Linton, Visual abstractions in an interactive programming environment. *Proc. ACM SIGPLAN, Sigplan Notices*, Vol. 18, pp. 14–21, Jun. (1983).
23. S. P. Reiss, Graphical program development with PECAN program development systems. *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symp. Practical Software Development Environments*, Apr. (1984), also printed as *Sigplan Not.* **19**(5), 30–41 (1984).
24. S. P. Reiss, A conceptual programming environment. *Proc. 9th Int. Conf. Software Engineering*, Monterey, Calif. Mar/Apr. (1987).
25. C Rathke, Human-computer communication meets software engineering. *Proc. 9th Int. Conf. Software Engineering*, Monterey, Calif. Mar Apr. (1987).
26. G. P. Brown, R. T. Carling, C. F. Herot, D. A. Kramlich and P. Souza, Program visualization: graphical support for software development. *Computer* **Aug.**, 27–34 (1985).
27. B. Melamed and R. J T. Morris, Visual simulation: the performance analysis workstation. *Computer.* **Aug.** 87–94 (1985).
28. A. Borning, The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Trans. Progrng Lang. Syst.* **3**(4), 353–387 (1981).
29. T. Dudley, Graphics in software design. *Computers. Graph. Wld* **Feb.** 35–42 (1986).
30. S. Greenberg and L. H. Witten, Adaptive personalized interfaces—a question of viability. *Behavior Inf. Tech.* **4**(1), 31–45 (1985).
31. E. A. Edmonds, Adaptive man–computer interfaces. In *Computing Skills and the User Interface* (Eds M. J. Coombs and Y. L. Alty), pp. 389–426. Academic Press, London (1981).
32. M. V. Mason and R. C. Thomas, Experimental adaptive interface. *Inf. Technol. Res. Des. Applic.* **3**(3), 162–167 (1984).
33. W. Sherman, SAUCI: self-adaptive user-computer interfaces. Ph. D. Dissertation, University of Pittsburgh, Pittsburgh, Pa (1986).
34. R. Reichman-Adar, Extended person–machine interface. *Artif. Intell.* **22**, 157–218 (1984).
35. W. B. Rouse, Human-computer interaction in the control of dynamic systems. *Computing Surv.* **13**, 71–100 (1981).
36. E. Risland, Ingredients of intelligent user interfaces *Int. J. Man–Mach. Stud.* **21**, 377–388 (1984).
37. W. Wahlster and A. Kobsa, Dialogue-based user models. *Proc. IEEE* **74**(7), 948–960 (1986).
38. H. Mozeico, A human/computer interface to accommodate user learning stages. *Communs ACM* **25**(2), 100–104 (1982).
39. E. Rich, User modeling via sterotypes *Cogn. Sci.* **3**, 329–354 (1979).
40. D. Crockford, Stand by for fun. In *Interactive Multimedia* (Eds S. Ambron and K. Hooper). Microsoft Press (1988).
41. G. Fischer, A. Lemke and T. Schwab, Knowledge-based help systems. *Proc. CH185 Human Factors in Computing Systems*, San Francisco, Calif., 14–17 Apr. pp. 161–167. ACM, New York (1985).
42. I. Monarch and J. Carbonell, Coal SORT: A knowledge-based interface. *IEEE Expert* **2**(1), 39–53 (1987).
43. W. B. Croft, The role of context and adaptation in user interfaces. *Int. J. Man–Mach. Stud.* **21**, 283–292 (1984).
44. R. Kass and T. Finin, The need for user models in generating expert system explanations. *Int. J. Expert. Syst.* (in press).
45. M. J Coombs and J. L. Alty, Expert systems: an alternate paradigm. *Int. J. Man–Mach. Stud.* **20**, 21–43 (1984).
46. D. A. Norman, Design rules based upon analyses of human error. *Commun. ACM* **26**, 254–258 (1983).
47. M. Matz, Towards a process model for high school algebra errors. In *Intelligent Tutoring Systems* (Eds. D. Sleeman and J. S. Burton). Academic Press, New York (1982).
48. R. Burton and J. S. Brown, A tutoring and student modeling paradigm for gaming environments. *Proc. ACM SIGCSE/SIGCUE Joint Symp.* (1976).
49. T. Finn and D. Darger, GUMS: A general user modeling system. University of Pennsylvania School of Engineering and Applied Science, Technical Report MS-CIS-86-35, May (1986).
50. K. Morik and C. Pollinger, The real estate agent—modeling users by uncertain reasoning. *AI Mag.*, pp. 44–52 (1985).
51. J. J. Allen and C. R. Perrault, Analyzing intentions in utterances. *Artif. Intell.* **15**, 143–178 (1980).
52. R Burton and J. S. Brown, An investigation of computer coaching for informal learning activities. In *Intelligent Tutoring Systems* (Eds. D. Sleeman and J. S. Brown), pp. 137–155. Academic Press, New York (1982).
53. M. R. Genesereth, The role of plans in intelligent teaching systems. in *Intelligent Tutoring Systems* (Eds D. Sleeman and J. S. Brown), pp. 137–155. Academic Press, New York (1982).
54. L. Aiello, C. Cecchi and D. Sartini, Representation and Use of Metaknowledge. *Proc. IEEE* **74**(10), 1304–1321 (1986).
55. M. Gordon, R. Milner and C. Wadsworth, Edinburg LCF: a mechanized logic for computation. *Lecture Notes Computer Science 78.* Springer, New York (1979).
56. A. Wikstrom, *Functional Programming Using Standard ML.* Prentice-Hall, Englewood Cliffs, N.J. (1987).
57. R. Weyhranch, Prolegomena to a theory of mechanized formal reasoning. *Al J.* **13**, 133–170 (1980).
58. Y. Lirov, Computer aided software engineering of expert systems. *Expert Systems with Applications.* Pergamon Press, Oxford (in press).