STARKEEPER® NETWORK TROUBLESHOOTER: AN EXPERT SYSTEM PRODUCT

Todd E. Marques

Todd E. Marques *is a member of technical staff in the Network Management and Host Interfaces Department at AT&T Bell Laboratories in Liberty Corner, New Jersey, where he works on design and implementation of the StarKeeper network troubleshooter. He received a B.S. in psychology from the University of California, Davis, and M.A. and Ph.D. degrees in psychology from Rice University. He joined AT&T in 1980.*

StarKeeper network troubleshooter is a real-time interactive expert system that assists clerks, technicians, and network administrators with varying skill and experience levels in isolating and correcting faults in Datakit® virtual-circuit switch networks. The system interrogates and manipulates network components via a standard login to an existing element management system. Given a symptom description and minimal background information, the system can automatically isolate and correct network faults. It provides detailed corrective procedures in cases where human intervention is required. Feedback mechanisms let the system adapt to different networks and to individual operators.

Background

The Datakit virtual-circuit switch (VCS)[1] provides multiprotocol communication between host computers and terminals of various types. Its modular or kit-like architecture yields a variety of configurations to suit different communications needs. Datakit VCS nodes can function independently as hubs for local-area networks, or they can be interconnected to form massive wide-area networks. Datakit VCS networks are widely deployed within AT&T and in government, business, and universities.

In late 1987, StarKeeper network troubleshooter became generally available for commercial use in Datakit VCS networks. The troubleshooter is an expert system designed to assist network technicians and administrators in isolating and correcting network faults.

An expert system is a program or collection of programs that performs complex problem-solving tasks. The logic used by expert systems in attacking problems is frequently patterned after the problem-solving strategies employed by acknowledged experts in the subject matter. Applications where expert systems are currently in use include chemistry, electronics, geology, law, and medicine. Typical *problem domains* served by expert systems include diagnosis, prediction, planning, interpretation, design, instruction, monitoring and repair.[2]

Panel 1. Terms in This Paper

EMS	element management system
DTR	data terminal ready
EIA	Electronic Industries Association
FIFO	first in, first out
lcmd	local command mode
LHS	left-hand side
Lisp	list programming language
lproc	local processing mode
NMS	network management system
rcmd	remote command mode
RHS	right-hand side
rproc	remote processing mode
SQL	standard query language
VCS	virtual-circuit switch

The troubleshooter is one of the first expert systems available for commercial use in data communications that interacts with network elements in real time and, when appropriate, intervenes to correct faults. It is also distinguished from many other expert systems on the basis of its ability to "learn," or change its behavior to accommodate different operating environments and to suit individual differences in the way people describe network problems.

The goal in introducing the troubleshooter was to reduce significantly the complexity of isolating and correcting faults in Datakit VCS networks. Our approach to task simplification involved mechanizing the diagnostic procedures and key judgment processes associated with network troubleshooting. The troubleshooter is able to assume responsibility for hypothesis generation, planning, testing, analysis, and other cognitive aspects of fault isolation. This enables people with minimal data communications experience to function as expert diagnosticians.

The troubleshooter uses an existing element management system (EMS) called StarKeeper network management system (NMS) to gather real-time status and configuration information from the network, and to issue operations commands to all points within the network. Figure 1 shows the relationship between the troubleshooter and the EMS. Before the troubleshooter became available, network administrators and technicians interacted directly with the EMS. Now, as shown in Figure 1, operators can interact with the troubleshooter, which in turn converses with the EMS. Technicians are shielded from the EMS, the underlying network, and most knowledge of fault-isolation procedures. With the troubleshooter, therefore, technicians can choose to limit their involvement in the fault-isolation process to describing symptoms and responding to system directives.

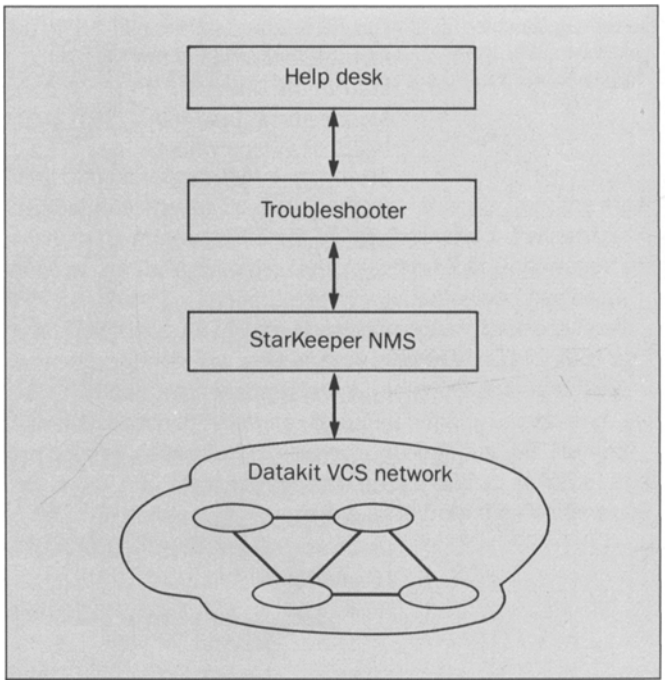


Figure 1. The troubleshooter operating environment.

This paper describes the troubleshooter's architecture and primary modes of operation. It also addresses principal issues and concerns that arose during the troubleshooter's progression from prototype to product.

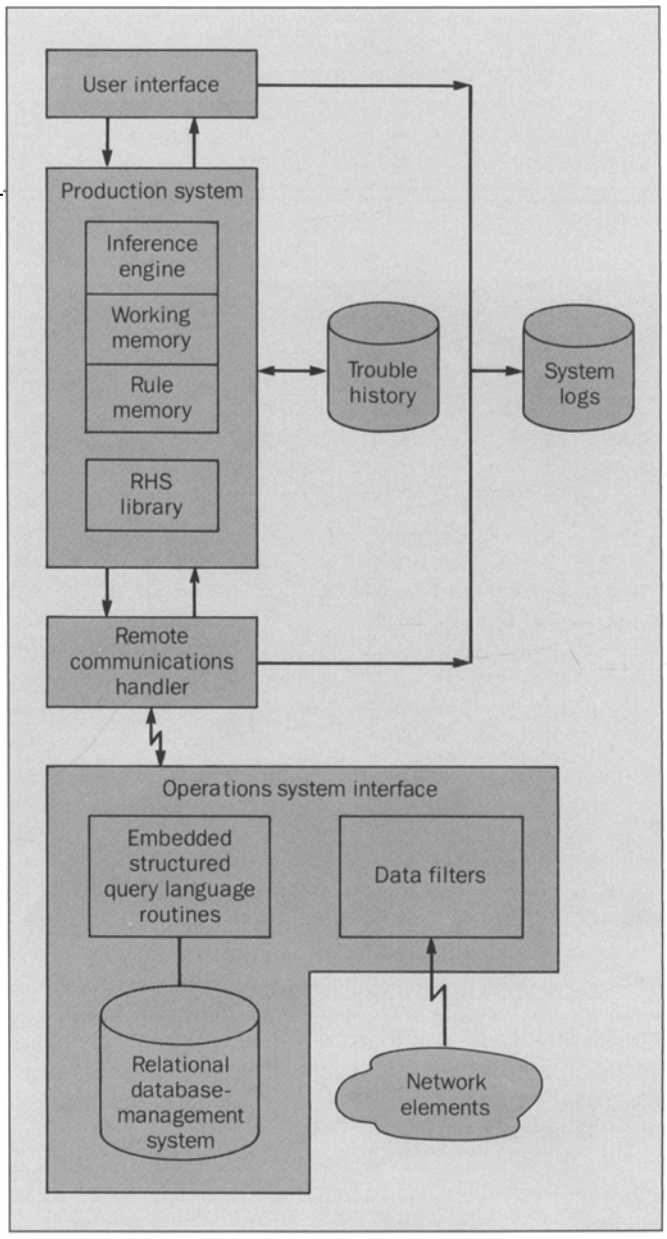


Figure 2. The troubleshooter architecture.

System Architecture

A high-level view of the architecture appears in Figure 2. As shown, the system is a collection of several programs that work cooperatively to isolate and correct faults. The troubleshooter runs under the UNIX® operating system on the AT&T 3B2 computer line.

Production System. At the core of the troubleshooter architecture is a production system. The production system is a member of a general class of systems referred to as *pattern-directed inference systems.*[3]

The production system consists of rule memory, which is an unordered set of condition-action pairs called *production rules*, a database of assertions or elements called *working memory*, and a global control mechanism referred to as an *inference engine*. It is implemented in OPS4,[4] a Lisp-based production-system language developed at Carnegie-Mellon University.

Within the troubleshooter, the condition element, or left-hand side (LHS) of a rule, generally refers to the state of a network element—especially to an abnormal state. The action element, or right-hand side (RHS), typically refers to an appropriate corrective response.

Consider the simple rule:

```
(
if

   (device type: TERMINAL)
   (eia lead status: DEV_DTR DOWN)

then

   (check continuity of RS343 cabling)
   (determine whether symptom has
    disappeared)
)
```

The rule states that, if the device under test is a terminal and investigation of the Electronic Industries Association (EIA) RS-232 leads indicates that the device DTR (data terminal ready) lead is not asserted, then the operator should verify that the RS-232 cabling is securely attached to the end-user terminal, and finally, determine whether the initial symptom has disappeared.

The rule base is the central repository of the troubleshooter's *domain knowledge*, or the facts and proce-

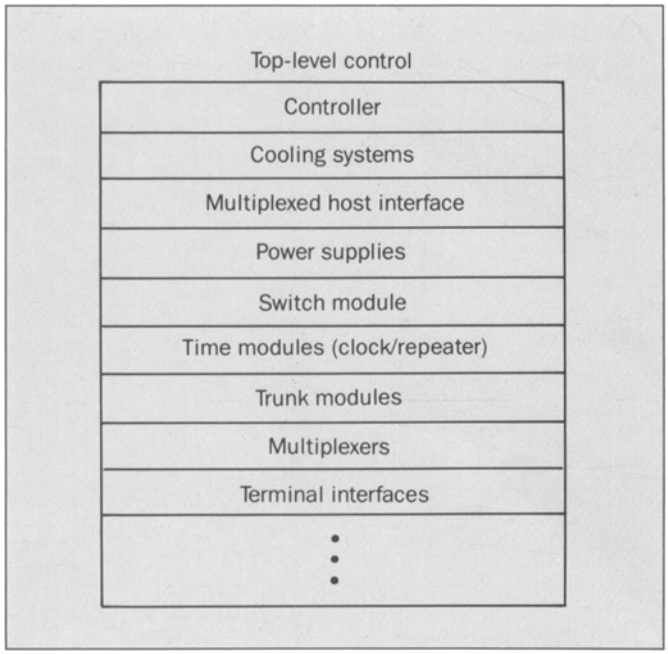


Figure 3. Rule base organization.

dural knowledge related to isolating and correcting faults in Datakit VCS networks. The overall organization of the rule base is modeled after the physical design of a Datakit VCS node, which is shown in Figure 3.

A Datakit VCS node consists of one or two cabinets, each containing one to four slotted shelves that house plug-in circuit cards. The cards perform specialized functions such as clocking, switching, and interfacing with terminals, hosts, and other nodes. Each cabinet is also equipped with power and cooling subsystems.

The node cabinetry comes in several versions for different operating environments (e.g., central office and customer premises) and cost configurations.

The rule base consists of approximately 1000 rules which are grouped into collections called *specialist modules*. Most specialist modules are responsible for diagnosing individual pieces of equipment. However, some specialist modules perform other highly involved tasks such as network circuit tracing, which is used in planning analyses. As shown in Figure 3, the troubleshooter currently supports Datakit VCS common equipment such as cabinetry, packet switch, clock, and controller, as well as multiplexers, internodal trunks, and many interface boards. A new release that is being readied for field testing will support additional synchronous and asynchronous Datakit VCS interface boards, components, and bridges to public networks, as well as the forthcoming Datakit II VCS product line. The upgraded rule base consists of approximately 1400 rules.

A process known as *knowledge engineering* was used to compile the knowledge base. As detailed in Reference 5, the process involved extensive interviews with Datakit VCS field-support personnel and the hardware and software developers responsible for each of the supported network elements. The information obtained from the interviews was combined with information gleaned from other sources such as internal design documents and equipment operations manuals. The knowledge engineering process resulted in a collection of rules that embodied the knowledge of several subject-matter experts.

The inference engine determines if and when rules contained in the rule base are applied during the course of an investigation. Basically, it searches the contents of working memory for assertions or elements that match condition elements associated with rules contained in rule memory. After a match has been found for every condition element associated with a rule, the rule is said to be *instantiated* and it is eligible to be *fired*. When a rule is fired, its RHS or action elements are executed. Unlike conventional programming languages where control is governed by the sequential execution of programming statements, the production system is *data-driven*, that is, driven by the contents of working memory.

In the example above, the presence of the condition elements:

```
(device type: TERMINAL)
(eia lead status: DEV_DTR DOWN)
```

in working memory could result in the execution of the corresponding action elements. The firing of a given rule is often uncertain because the match-act cycle used by the inference engine is actually more than a straightforward match operation. For most production systems, there will be times when several rules are eligible to fire simultaneously. The eligible rules comprise what is referred to as the *conflict set.* Much of the complexity in the match-act cycle relates to reducing the conflict set to a single rule, a process known as *conflict resolution.*

The criteria applied during conflict resolution vary somewhat according to the production system implementation, and may in some cases be customizable or completely operator-defined. The OPS4 criteria are:

1. How recently elements have been added to working memory (most recent dominates)
2. The complexity of the LHS in terms of the number of condition elements present (greater complexity dominates)
3. The specificity—the number of constants in relation to the number of variables—in the LHS (greater specificity dominates)
4. How recently the rule has been compiled (most recent dominates).

The rule-based programming methods employed in production systems are particularly well-suited for network fault-isolation tasks. The number of logical and physical dependencies among network elements in combination with the large number of problem manifestations (alarms, outages, other symptoms) makes coherent, algorithmic description difficult.

The condition-action pairing that is the essence of rule-based programming provides a natural and maintainable mode of expressing complex dependencies and contingencies. This mode is natural in the sense that it conforms to the way people often explain their reasoning: "If this were the case, then I would do the following." Most narrative associated with complex reasoning often is so fraught with ifs, ands, and buts that its representation in procedural form could be extremely convoluted.

Production systems are especially useful in building systems that will operate in rapidly evolving domains, such as data communications, where new products and services are frequently introduced. Knowledge can be added to (or deleted from) a rule base with relatively little impact on the operation of the system as whole. This is due largely to the fact that, while rule bases usually contain some control knowledge, most aspects of control are achieved implicitly through the operations of the inference engine.

Not all tasks performed by the troubleshooter are well-suited to production system implementations. Many tasks peripheral to the troubleshooter's central problem-solving role are better suited to conventional procedural implementations. These tasks include statistical computation, sorting, database access, and input/output processing. In general, stable, well-defined tasks requiring high-speed execution should be performed using conventional procedural languages. It is difficult for production-system languages to equal the speed of procedural languages because the match-act cycle is used to control execution.

Fortunately, it is possible to intersperse rule-based and procedural code in many production-system languages. The troubleshooter's production system is augmented by a library of procedural functions written in Franz Lisp™ programming language,[6] a relatively high-level procedural language developed at the University of California, Berkeley. (Franz Lisp is a trademark of Franz, Inc.) These functions are RHS functions, invoked from the right-hand side, or action side, of a rule. The RHS functions perform the various computational and data manipulation tasks best managed by procedural code.

Support Components. Surrounding the production system and run-time library are support components that allow the troubleshooter to interface with its operators and principal data sources and provide basic utilities. All sup-

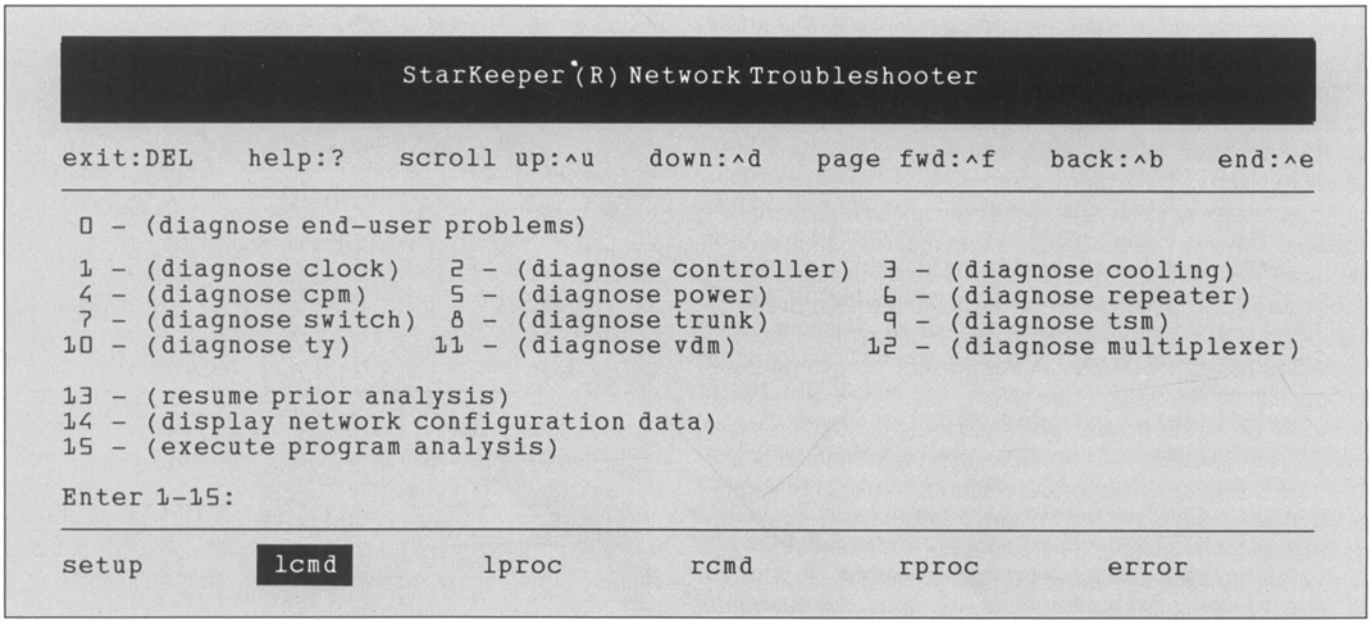


Figure 4. The troubleshooter operator interface.

port components are written in the C programming language.[7]

The operator interface is shown in Figure 4. It consists of a screen divided into four sections. Below the system banner and control key definitions is a one-line text area that displays error and informational messages. Also displayed is the exact text of operations commands, database queries, and other commands that are transmitted by the troubleshooter to the EMS. The main interactive text area follows the message area. All menus, prompts, displays, and other information are provided here. The text window may be paged or scrolled in forward and reverse directions. The bottom border of the operator interface is a status annunciator system that provides information about the operational state of the system. The indicators are highlighted when the system is:

- Establishing communications with the EMS (setup)
- In local command mode (lcmd)—that is, expecting operator input from the terminal
- In local processing mode (lproc)
- In remote command mode (rcmd)—that is, transmitting a command to the EMS
- In remote processing mode on the EMS (rproc)
- Handling an error condition of some sort (error).

The troubleshooter and the EMS host computers communicate with each other by way of the Datakit VCS network. Communications software is distributed across the two machines to establish and maintain error-free data transport. The software is also sensitive to losses in host-to-host connectivity. It will automatically reestablish connectivity when lost and resume dialogue at the point where it was disrupted. Robust machine-to-machine communications are usually important, but especially so when data-driven systems are involved. Since production systems

```
(class_name attribute_1 value_1 . . . attribute_n value_n)
```

(a)

```
87-12-31 18:12:18
 M verify amux board 2 5

   MODULE ADDR: 2  CABINET NO: 0  MODULE TYPE: amux (16-board)
   TRUNK TYPE: hs
   TOTAL BOARDS PROVISIONED: 7  CHANNELS: 232 SVC  STATE: out
   VERSION: standard  SERVER: controller

     BOARD       SERVICE
     ADDR        STATE
     5           rfs
```

(b)

```
(vamux maddr 2 cab 0 mod_type amux)
    (vamux trk_type hs)
    (vamux provision 7 channels 232 msvc_state out)
    (vamux version standard server controller)
    (vamux baddr 5 bsvc_state rfs)
```

(c)

Figure 5. Representative reports. (a) Format of a transformed report. (b) A report before transformation. (c) The same report after transformation.

often lack precise *a priori* knowledge about the data to be processed, it can be difficult to identify data corruption. Therefore it is essential that data integrity be guaranteed by upstream processes.

Upstream processing is also required to transform human-readable reports and displays produced by the EMS into a form that can be interpreted by the production system. The data filters shown in Figure 2 serve that purpose. They transform reports into a series of parenthesized expressions of the form shown in Figure 5a. A sample preprocessed Datakit VCS report as an administrator would see it appears in Figure 5b. The same report after filtering is in Figure 5c. Note that headings and fields regarded as extraneous are filtered out of the original report.

Not all information required by the troubleshooter to investigate a problem can be obtained from the real-time status displays generated by Datakit VCS. Much of the information pertains to network configuration. Examples of required configuration data include mappings between logical and physical network addresses, the contents of specified slots within a given Datakit VCS shelf, and the locations and interconnections of all internodal trunks within the network. The system uses this information to plan its investigations and to navigate throughout the network.

Configuration information is stored and maintained on the EMS under an Informix® relational database-management system (registered trademark of Informix Software, Inc.). The database is shared by the troubleshooter and EMS applications. C routines with embedded structured query language (SQL) statements are used to

extract necessary tables and records from the database.[8] The configuration data are transmitted over the communications link to the troubleshooter host and loaded directly into working memory.

The system's principal utilities provide logging and archiving capabilities. All dialogue between the operator and the troubleshooter and all communications between the troubleshooter and the EMS are logged. The operator of the system is also identified in the logs, and it is therefore possible to determine precisely what the system did and who was in control of the system at any given time. The logs are useful as audit trails for security purposes, and they are instructive for novice operators, providing them with numerous examples of how to go about sectionalizing network faults.

System Operation

There are many reasons for using an expert system to isolate faults. Some people do not know much about sectionalizing faults and, as suggested above, can learn much by observing the system in action. Others are not knowledgeable about data communications and have no desire to become experts. They often wish the network would take care of its own problems. And, of course, there are experts in the field who simply need "power tools" for performing tasks that may not be perceived as intellectually demanding but are tedious or error-prone.

The troubleshooter has been designed to be useful in a variety of situations, and useful to people with widely varying skill and experience levels. Operators can draw on a subset or all of its capabilities according to their skill levels and task demands.

This section discusses the basic modes of system operation in the context of three different kinds of tasks:

- Diagnosis of end-user problems
- Analysis of alarms and other network-generated messages
- Preventive maintenance activities.

End-User Problems. Problems reported by an end user are potentially the most challenging of all network problems a technician may encounter. When an end user calls a help desk or operations center to report slow response time, or the presence of stray characters on the screen, or other often vague symptoms, the technician faces several problems. The first problem relates to identifying the suspicious components. Initially, the technician may know only that the problem is "out there" somewhere. Even small networks may consist of hundreds of network elements. The question is: which elements could be involved and where are they? In the interest of maximizing network availability and technician productivity, it is important to solve problems as quickly as possible. Clearly, not all suspicious components are equally likely to be at fault in a given situation. Knowledge of the relative likelihoods can be exploited in formulating a plan of attack. The second problem, therefore, pertains to determining which suspicious elements are most likely at fault. The third problem relates to selecting test strategies appropriate for determining whether a suspicious network element is indeed faulty.

The troubleshooter can handle all the above problems, which are essentially subtasks associated with the overall fault-isolation process. Whether the system actually performs each of the subtasks depends primarily on the skill and experience levels of the technicians using the network troubleshooter. The discussion will focus initially on the case where a novice technician or clerk is confronted with an end-user symptom.

After invoking the system and selecting the option for diagnosing end-user problems, the system displays a menu of symptom descriptions that have been reported in the past. The operator is prompted to select the description that most closely matches the symptom just reported by the end user. If a suitable match between menu items and the current description is not found, the operator is free to enter a new symptom description. Once the symptom is encoded, the system prompts the operator for the end user's logical or physical address in the network. This address refers to the end user's network entry point. An example of a physical address is:

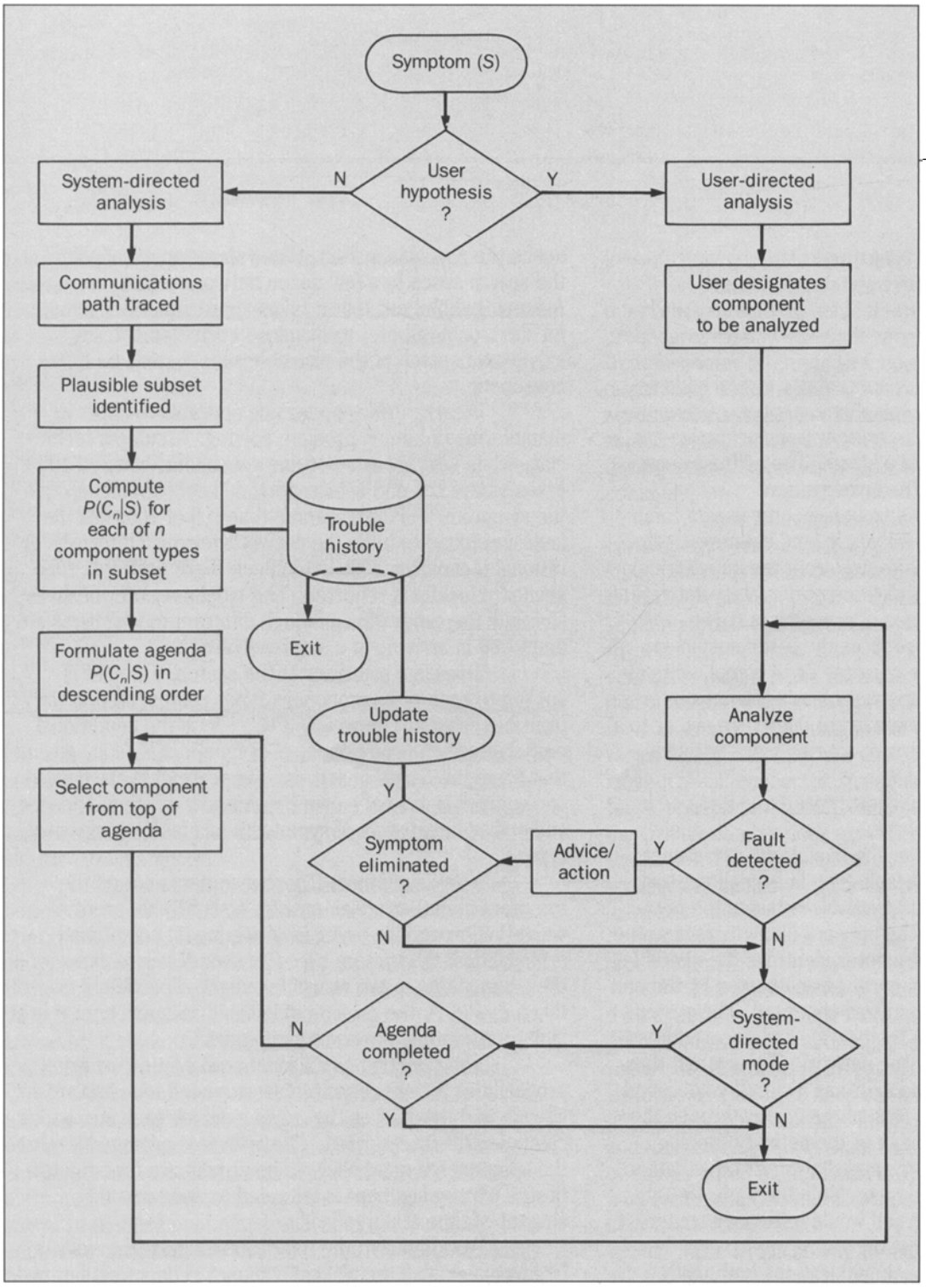


Figure 6. Methods for analyzing end-user problems.

```
node:  nj/garage/3n0
slot:  3
board: 4
port:  4
```

Operators have the option of specifying a logical address that may be the end user's telephone number, office number, or other mnemonic. The system transforms this name into the appropriate physical address. The address constitutes the starting point of the investigation.

From this point, the troubleshooter uses a combination of deterministic, probabilistic, and heuristic reasoning methods to work its way from the initial symptom description to the identification of the faulty network element. Through a sequence of procedures (to be described), it successively reduces the list of suspicious network elements from the universe of all components in the network to a single element such as a terminal interface module or an alternating-current power supply, or to a subcomponent such as a current-sharing block within a given power supply.

Figure 6 shows the basic procedures involved in dealing with end-user problems.

As suggested above, the troubleshooter cannot conduct a blind search for a fault even in a small network. However, it can operate on a *plausible subset* of the network—that is, a subset of the network likely to contain the faulty element. The troubleshooter identifies the plausible subset by tracing the communications path used by the end user at the time the symptom was observed. The system begins the trace given only the logical or physical address where the end user enters the network. The system then determines whether the end user has a circuit established and, if so, traces the exact path of the circuit through the network. It uses its knowledge of the network topology, Datakit VCS routing strategy, and real-time status data to assemble the path. If it finds a break in the path, the system can determine how the call would have been routed from that point to its destination.

By identifying the communications path, and hence the plausible subset, the system can usually limit the search space to a few dozen network elements. Assuming, however, that it takes approximately 1 minute for the troubleshooter to diagnose each element, an exhaustive search of the plausible subset could be time-consuming.

Clearly, prior experience operates to improve many forms of human problem solving. Assume a technician, while serving as a hot-line consultant, handled 10 cases where the end user reported "keyboard lockup" as the symptom. Further, assume that in 9 of 10 cases the fault was found to be in the network-terminal interface. A rational technician is likely to check there first next time keyboard lockup is reported. The troubleshooter behaves basically the same way, although different mechanisms are employed in arriving at a similar strategy.

To assign priorities in the search, a weight is applied to each of n component types represented in the plausible subset. The weight $P(C_n \mid S)$ is the conditional probability that a component C of type n is at fault, given that symptom S was observed. These weights are based on the troubleshooter's own experience, which is encoded and stored in a matrix of symptoms and faulty component types.

Novel symptoms (those symptoms not on the symptom menu) are dealt with by a slightly different weighting procedure. Instead of relying on conditional probabilities, the system uses the overall failure rates for the components in the plausible subset. The weight is taken as $P(C_n)$, the prior probability that component n is at fault, considering all known symptoms.

The resulting conditional probabilities, or prior probabilities for novel symptoms, are used to formulate an agenda that establishes the order in which each suspicious component is investigated. The agenda is arranged so that the components most likely to be at fault are investigated first. This arrangement is intended to minimize the amount of time taken to isolate faults.

Working from the top of the agenda, the system begins diagnosing the plausible subset. The specialist mod-

Table I. Examples of Initial Checks for a Terminal Interface Module

Check	Value
Hardware type	ty12
Configuration	Asynchronous terminal
Software release	3.3.1
Software service state	In service
Hardware service state	In service
Port service state	In service
Module reset count	0
FIFO reset count	0
FIFO overflow	0
Hardware errors	None
Sanity errors	0
Parity errors	0
Connection status	Idle

ule designed to diagnose the particular element under test is then invoked. In so doing, the troubleshooter looks for the same evidence of malfunction and uses many of the same strategies that would be used by a human expert in similar circumstances.

The troubleshooter typically begins by issuing a series of commands to the element to assess its current operational state. For a terminal interface card, the specialist module would prescribe checks contained in Table I. If the scan fails to reveal evidence of a fault, the element is passed, and the next element on the agenda is examined. However, if evidence suggests otherwise, additional information is collected in an attempt to pinpoint the fault.

Follow-on tests would be warranted if, for example, the connection status indicator for the end user's port on the terminal card above had registered *idle*. The reason is that, under certain circumstances, this state indicates disrupted connectivity between the end user and the network. In this example, the troubleshooter would execute a sequence of commands that interpret the RS-232 signals from the terminal. If the DTR lead is not asserted, then loss of connectivity is confirmed.

Once a fault has been isolated, the system either intervenes directly to solve the problem, or issues detailed procedures on how the fault should be corrected. The troubleshooter attempts to fix problems itself to minimize dispatching of technicians. It can clear many software faults automatically. In the example above, if the software service state had registered *out*, the system would restore the port to an operational state by issuing an operations command.

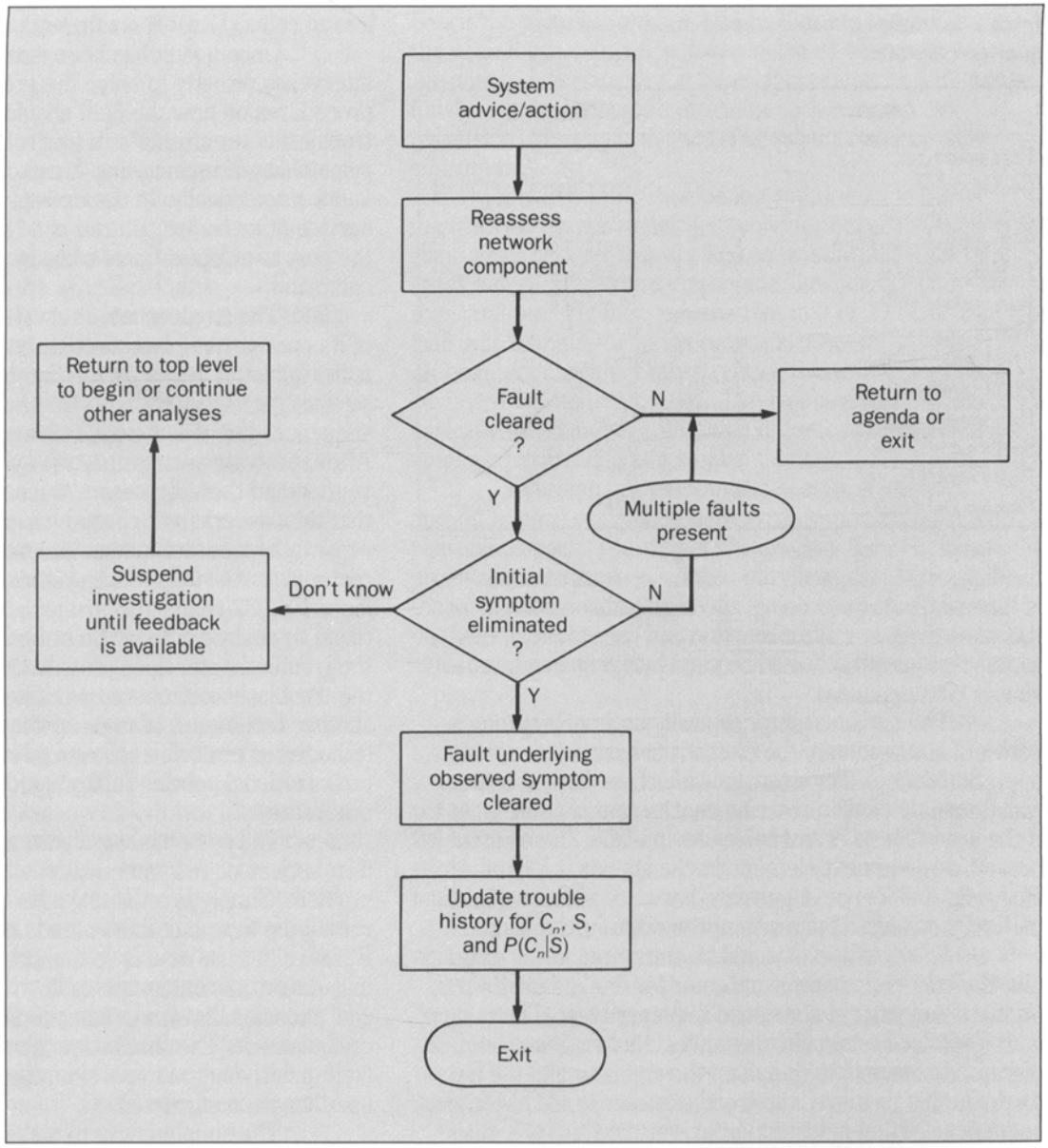
The troubleshooter always seeks the permission of its operator before actually intervening to correct a fault. In fact, it seeks permission before performing any *service-affecting* operation—that is, any operation (or test sequence) that will disrupt service to a network user. After receiving permission, the system performs the intervention and then reassesses the network element to verify that the desired state change has taken effect.

Sometimes human intervention is required to correct a fault. Obviously, the system cannot reconnect a loose RS-232 connector, manipulate hardware switch settings, or replace a defective power supply. In those cases, the troubleshooter does provide detailed procedures that the troubleshooter operator can perform or can pass to another technician. Who does what depends on the troubleshooter operator's experience level and the organizational policies regarding job assignments and work procedures.

The system always gathers feedback to determine if its actions or recommendations have eliminated the problem. Simply terminating after it has isolated and corrected the first fault in its path is not necessarily sufficient. Because its rule base is so comprehensive, the system may detect more than one fault within the plausible subset and, occasionally, more than one fault within a single piece of equipment. The feedback is aimed at ensuring that the fault underlying the initial symptom reported by the end user has been corrected.

The simplest way to make this determination is to ask the end user if the initial symptom has disappeared. As

Figure 7. Incorporating feedback.



shown in Figure 7, the feedback process typically works as follows: The troubleshooter isolates and corrects (or advises the operator to correct) the fault. Next, it instructs the operator to contact the end user who initially reported the symptom. If the symptom has disappeared, then the troubleshooter concludes that the fault just cleared was the cause of the symptom. It then terminates its analysis.

There are instances when immediate feedback is not available to the troubleshooter. For example, the end user may have stepped out of the office. Also, the symptom may be intermittent, in which case a lengthy observation period can be required before the troubleshooter can conclude with certainty that the symptom has disappeared.

Whenever immediate feedback is not available, the operator may suspend the current analysis—that is, put it "on hold." The operator can assign a mnemonic code (such as a trouble ticket number) to a particular analysis for later reference. After the operator assigns the code, the system saves the current analysis for resumption at a later time, and the technician is free to begin (or resume) work on other problems. Hence, operators can in effect work on many problems simultaneously and maximize their productivity. When the feedback becomes available, the operator enters the mnemonic code, and the prior analysis is resumed exactly at the point where it left off.

The ability to suspend and resume analyses also provides a mechanism for dealing with cases in which technicians need to be dispatched to clear faults. Depending on scheduling constraints, it may be several hours before manual procedures can be performed. With suspension capability, the operator can move on and address other problems while awaiting results from dispatched technicians.

After receiving confirmation that a fault is cleared, the system updates its trouble history to indicate both the symptom and the type of network component found to be at fault. This strengthens the statistical association between symptoms and fault locations, and constitutes the basic mechanism by which learning occurs.

Learning is demonstrated whenever a system improves its performance over time.[9] The troubleshooter uses its experience to determine which components are most likely at fault in a given situation and to arrange the agenda accordingly. The speed with which the troubleshooter isolates faults increases over time as it modifies the agenda to reflect the network trouble history.

The troubleshooter can maintain a separate trouble history for each authorized operator. This allows operators to establish their own symptom taxonomies. Thus, the system adapts to the pattern of failures within a network as well as individual differences in how these failures are described.

The method for sectionalizing end-user problems presented thus far presumes the troubleshooter operator is inexperienced in dealing with network faults. In fact, there is no need for an experienced operator because the troubleshooter can draw on its own experience and knowledge in handling all facets of the process. However, the troubleshooter does let expert technicians use their own domain knowledge.

An experienced technician is likely to identify the terminal interface card as the most likely location of a fault when an end user complains of keyboard lockup. The technician can direct the troubleshooter to investigate the network element believed to be faulty. In other words, technicians can formulate their own agendas when working on end-user problems with which they are familiar. In some cases, this can actually hasten the fault-isolation process because the system does not need to trace circuits, consult the trouble history, and perform other planning activities.

Much of the functionality of the planning operations described previously is unbundled so that it can be used as needed by expert technicians. For example, the expert technician can request that the troubleshooter simply trace the communications path used by an end user. Technicians can use the system to identify the plausible

subset, but assume total control for assigning priorities to the search within that subset.

Even in the operator-directed mode, the troubleshooter continues to solicit feedback on the effectiveness of its interventions and advice. This way, the troubleshooter can update its trouble history and use this information next time it is called upon to plan an analysis.

Analysis of Alarms and Messages. As almost any technician can attest, network components can produce a flood of alarms and messages, many of them useful and many of them not. It takes considerable experience to determine if and when a particular alarm or combination of alarms and messages is relevant.

Many error messages are self-explanatory and can be dealt with by rote. As shown below, a typical Datakit VCS alarm or message includes the physical address of the element, some descriptive information, and often a recommended action.


```
REPORT ALARM: Mode switch not enabled
CAB=1 SLOT=44
MODTYPE=ty12
Rec Act: Enable switch
```

Perhaps the recommended action could be embellished to include procedures on how to change switch settings. But expert-system technology cannot contribute much in working out these problems. There is little uncertainty about what is wrong and how it should be fixed.

There are cases where greater expertise is required. Some alarms and messages, while seemingly straightforward, are more obscure and may arise from one of many conditions. Consider the following message:

```
REPORT ALARM: Power supply failure CAB=1
Rec Act: Replace power supply
```

The power supply is extremely complicated and comes in several versions, each with different diagnostic procedures. In most cabinet configurations, there are three redundant power supplies. Detailed procedures are sometimes required to identify the specific power supply that has failed and to determine whether the fault can be cleared without replacing the equipment.

Because the alarm message contains the address of the power supply, the technician can invoke the operator-directed mode of operation discussed above, and direct the troubleshooter to investigate the power supply. In turn, the troubleshooter will analyze the power supply and lead the technician through the appropriate manual diagnostic procedures.

Preventive Maintenance. Thus far, discussion has focused on symptom-driven operations. The goal of most preventive maintenance activities is to prevent symptoms from arising in the first place. This entails routine checking of components to identify any evidence of impending problems.

The troubleshooter supports this type of preventive maintenance with a mode of operation referred to as *programmed analysis*. This mode lets technicians specify in advance arbitrarily large and diverse collections of network elements that are to be scanned for any indication of malfunction.

The collections are specified in ordinary text files that are stored on disk. They may contain the addresses of all trunk modules within a network, or all dial-in ports in the network modem pool. Any collection of elements that is meaningful to the technician may be included in the text file.

When operating in the programmed analysis mode, the technician directs the system to load a predefined text file. Once loaded, the system begins analyzing each element specified. The troubleshooter identifies all abnormal conditions associated with the element under test, and specifies corresponding corrective procedures. Using this mode, technicians can identify and correct numerous faults before they become symptomatic.

History and the Future

The troubleshooter started as an exploratory project in 1983. The approach was clear and simple: write a symptom-driven program (it was not called a system at that time) that would emulate the problem-solving behavior of a network administrator. It was to formulate hypotheses about the location of faults and provide expert suggestions about how to clear faults. Like a human administrator, it was to learn from its experiences and improve its performance in time. Eventually, it was to go beyond consultation and perform fault-isolation and correction procedures completely without human involvement.

The project started with a prototype consulting system that specialized in handling a few network-element types. After a series of successful feasibility demonstrations, work went forward in extending its knowledge base and developing an interface to enable the system to communicate with an existing EMS and a live Datakit VCS network.

In several respects, the resulting product exceeded expectations. But there were also complications and compromises in the transition from prototype to product. One area of compromise was in the autonomy accorded the troubleshooter. Originally, the troubleshooter was designed to identify faults and, wherever possible, automatically correct them. Demonstrations of this capability to colleagues and prospective operators revealed a strong preference for human involvement in key decisions such as whether a particular service-affecting test should be conducted. These concerns led to the permissions scheme, discussed earlier, whereby the system seeks approval before performing any service-affecting operation.

The breadth and depth of the domain knowledge have been central concerns from the outset. Real-world operating environments imposed new constraints not felt in the laboratory environment where the troubleshooter had its start. As the system moved toward product realization, issues concerning *run-time performance* (execution speed) and testing and validation began to surface. Of course, these latter issues are common to conventional EMSs and most other software projects. As many of these issues were addressed, the troubleshooter became larger and more complex than originally envisioned.

This paper continues with a discussion of issues that shaped the development of the product and issues that remain to be resolved.

Comprehensiveness of Knowledge. Datakit VCS networks comprise vast numbers of hardware and software elements that interact in complex ways. Many elements have undergone significant change over several releases, and often several versions of the same element exist in a single network. Each element variant presents a slightly different *signature*, or pattern of distinguishing physical characteristics, parameters, and values (see Table I). The number of elements and element variants and the complex interactions among elements give rise to an enormous problem space. An enormous amount of knowledge would be required to interpret with precision all possible states within that space.

The basic question that concerned us was: How much knowledge is enough? Stated another way, how far should the system go in isolating a fault? How we answered this question would have direct bearing on cost and complexity of system test procedures, and would have maintenance and performance implications as well.

An individual circuit card may include dozens of specialized chips, switches, and other devices that could conceivably be replaced individually by highly trained technicians. And often there are tests and procedures that could be performed to isolate faults at this level. Yet most recognized maintenance procedures call for replacement of the entire circuit card when a hardware fault (e.g., circuitry abnormality) is detected. This being the case, it would be wasteful for the troubleshooter to contain in its rule base the procedural knowledge for identifying faulty subcomponents on a circuit card. However, it is practical to delve further into major subsystems, such as certain ver-

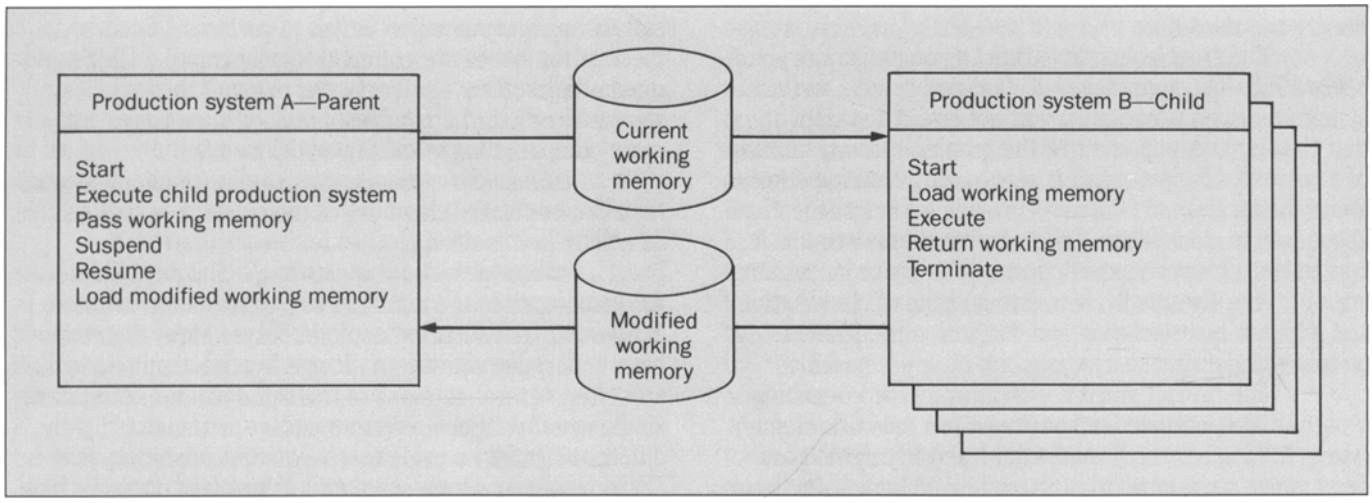


Figure 8. Distributed cooperating production systems.

sions of the power and cooling supplies, that consist of easily interchangeable parts.

In general, we limited the knowledge in the system to that which was required to correct the fault in the most generally accepted and cost-effective manner. The limits placed on the amount of information contained in the rule base constituted another compromise of sorts. But it was necessary to establish tradeoffs between the usefulness of domain knowledge and the burden the knowledge could place on system run-time performance and resource utilization.

Performance. A key concern in production systems is speed of execution. Production systems seldom attain the speed of conventional procedural systems. As indicated earlier, this is due largely to the processing overhead imposed by the match-act cycle that controls production-system execution. A performance deficit exists even though match-act algorithms usually are designed to minimize the number of comparisons required during each cycle.

There are rule-based programming techniques that can improve run-time performance and, to a lesser extent, resource utilization of production systems[10,11] in certain situations. More significant performance improvements will come from production-system implementions that are based on lower-level languages such as the C programming language. OPS83,[12] written in C and assembly languages, and C5,[13] written entirely in C, are production-system languages that can offer substantial improvements in run-time performance with minimal loss in the power and flexibility that characterized earlier languages such as OPS4.

Resource utilization, especially memory consumption, is another area of concern for production-system implementations. Encoding all domain knowledge within a single rule base has distinct advantages in terms of simplicity and overall system maintainability. Yet knowledge cannot be added to the existing rule base *ad infinitum* without taxing computer resources and, ultimately, degrading system performance. This is true regardless of the production-system language used to implement the rule base, although C-based production-system languages such as OPS83 are more economical than OPS4.

The troubleshooter's knowledge base must keep

step with the Datakit VCS network environment, which is growing and diversifying at an impressive rate. Several new pieces of equipment are to be introduced in the next year. With the anticipated domain growth and associated performance implications in mind, we have begun experimenting with alternative architectures that support distributed problem solving.

As an alternative to a single production system with a monolithic rule base, we are experimenting with multiple cooperating production systems. As shown in Figure 8, a circulated working memory has been used to coordinate the activities of two or more systems. Early exploratory work has involved the cooperation of two OPS4 systems (production systems A and B in Figure 8), as well as hybrid architectures consisting of OPS4 and C5 production systems.

A major goal of our ongoing exploratory work is to identify a graceful strategy for migrating from our current Lisp-based monolithic production-system architecture to a C-based distributed production-system architecture.

Validating Expertise. It is well-known that there is a significant element of subjectivity in testing and validating the expertise imparted by expert systems.[14] This should not be surprising, however. Political pundits, legal and economic advisers, and other experts on any given topic frequently disagree, often on seemingly fundamental tenets of their respective disciplines. Experts in data communications are no exception.

Any two data communications experts can observe the same evidence and formulate markedly different hypotheses about the location of a fault. In fact, this was observed during trials and demonstrations of prerelease versions of the troubleshooter. Experts were asked to comment on the effectiveness of its problem-solving behavior. While there was general agreement with its conclusions, some experts noted minor disagreements with its methodology. Most of these differences related to the order in which diagnostic procedures were executed. Currently, however, there is no evidence to suggest that experts disagree more with the troubleshooter than they do among themselves.

Our knowledge-engineering work concentrated on encoding the strategies of highly specialized individuals. For reasons already cited, we decided not to develop normative strategies for diagnosing each of the many elements represented in the system's knowledge base. There was no attempt to achieve agreement among panels of experts on any given topic. Validation efforts were designed to ensure that the system's behavior accurately reflected the strategies of the subject-matter experts who were consulted during the knowledge-engineering process.

Like its human counterparts, the troubleshooter can conceivably make an occasional inferential error. The permissions scheme discussed earlier in the context of system autonomy minimizes the impact of potential errors. Should the troubleshooter propose to do something unwarranted or erroneous in the judgment of the operator, it can be overridden. Thus, operators can prevail whenever the "professional opinion" of the troubleshooter differs from their own. In addition to providing a measure of security, it is hoped the permission scheme will engender a nonthreatening atmosphere in which expert operators and expert systems can collaborate in the problem-solving process.

Generality. The troubleshooter serves as an intelligent front end to StarKeeper NMS. There are several other voice and data EMSs in production at AT&T that might be augmented by expert-system front ends. Not surprisingly, the question of the troubleshooter's generality, or applicability to other domains, has arisen frequently. While the troubleshooter was designed for the Datakit VCS network environment, components of its architecture and its general approach to problem solving and adaptation appear suitable to other domains.

Currently, our colleagues within AT&T Bell Laboratories are adapting the troubleshooter to another networking environment. This exploratory work will prove useful in assessing the troubleshooter's utility as a platform for other real-time interactive expert systems. The major goal of platform development is software reusability, which has become important in increasing productivity in soft-

ware development projects and improving the maintainability of the resulting systems.

Conclusion

Starkeeper network troubleshooter is a real-time interactive expert-system product designed to be used by persons with widely varying skill and experience levels. It is one of the first systems of its kind available for commercial use in the data communications industry. Development of commercial-grade expert systems incorporates all the concerns over performance, human factors, reliability, and maintainability shared by developers of conventional systems. Additionally there are concerns over how best to cope with vast and expanding task domains, and how highly subjective qualities such as expertise should be represented and validated.

Feedback from field tests suggests the troubleshooter is extraordinarily easy to use and that it permits operators to deal successfully with many types of Datakit VCS network problems. This is encouraging because making difficult tasks appear easy is a central goal of expert-system technology. But networks are continually growing in size and complexity. Thus many challenges lie ahead in developing faster and smarter expert systems.

References

1. M. Al-Chalabi and W. J. Liss, "Design of the Bank of America California Data Network," *AT&T Technical Journal*, Vol. 67, No. 6, November-December 1988, pp. 87-106.
2. F. Hayes-Roth, D. A. Waterman, and D. B. Lenat, *Building Expert Systems*, Addison-Wesley, Reading, Massachusetts, 1983.
3. D. A. Waterman and F. Hayes-Roth (editors), *Pattern-Directed Inference Systems*, Academic Press, New York, 1978.
4. C. L. Forgy, "The OPS4 User's Manual," Technical Report CMU-CS-79-132, Computer Science Department, Carnegie-Mellon University, Pittsburgh, 1979.
5. T. E. Marques, "A symptom-driven expert system for isolating and correcting network faults," *IEEE Communications Magazine*, Vol. 26, No. 3, 1988, pp. 6-13.
6. J. K. Foderaro and K. L. Sklower, *The FRANZ LISP Manual*, Regents of the University of California, Berkeley, 1981.
7. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
8. *INFORMIX-ESQL/C Embedded SQL and Tools for C Programmer's Manual*, Relational Database Systems, Inc., Menlo Park, California, 1986.
9. H. A. Simon, "Why Should Machines Learn?," *Machine Learning: An Artificial Intelligence Approach*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (editors), Tioga Publishing Co., Palo Alto, California, 1983.
10. E. Charniak, C. K. Riesbeck, and D. V. McDermott, *Artificial Intelligence Programming*, Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1980.
11. L. Brownston et al., "Programming Expert Systems in OPS5," *An Introduction to Rule-Based Programming*, Addison-Wesley, Reading, Massachusetts, 1986.
12. C. L. Forgy, *The OPS83 User's Manual System Version 2.2*, Production System Technologies, Pittsburgh, 1986.
13. G. T. Vesonder, "Rule-Based Programming in the UNIX System," *AT&T Technical Journal*, Vol. 67, No. 1, January-February 1988, pp. 69-80.
14. P. Smith et al., "Validating Expert System Performance," *IEEE Expert*, Winter 1987, pp. 81-89.

(Manuscript received July 18, 1988)