

# EXPERT SYSTEMS: THE USER INTERFACE

edited by

**James A. Hendler**

*University of Maryland*

*Copyright 1988*



ABLEX PUBLISHING CORPORATION  
Norwood, New Jersey 07648

## FOUR

# DARN: Toward a Community Memory for Diagnosis and Repair Tasks

**Sanjay Mittal,  
Daniel G. Bobrow,  
Johan de Kleer**

*Intelligent Systems Laboratory  
Xerox Palo Alto Research Center  
Palo Alto, CA*

### **Abstract**

*DARN is a plan-based knowledge system designed to aid an inexperienced technician in a diagnosis and repair task. Our approach to building this knowledge system is contrasted with theory-based systems such as Sophie, or a classification-based system such as MDX, using a characterization of the tasks, and the leverage for the different approaches. DARN provides a compact and convenient representation for capturing knowledge available in the community of experts for this task. Graphic interfaces are used to guide rookie technicians in the repair task. DARN also provides an "expert's interface" that allows members of the community to extend and modify the knowledge base without intervention of "knowledge engineers."*

### **I. Introduction**

DARN is a knowledge-based system to aid in a diagnosis and repair task. In this task, the problem is to localize the cause of a malfunction sufficiently to allow an action that will repair the faulty artifact. Traditional approaches to the development of knowledge-based trou-

bleshooting systems have been based on deep models of the artifact or on a classification of ways in which the artifact can malfunction. In this chapter we identify a third approach, called plan-based, which may be more suitable for some systems. The plan-based approach derives its power from plans for debugging developed by a set of experts, and *allowing them to formalize, extend, and propagate their community knowledge base*. We have focused on the development of a representation to express this diagnostic and repair experience. We support the use of this representation with an interface to enable experts to create and modify the knowledge bases without the mediation of a knowledge engineer.

A spectrum of approaches are possible for the debugging task depending on characteristics of the artifact being debugged. To understand when a plan-based approach is useful, we look at sources of leverage in the computer programs based on existing approaches, and the assumptions made by these approaches about the nature of the artifact being debugged.

*Theory-based systems* derive their leverage from use of a number of domain-independent reasoning mechanisms operating on large amounts of domain-specific knowledge. For these systems, one must build models of their structure and function. It is then possible to reason from differences between their modeled and actual behavior to locate the cause of the malfunction. SOPHIE (Brown, Burton, & de Kleer, 1982) and DART (Genesereth, 1982) are programs that typify this approach. These programs derive their power from general-purpose reasoning methods that include dependency-directed backtracking (de Kleer et al., 1979; Sussman & Stallman, 1975) and envisioning (de Kleer, 1984). These programs can reason about behavior based on descriptions of the artifact. In order for this approach to succeed, one must assume, of course, that the artifact can be completely described. This is true for circuits analyzed by SOPHIE but is often not true for electro-mechanical systems or hardware-software systems.

*Classification-based systems* derive their leverage by classifying malfunctions in equivalence classes based on effective actions to ameliorate the malfunction. Medicine is a typical domain in which this approach is used; although they may wish to know more, it is often sufficient for doctors to find differential diagnosis on which suitable therapeutic measures can be devised. MDX (Chandrasekaran & Mittal, 1982), a computer program for medical diagnosis, organized knowledge about liver disorders as a hierarchy of diagnostic states. Szolovits and Pauker (1978) have argued that even rule-based diagnostic systems such as MYCIN are really performing such classification. These sys-

tems derive their power from appropriately structured diagnostic hierarchies and employ a more special purpose problem-solving technique called heuristic classification (Chandrasekaran, 1984; Clancey, 1985).

A major cost in the development of classification-based diagnostic programs is the creation and maintenance of suitable hierarchies. One reason for this is that such hierarchies have often not been made explicit by practitioners, though the practitioner's behaviors may be described that way. Even in medicine, where there are many extant ways to classify any given set of diseases, it is clear that none of the existing classifications are a suitable basis for organizing diagnostic knowledge in this way. A medical diagnosis hierarchy has to be crafted from pieces of anatomical, physiological, biochemical, and other views (see Mittal, 1980, for an extended discussion of this issue). It would seem, therefore, that a classification approach would be pragmatically useful only for systems which have a long lifespan of interest over which they stay relatively stable. Some other suitable examples would seem to be nuclear power plants, the space shuttle, automobiles, and Boeing 707 airplanes.

*Plan-based systems* for diagnosis and repair are based on capturing the plans of experienced technicians for debugging an artifact. They capitalize on the incremental nature of the knowledge acquisition process and the combination of modeling and testing knowledge that the technicians have. It seems most useful for complex systems containing a number of different kinds of subsystems, for example, electrical, mechanical, and software, that have alternative implementations. Examples are computers, printers, copiers, modern automobiles, and maybe even semiconductor fabrication processes. Complications often arise from the short lifespan of these systems, which is often caused by obsolescence. These systems can be further characterized by unavailability of complete underlying models, and complex interaction between the different subsystems. The rapid advancement in technology is causing a maintenance nightmare in many areas. The short lifespan of an artifact prevents the development of a sustained body of experience in troubleshooting these systems, often a precondition for the development of classification of malfunctions.

We believe that the development of knowledge-based tools for assisting in the maintenance of this latter class of artifacts requires a shift in paradigm away from reasoning and deep knowledge structures. A more suitable paradigm is suggested by the metaphors of a knowledge medium (Stefik, 1986) or a community memory. The basic idea is that one should provide a framework in which the experts can themselves articulate their relevant knowledge precisely enough that it can then be

used by the computer not only to aid in solving problems in the domain but also made subject to peer review and revision. Such a community memory may, over time, integrate the knowledge from many experts.

In this paper we describe one experiment towards better understanding how to build such community memories. The problem domain for our study was the diagnosis and repair of a class of personal computer at Xerox. As part of this experiment we also developed a knowledge-based system called DARN (Diagnosis and Repair Network). This chapter is organized as follows: We start by reporting some observations about the problem domain and the current practice of repairing these computers. In the next section, we present a relatively simple representation which can be used to encode a large fraction of the experiential plans and knowledge of the computer technicians. Next, we discuss some of the user interface issues in enabling a community of users to themselves create and modify a knowledge base. We also describe some of the user interface tools that were implemented as part of the DARN system. The concluding section summarizes our experience in trying to use DARN, including a follow-up experiment in trying to adapt the DARN framework for the repair of copiers.

## II. Observations About the Repair of Personal Computers

Most personal computers, regardless of size or price, are very complex electro-mechanical systems. While our observations here are based on the Xerox 8000 (Star or Dandelion) series machines, many of them are also relevant for other popular series such as the IBM PC or Apple II. A typical Dandelion is configured with a bit-mapped display, a pointing mouse device, a floppy drive, a high-capacity fixed disk, an ethernet connection, and a CPU. We talked to a number of people involved in servicing these machines—technicians in research labs, field service technicians, diagnostic manual writers, and people providing service advice over the phone. The problem of diagnosing and repairing the fixed disk emerged as one of the more challenging and time-consuming problems. We have focused primarily on the disk subsystem in the work reported here.

A fixed disk has electronic (control circuitry), mechanical (disk drive, platters), and electrical (power supply, cooling fan cables) components which can interact in a complex way, especially when there are problems. The problem of how to diagnose and repair problems relating to a malfunctioning disk system in a personal computer has a number of interesting aspects. First, it is not just a diagnosis problem; one is required to find actions which restore the disk to a state where it

---

can continue to be used. This requirement raises interesting issues because of the interplay of hardware and software models. The diagnostic procedures are often programs whose outputs do not precisely indicate a single hardware fault. Moreover, many of the diagnosed problems are *soft* in nature e.g., garbled data fields on a disk, which have been caused by hardware malfunctions. However, these soft faults often cannot be diagnosed until the malfunctioning hardware has been fixed. This interplay presents an interesting challenge in representing the interaction between hardware and software during diagnosis and repair.

Second, there are situations in which there are multiple faults in the machine, and where repairs can fail because newly replaced parts are faulty. In general this is a hard problem. However, there seem to be interesting heuristics for dealing with the more common cases. Finally, the disk repair problem is complicated by the tension between competing constraints: one wants to minimize the amount of hardware replaced on the machine, minimize the amount of time the machine is unavailable to the user, and minimize loss of data on the current disk. The problem of loss of data is often critical when fixing disks.

### **Current Practice**

We started out by taking some very detailed protocols from a couple of the expert technicians of their diagnosis and repair strategies when faced with a problem with a fixed head disk. It was clear that the experts knew a lot about the computer system—they were not approaching it as just a black box. They had a fairly complete model of the architecture (which subsystem was connected to which one and how) and knew how the various subsystems functioned, at least at some level. They also had some partial theories about how the various diagnostic programs worked.

At the same time it was striking to observe how much their ability to troubleshoot the machines was limited by the availability of the diagnostic tools. For example, the diagnostic programs have limited coverage, perform tests in inflexible sequences, and often make unarticulated assumptions about the state of the computer system. Similarly, complex electro-mechanical systems, such as a disk, can malfunction in numerous ways, only a small fraction of which can be directly observed. The real issue, we believe, is not really one of devising better diagnostic procedures, which is always possible, but of using the existing ones to get the job done. In this and many other similar situations the primary objective is making the machine functional again under some of the constraints discussed earlier. We found that our experts

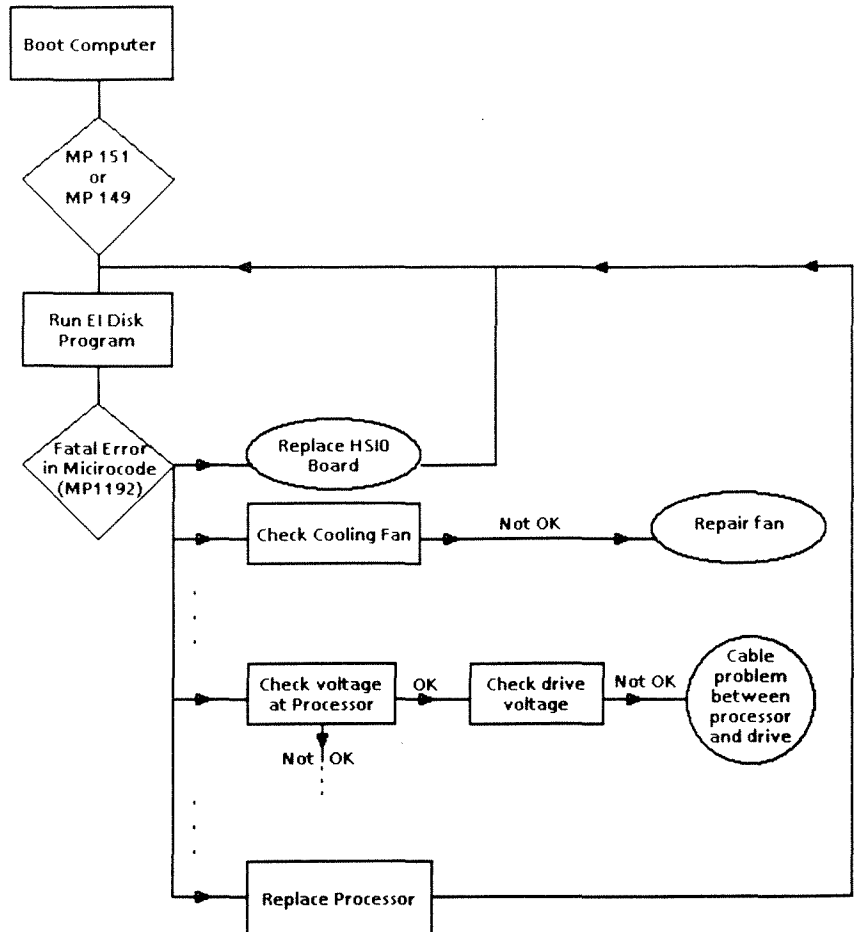


Figure 1. Plan Fragment from Protocols A fragment of a plan to repair one of the fixed-disk malfunctions. The plan is based on the initial protocols from the expert technicians before a computer system was implemented.

were articulating their knowledge in the form of diagnosis and repair plans. *These plans were organized around the typical problems that were encountered by the technicians and encoded their experience of how to go about isolating and fixing the problem.* Figure 1 shows a plan fragment from our initial protocols. This figure encodes the plan for fixing one of the problems that shows up when a user boots the machine. The plan might be read as:

After booting the machine for a cold start, if the maintenance panel (MP) shows 151 or 149, then run EI Disk diagnostic program. If the computer stops with MP code 1192, then execute the following plan. First, try replacing the HSIO (High-Speed I/O board) and rerunning the EI Disk program. If the computer runs fine, then the problem is fixed. Else, check the cooling fan. If the fan is faulty, then fix the fan and rerun the EI Disk program. . . .

We will defer a discussion of these plans to the next section, but a few general comments are in order here. There were many such plans, with each plan being designed to cover a family of problems. As we will see, the plans were not completely independent and often shared subplans. Interestingly, there were variations even among the two experts that we probed in some depth. The plans seemed to compile the experience of and constraints under which the experts were working. It is easy to speculate that the plan variations reflected different experiences and attempts to cope with an imperfect diagnostic situation.

Finally, we had to confront the issue of who would be the users of a computer-based system such as DARN. The experts whose knowledge was encoded in the system would possibly benefit from a precise articulation of their expertise in a form that could aid them in later situations. In the longer term, the experts could share their expertise with other experts, possibly combining their knowledge to form a whole that was larger than the sum of its parts. However, the one clear class of users that could immediately benefit were the so-called "rookie" technicians, i.e., people just being trained to service a new machine.

Currently, inexperienced technicians are provided with manuals of repair that encode Fault Identification Procedures (FIPs). These FIPs suffer from some major problems. First, they are often prepared before any serious experience has accumulated about a machine. Thus, they are incomplete at best and grossly incorrect at worst. Second, while they are occasionally updated, they are usually obsolete before they become available. Furthermore, much of the experience of the expert technicians rarely (or too late) makes its way into the field service manuals. Finally, even though our experts seemed to have a relatively

simple way of expressing their repair plans, these plans, nevertheless, could not be suitably expressed in or used from the medium of printed books. This last point should become clear as we move to a discussion of the plan representation.

### III. Representation of Repair Plans

In this section we describe a language for representing repair plans. Our primary criterion in devising a representation was to ensure that there be a conceptual match between the representation in the machine and the apparent representations used by the technicians. This criterion is itself motivated by the requirement that the domain experts themselves create the knowledge base, modify parts of it, understand the representation well enough to know the implications of their representational choices, and have confidence in the advice provided by a system that uses the knowledge base. Some recent projects such as PIES (Pan & Tenenbaum, 1986), are also exploring similar set of issues.

#### Plan Elements

The representation in DARN is based on the following observations. Most of the repair plans such as the one discussed in the previous section (see Figure 1) have a fairly similar form. Typically, a plan includes starting with some initial *test* on the machine which indicates a malfunction, running some diagnostic tests to pinpoint the problem, applying a *fix* for the problem, verifying that the problem is indeed fixed, otherwise iterating this process.

A plan can thus be viewed as a graph where the nodes can be classified into a fixed number of abstract classes. As a first approximation, we have found that three abstract types—Tests, Observations, and Actions—are sufficient to represent the different elements of the plans. Examples of these from the plan described in the previous section are: *tests* (booting the machine, running EIDisk, checking cooling fan); *observations* (MP151, fatal error microcode, processor voltage OK); and *actions* (replace HSIO board, replace cable).

#### Rules of Composition

The primitive plan elements (graph nodes) are not composed in arbitrary fashion. Instead, there are some well-defined rules that dictate how nodes of a certain class can be linked to nodes of other classes to form legal plans. The basic rules are:

---

A *test* can be connected to *observations* only, representing the results of running a test.

An *observation* can be connected to either *actions* and/or further *tests*. The former case represents actions to be taken in fixing the problem manifested by the observation. The latter case enables further exploration to pinpoint the problem.

An *action* is implicitly connected to a *test* which is used to verify that the action indeed fixed the problem. More complex rules about representing actions are described later.

These rules not only define the plan structure but in effect describe the semantics of the structure as a plan for diagnosing and repairing the target machine. In other words, an element of the plan that is marked as a test node, represents a test to be performed because it is followed by the observations that can be made from running the test. Similarly, an observation node derives its meaning from two structural properties. One, that it follows a test and thus represents a possible result of running the test. Two, it is followed by further actions or tests which represent what should be done with this observed state of the machine in further identifying or fixing the malfunction. *We hypothesize that such a structural transparency is crucial in enabling the experts to manipulate the knowledge base comfortably.*

### Extensions to the Plan Representation

The basic TAO (Test-Action-Observation) framework described above is rather abstract because it is only sufficient in a computational sense. It does not capture many nuances and specializations that the experts use in describing the repair plans. Here we discuss some of the extensions. Most of the extensions can be viewed as specializations of the three abstract classes. Each of the specialized plan element may also specialize the structural rules associated with that class. Figure 2 shows some of the specializations as a class lattice.

### Tests

A top-level test in DARN describes the context in which the user discovered a malfunction. For example, at the top level the system uses the following query as a test to discover the context:

```
Which situation are we in:  
1) Testing a new disk
```

---

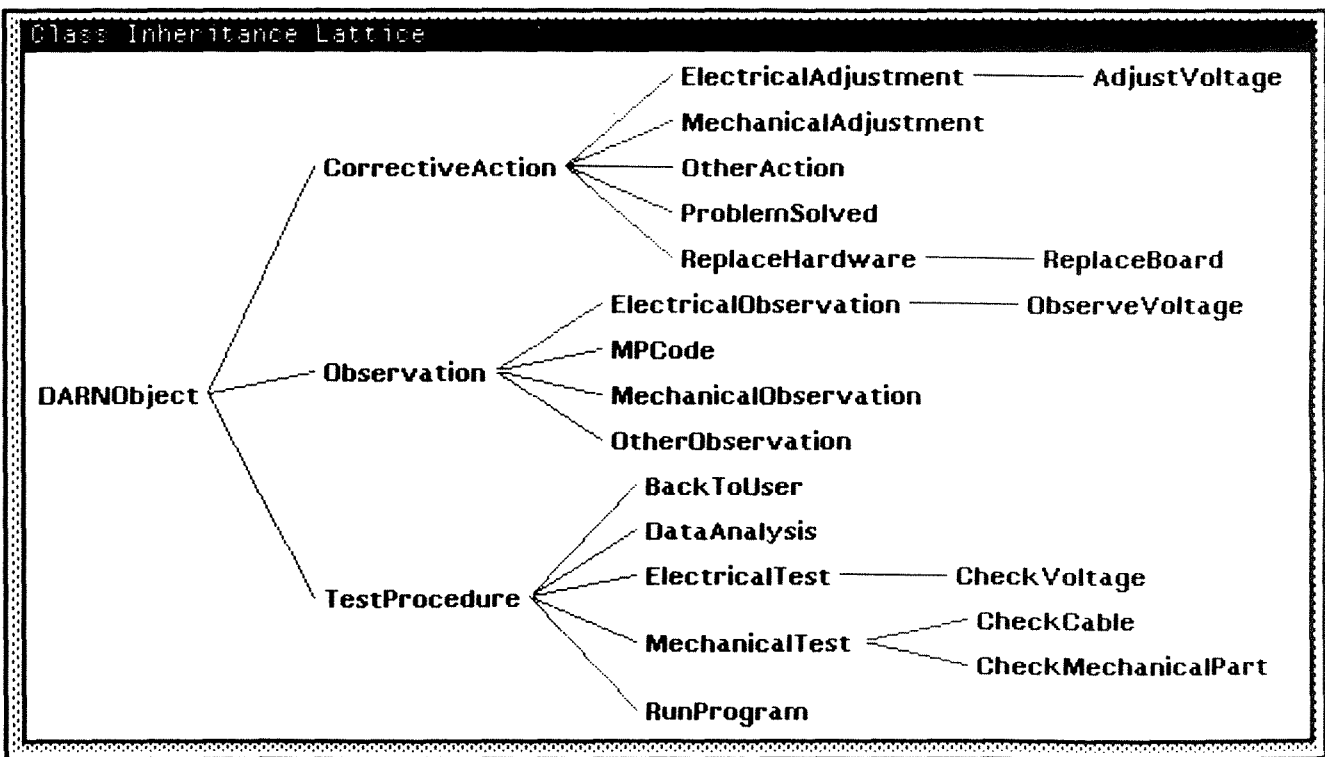


Figure 2. Primitives for Describing Plans. A class lattice showing the ontological primitives for describing repair plans.

- 2) A problem found while booting the machine for a cold start
- 3) A problem found while running user software

Performing a test leads to *observations*, often indications of problems to be fixed. Often some action may be taken to try to "fix" the problem. If the action (or any of its subactions) changes the machine, a test indicating a problem must be run again to verify whether the change affected the problematic observation. For example, suppose we have the test:

```

Check voltage at Processor
None    -ReplacePowerSupply
Not24V  -CheckPowerCable
OK      -CheckVoltageAtDrive

```

If no voltage is found at the processor, then after replacing the power supply, we must check that the new power supply is providing the correct voltage at the processor.

Verifying correct behavior is more crucial when running a diagnostic program, and choosing one of a number of potential repairs for the problem. For example:

```

Select the result of running the EIDisk diagnostic
program:
1) Fatal Error in Microcode
2) Maintenance Panel says 1611-No interface signals
3) . . .
4) No errors

```

If (1) Fatal Error in the Microcode is the symptom reported by EIDisk, the technicians can replace one of several printed circuit boards that could cause this error. After replacement of one, the diagnostic program is run again to test whether replacing that board fixed the problem. If not, the technician will try other possible replacements.

The generic notion of a test is further elaborated by the kinds of tests available in a particular domain. For example, in the disk domain, some of the elaborations of *test* are: *Electrical Test*, *Mechanical Test*, *Diagnostic Program*, *Query User*. Each of these may be further specialized. Similar elaborations exist for *observations* and *actions*. The importance of these domain-dependent elaborations lies in providing the system restrictions on the general rules of interconnections between nodes. For example, the observations of a *diagnostic program*, in our domain, are restricted to *maintenance panel codes (MP Codes)* and something which signals successful completion of the test. These restrictions are exploited in customizing the user interface as discussed

in the next section. Eventually, these elaborations provide a means for attaching deeper models. For example, knowing that the MP Codes are generated in an ascending sequence allows the actual MP Code observed to be used to rule out problems associated with other codes lower in the sequence.

### Observations

There are two classes of observations: ones which indicate successful completion of the associated test and ones which indicate some problem. The former need not be connected to any further nodes, thereby modifying the default interpretation of the associated test. For example, if a test is used for further elucidation of the problem (discussed later in this section), then a successful completion of the test *without any following action* implies that a different action must be taken for fixing the original problem.

### Actions

The kinds of nodes that can follow after an observation may be another *Test*, or actions such as *SimpleRepair*, *FixList* or *CompoundAction*. These nodes are ordered on the basis of the desirability. Desirability is an implicit combination of likelihood of the action specified fixing the problem, the cost of trying the repair, etc. Some *SimpleRepairs* are:

Replacement of a part, e.g., Read Write Board  
Adjustment within the system, e.g., 5=Volt Level  
Software changes to data on disk, e.g., Rewrite Broken Headers

### Fault Isolation

For some tests, the symptom found may not uniquely determine a repair for the problem found. At this point, one possibility is to try to isolate the problem. The simplest option is to just ask the user to narrow the choices (using a *QueryUser* test). Our experts often suggested this course in the intermediate stages of building the knowledge network. In this case, the original *Test* must be used to verify if any action taken fixed the problem. Sometimes a fault may be isolated by performing a secondary test; in this case both the primary and secondary test must be used to verify any fix. Finally, isolation can be done with a *SubTest*; in this new type of *Test*, verification of the *SubTest* is deemed sufficient to verify a fix for the primary *Test*.

---

### Trying Alternatives

In some situations, no tests may be available to isolate the fault. In this case, experts (and the DARN system) suggest an ordered sequence of actions. If any of the actions on this *FixList* lead to a change in the machine, the last test indicating a fault symptom is tried to verify if the change had the desired effect. For example, in running the EIDisk diagnostic, when one gets a fatal error in the microcode, one tries in turn: Replacing the HSIO Board; Checking the Cooling Fan; Replacing the Control Board; . . . ; Replacing the Disk. After trying each action, the EIDisk diagnostic is run again.

The ordering of actions in a *FixList* is dependent on much information not explicit in the model. It depends on the frequency of occurrence of a particular action in fixing a particular symptom, the cost to the technicians of trying the repair, and the ease of performing an action after having taken other actions. We discuss later how such information might be explicitly embedded in the model as annotations for particular purposes.

### Remembering the Context of Fixes

Sometimes the same sequence of potential fixes is applicable in response to different symptoms found by other tests run by the technicians. In each case, the verification condition is determined by the test which manifested the symptom to be fixed. This requires that the DARN interpreter remember the "caller" of the *FixList* in order to use the correct test for verification.

Expert technicians often use previously defined *FixLists* as models for later lists. For example, they would say things like "First replace the ControlBoard, then replace the VFOBoard, and then do the rest of the actions taken for Fatal Error in the Microcode." Implicit in this statement is the fact that any actions already taken should not be repeated. *DoRest* nodes in DARN directly model this expression of the technicians. This of course requires that the DARN interpreter keep a history of actions that have already been done.

A special class of actions were created to express concisely another combination of activities. For some parts of the system, one could guarantee that if a symptom were found, then the associated repair would fix that symptom. For example, if the voltage level in the power supply was outside some specified tolerance, then adjusting it would ensure that it was then within tolerance. Similarly, replacing a broken cooling fan by an obviously working one would not require checking the cooling fan again. These actions are generically called *VerifiedAction*.

### Fault Identification

How does DARN recognize when it has identified a problem? When it runs a test a second time after taking some action A, and the symptom has changed, there are two possibilities. First, if the symptom changes from an error indication to OK, then clearly the action A fixed the problem. If the action was the replacement of a part, then the replaced part was faulty and can be labeled as such. The action could also have been an adjustment, and the system can record that there was a need for such an adjustment on this particular machine.

The situation is more complicated if the symptom changes from one error indication,  $E_1$ , to another,  $E_2$ , when replacing a part  $P_o$ . It could be that the original part,  $P_o$ , had no fault and that the replacement part  $P_{r1}$  has a fault. We have been told that this is not completely unlikely. To test this hypothesis, a second replacement part  $P_{r2}$  is inserted in the machine. If the error indication returns to  $E_1$  then the original part,  $P_o$ , is determined to be fault-free, and  $P_{r1}$  is marked as being faulty. If the error indication stays at  $E_2$ , then  $P_o$  is marked as being faulty,  $P_{r1}$  as being all right, and there is a second fault in the machine.

In some circumstances the technicians know that a symptom cannot be caused by a newly replaced board, and there is no need for the above *majority verification procedure*. There are two different cases. The rare case is when a second hardware fault is recognized, but the symptom precludes it from being the replaced part. In this case, the replaced part  $P_o$  is marked as faulty, and the repair process is continued. Much more frequently, the second error seen is a *software fault*, which may have been caused by the hardware fault (now fixed)—for example, data have been garbled on the disk by the formerly malfunctioning hardware. In this case,  $P_o$  is marked as faulty, and a second procedure is used to try to restore the disk data. A similar diagnostic and repair network also represents the procedures to be followed in this case of servicing.

## IV. User Interfaces

There are two distinct sets of users for this system, and the interfaces for the two reflect the differences in their needs. One type of user is a consumer of the knowledge base—say a less-experienced technician trying to use this knowledge as a guide through a service session. The second is an expert technician trying to examine and augment the knowledge base, i.e., a producer of the knowledge. Both kinds of users need the capability to interact with the system for solving cases, but the producers also need to have an overview of the entire knowledge base

so that the knowledge base can be easily extended and modified. We have experimented with a variety of interfaces and in this section we describe some of these interfaces, paying particular attention to the different kinds of needs.

### A Browser for Repair Plans

A basic need for all users is the ability to browse around in the plan structure. The graph-like structure of the plans makes it easy to build graphical browsers in languages such as Interlisp-D (Sannella, 1983). DARN was implemented in Loops (Bobrow & Stefik, 1982; Stefik & Bobrow, 1985), an object-oriented extension of Interlisp-D. Figure 3 shows a fragment of the plan being browsed inside a display window on the screen of a Dandelion lisp machine. The plan browser explicitly shows the plan elements as distinguished nodes. For example, the *observation* nodes are shown enclosed inside braces ({}), *test* nodes are enclosed inside square brackets ([]), and *actions* are enclosed inside angle brackets (<>). Some of the newer extensions to the browsers in Interlisp/Loops allow even more iconic display of the nodes.

The plans are usually too big to fit inside a window. Some of the basic scrolling mechanisms provided by Interlisp allow a user to traverse the graph structure easily. Sometimes, a user might want to restrict their attention to some subplan. This can be accomplished by creating a subbrowser rooted at any of the nodes in the original browser. We have also experimented with path browsers. These browsers grow as the user traverses them, keeping the branching factor low.

### Plan Creation Interface

The *producers* (including the knowledge engineers) of the knowledge base need a specialized interface that lets them easily create and modify the plan structures. Our working hypothesis was that it is easier for users to create the plans by directly manipulating the graph structure of the plan, rather than by describing the plan in some textual form. The structural transparency of the plan structure was an important consideration in this regard. In other words, given that the graphical representation of the plan adequately mirrored its structure, it should be possible for producers to enter new knowledge or modify existing one by directly modifying the graphical representation. The plan browser described earlier was extended to allow plan creation and modification actions at any of the nodes in the plan. Each node in the browser was made "active," i.e., it could be selected by a pointing device such as the

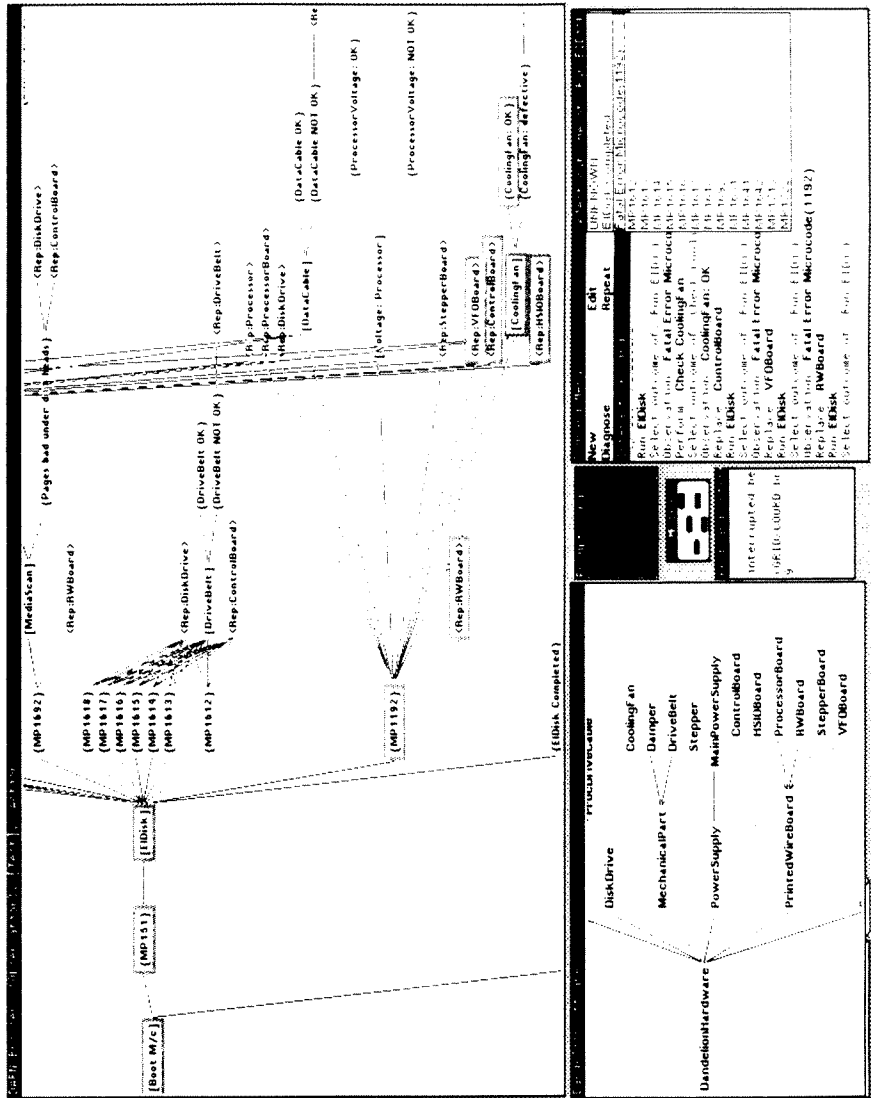


Figure 3. Typical Screen Layout During Interaction with DARN. The top half of the screen shows the plan browser. A node in the browser is either a *Test*, an *Observation*, or an *Action*. The browser window can be scrolled or reshaped. Selection of a node with left-button presents a menu of choices which allows addition of new information or modification of existing information. The menu on the bottom right of the screen shows an example of the menu typically presented to a user while the system is being used for servicing a broken machine. The other window on the bottom right shows a brief history of the current interaction with the system.

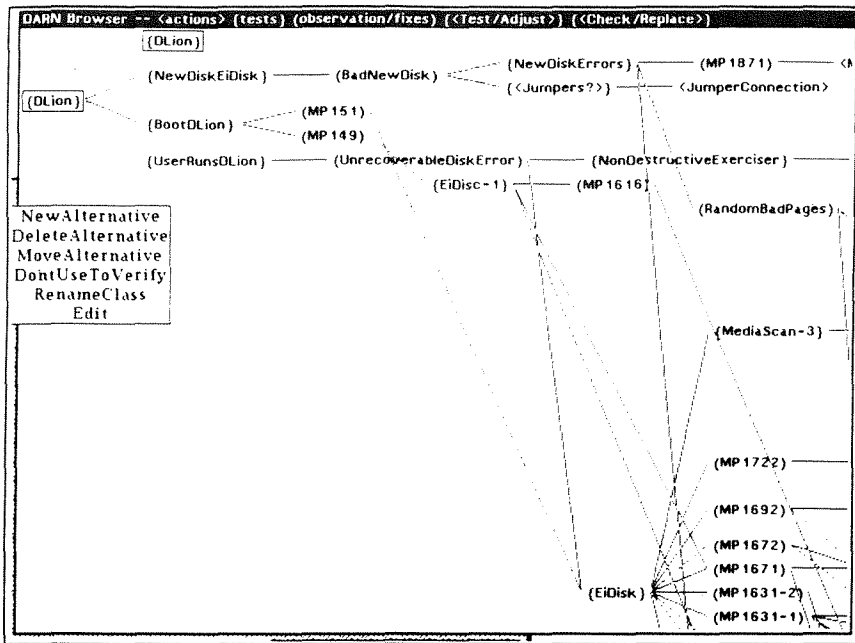


Figure 4. Plan Creation Interface The plan browser is shown with a menu that comes up when a node is selected for making plan modifications. Complete plans can be created by interactively creating nodes and extending them.

mouse to bring up a menu of choices. Selection with the left button brought up a menu of plan-editing choices and selection with the middle button brought up a menu of plan execution choices (to be discussed later). Figure 4 shows a plan browser with the menu of choices for modifying the plan. Some of the basic options are:

NewAlternative  
DeleteAlternative  
MoveAlternative  
Rename  
Edit

Notice that these commands are generic commands which are interpreted differently by the different kinds of plan elements. Generically, *NewAlternative* allows a plan to be extended from any of its elements. However, the permissible extensions are dependent on the selected node. For example, selecting the *NewAlternative* command for an *observation* displays a browser menu showing all the generic tests and actions known to the system. Once the user selects a class from this

menu, the system displays a second menu listing all the known instances of that class already in the knowledge base (in case the user wanted a previously created node) and an option to create a new one. For a new node, the same process is recursively followed to allow the user to fill in the description of this new node. A complete subplan can be created this way, with the system prompting the user for appropriate choices along the way. This is made possible by representing with each kind of node a complete description of the rules of interconnection, including the specializations for domain-dependent elaborations of the basic representation.

Our initial experience with DARN has been that our domain experts (knowledge producers) found the graphical representation to be a natural way to interact with the system with minimal training. We were very impressed with how quickly they could extend the knowledge base. We discuss our experiences in the last section.

### Execution Interfaces

Consumers (trainee technicians or others trying to use the system as a consultant) of the knowledge base primarily need two kinds of interfaces. The first is a plan execution browser that graphically shows where the user is in the plan. The second is a history interface that contains a summary of the interaction with the system and can be used as a transcript of the consultation. We briefly describe both kinds of interfaces.

#### Plan Execution Interface

DARN can guide a user through a diagnosis and repair process by executing the plans in the knowledge base. The test and action elements are used to prompt the user to perform the relevant action. The observations are typically used to prompt the user so they could inform the system of the results of performing some test. The following is a simplified consultation session between a rookie and an expert (or the rookie and DARN if not taken literally). [Bold face indicates remarks by the rookie, and italics indicate advice by an expert technician]:

**The machine was booted. Stopped with maintenance panel code (MP) showing 151.**

*Run EIDisk diagnostic program.*

**The machine stopped with MP 1192 (Fatal Error Microcode).**

*Replace HSIO board.*

*Rerun EIDisk.*

**The machine stopped with MP 1192.**

*Check cooling fan.*

**Cooling fan is OK.**

*Replace control board and rerun EIDisk.*

**The machine stopped with MP1192**

*Check voltage at processor.*

**Processor voltage is OK.**

*Check voltage at disk drive.*

**Drive voltage is Not OK.**

*Replace the cable connecting the processor and the disk drive.*

*Rerun EIDisk.*

**EIDisk ran successfully.**

What you just saw was fine as a transcript of the consultation, but is clearly not suitable as the primary interface to help the user *during* the consultation. It is insufficient because it does not let the user see the plan as it is being executed. It does not show the choices at each stage. It does not visually focus the user on what has happened and what else is possible. It also does not allow a user to ask questions about the purpose of the actions taken. We have extended the basic plan browser to provide a more active execution interface. Essentially, the plan browser can be made active at any test node by selecting the service command at that node. This starts a service consultation session, primarily driven by the system but allowing some override by the user. Figure 5 shows a snapshot of the screen in the middle of a consultation. Notice that in the browser, the nodes that have been traversed at any given time are highlighted, giving a visual summary of where the user is in the plan. The system uses the alternatives in the plan to prompt the user. Thus the user can be prompted to select the correct observation after running a test or taking an action. A user can also see what other actions are possible when they are asked to carry out a particular action from a *FixList*. A user can override the order of fixes by selecting an action that makes more sense in a given situation. As the plan execution proceeds, the browser automatically scrolls, always positioning the relevant part of the plan in the display window.

**Case History Interface**

A complete history of the service consultation is kept in the browser as discussed above. It is also kept in a textual form in a separate window

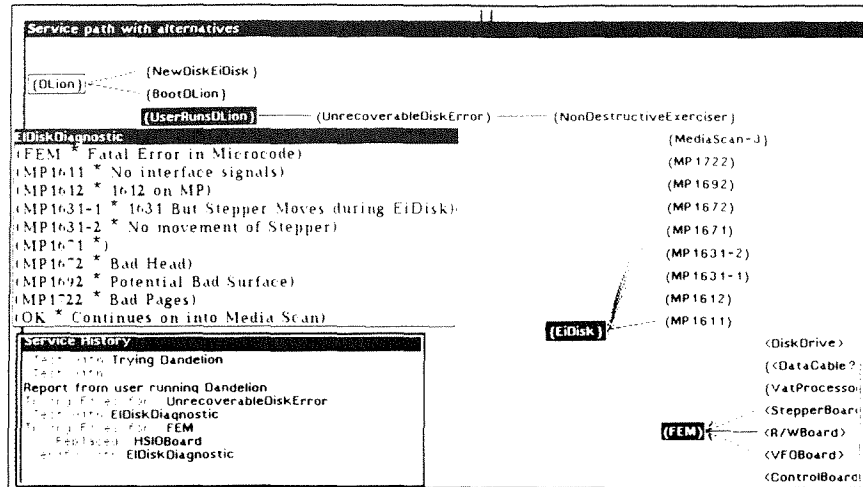


Figure 5. Plan Execution Interface The plan browser is shown with the path followed in the current consultation highlighted. The menu in the left middle of the window prompts the user to select the observation from running the *EiDisk* test. The user's selection, *Fatal Error Microcode (FEM)*, is also highlighted. The window in the bottom left provides a continually updated history of the dialogue.

(shown in the bottom left of Figure 5). This allows the user to scan the chronological sequence of events, as opposed to the logical sequence presented by the plan browser.

## Discussion

### Experience with DARN

The initial prototype developed by us contained only a fraction of the knowledge needed to repair the fixed disk. The first experiment we conducted was to try to get our expert collaborators to extend the knowledge base. We were surprised by how quickly they were able to learn enough both to extend the knowledge base as well as use the system as a consultant on test problems. In a matter of few days we were able to double the coverage of the system. Our experts were also very excited about using the system as a medium for making their experience available to new technicians. However, for a variety of reasons the project was terminated. One key reason was that the particular fixed disk model that was our initial domain was proving to be too

much of a maintenance headache and was phased out in the product line. The replacement disk had fewer replaceable parts and the new repair strategy was simply to replace the complete drive.

### **Version of DARN for Copier Repair**

A second major experiment that we conducted was to try to represent the knowledge for diagnosing and repairing a Xerox copier. Two important classes of copier faults were selected for the experiment: copy quality and paper tray elevator faults. Our collaborator on this project, who is an expert at repairing printers, tried to represent the knowledge in the Fault Isolation Procedures (FIPs). He found that, while he could represent the knowledge using the plan language, there were some significant differences between the disk and printer domains that called for extensions to the DARN framework. For example, copier diagnostics are far more complicated than those of disks. Failures need to be isolated at different levels before a suitable repair can be made; or there is often a long series of steps that have to be taken to create the proper context for performing a diagnostic test or corrective action.

Some of these observations suggest a need for a richer representation that allows setup procedures to be described, including guiding a technician who might make a mistake during these procedures. Another extension that was clearly needed was some abstraction capability that would allow a plan element to be expanded into a more detailed sub-plan when needed. The size and complexity of the copier repair plans also severely taxed the browsing features in DARN. It was clear that more powerful browsers are needed that could for example suppress selective detail, provide a top-down view, or maybe act like fish-eye lens, where the details are blurred around the edges, allowing more information to be displayed in a window.

### **Other Shortcomings**

There are both omissions in current capabilities of the system, and problems with the system by virtue of its structure. As an example of the former, we cannot handle in any easy way intermittent errors, or some types of dependent faults. The latter form of errors comes about because the repair plans contain no model of the function or structure of the system, and no real reasoning capability. Another problem is that the relatively simple interpreter in DARN forces an order in which data are gathered, precluding situations where a technician has already run some tests and fixes and needs assistance.

### Implicit Information

In its current form, the plans embody different kinds of implicit information. For example, suppose we have a nested test which checks for the voltage at a disk after finding that the voltage at the processor is all right. If the voltage at the disk is "all right," then the conclusion is drawn that the cable between the two should be replaced. DARN does not have any explicit information about that connection, and this structural information is implicit in the network. Furthermore, if some other node was connected to the node which tested the voltage at the disk in another context, but it did not follow the measurement of the voltage at the processor, then the implicit context of the measurement would be violated, and the conclusion drawn would be invalid.

### Annotations

Currently, we provide the users with the capability of annotating plan nodes with information to be used for construction and explanation purposes. Annotation of a node can indicate that it must appear as a subTest of a particular other Test. This kind of structural information can also be included in the descriptions attached to the nodes so that the system can itself check for violations and prevent them. Annotation can also indicate reasons for the ordering of actions. Information compiled into the order includes things like probability of fault causing a particular symptom, or the cost of trying a particular action (e.g., it takes 15 minutes to change this board, and only 5 for most other boards).

### Acknowledgements

We are very grateful to our domain experts, Milt Mallory, Ron Brown, and Ted Manley, for their time, patience, and energy. Julian Orr was primarily responsible for trying to adapt the DARN framework for the copier repair problem. We are also pleased to acknowledge the support and helpful criticisms of Mark Stefik. Clive Dym and some anonymous referees provided valuable comments on earlier drafts of this paper.

### References

- Bobrow, D. G., & Stefik, M. J. (1983). *Loops Manual*. Xerox PARC, December.  
 Brown, J. S., Burton, R., & de Kleer, J. (1982). Pedagogical, natural language and knowledge engineering techniques in SOPHIE I, II and III. In D. Sleeman

- & J. S. Brown (Eds.), *Intelligent tutoring systems*. New York: Academic Press.
- Chandrasekaran, B. (1984). Expert systems: Matching techniques to tasks. In W. Reitman (Ed.), *AI applications for business*. Norwood, NJ: Ablex, pp. 116-132.
- Chandrasekaran, B., & Mittal, S., (1983). Conceptual representation of medical knowledge for diagnosis by computer: MDX and related systems. In M. C. Yovits (Ed.), *Advances in computers*, Vol. 22.
- Chandrasekaran, B., & Mittal, S. (1982). Deep versus compiled knowledge approaches to diagnostic problem-solving. *Proceedings AAAI-82*, Pittsburgh, August.
- Clancey, W. J. (1985). Heuristic Classification. *Artificial Intelligence*, 27(3).
- de Kleer, J., et al. (1979). Explicit control of reasoning. In P. H. Winston & R. H. Brown (Eds.), *Artificial intelligence: An MIT perspective*. Cambridge, MA: MIT Press.
- de Kleer, J., (1984). How circuits work. *Artificial Intelligence*, 24(1-3).
- Genesereth, M. (1984). The use of design descriptions in automated diagnosis. *Artificial Intelligence*, 24(1-3).
- Mittal, S. (1980). *Design of a distributed medical diagnosis and database system*. Ph.D. dissertation, Department of Computer and Information Science. Ohio State University, Columbus.
- Pan, J., & Tenenbaum, J. M. (1986). PIES: An engineer's "Do it yourself" knowledge system for interpretation of parametric test data. *Proceedings AAAI-86*, Philadelphia, August.
- Sanella, M. (Ed.) (1983). *Interlisp reference manual*. Xerox Corpo. October.
- Stefik, M. J. (1986). The next knowledge medium. *AI Magazine*, Spring.
- Stefik, M. J., & Bobrow, D. G. (1986). Object-oriented programming: Themes and Variations. *AI Magazine*, Winter.
- Sussman, G. J., & Stallman, R. (1975). Heuristic techniques in computer-aided circuit analysis. *IEEE Transactions Circuits & Systems*, CAS-22.
- Szolovits, P., & Pauker, S. G. (1978). Categorical and probabilistic reasoning in medical diagnosis. *Artificial Intelligence*, 11, 115-144.