FROM APPLICATION SHELL TO KNOWLEDGE ACQUISITION SYSTEM

Gary S. Kahn
Carnegie Group Inc.
Pittsburgh, Pa. 15219

ABSTRACT

The **TEST** (Troubleshooting Expert System Tool) architecture greatly aided the development of **TDE** (TEST Development Environment). In particular, the choice of a schematic as opposed to rule-based representation led to a knowledge base characterized by the use of domain-familiar concepts, and sufficient conceptual structure to facilitate several **TDE** features, including knowledge base development through both directed interviews and the direct manipulation of icons; multiple knowledge-based browsing strategies; heuristic error analysis; and easily-understood debugging techniques.

1. Introduction

TEST[1] (Troubleshooting Expert System Tool) is an application shell, providing a domain-independent diagnostic problem-solver together with a library of schematic prototypes. These prototypes constitute the object types and the structure required by each domain-specific **TEST** knowledge base. Several **TEST** applications, including those aimed at diagnosing engine problems, VAX/VMS performance, and factory floor machine failures are currently in development or field test.

Application shells allow developers to focus on knowledge acquisition, rather than on knowledge base design and problem-solving control issues of typically greater complexity. However, it is still a time-consuming process to interview experts and code a knowledge base. For this reason, several systems, including ROGET [1], ETS [2], and more recently, MORE [4], and SALT [6], have looked at ways to interrogate experts under the assumption that a chosen underlying architecture can solve the problems of interest.

TDE (TEST Development Environment) similarly enables knowledge engineers and trained domain experts to interactively build knowledge bases that can be used by **TEST**. **TDE** provides several advantages in comparison to MORE. In particular, it uses a problem-solving strategy that is more comprehensible to troubleshooting technicians in the manufacturing and customer service domains. It also addresses the need for knowledge acquisition systems to conform to the domain expert's desire to provide information as it comes to mind. Lacking this capability, MORE struck users as rigid and tedious in its interrogation technique.

[1]**TEST** is an internal name used at Carnegie Group Inc. **TEST** is implemented in Knowledge Craft[tm].

TDE knowledge bases represent the causal consequences of component and functional failures (failure-modes), in a manner similar to Heracles [3], as well as diagnostic methods, including the effective ordering of diagnostic tests. **TDE** acquires knowledge through automated interviews with domain domain experts. However, as a mixed-initiative workbench, **TDE** enables developers to provide information as they wish, while at the same time offering guidance and direction as it is needed. A number of other tools that aid in knowledge base development and modification are provided.

The following sections examine three facets of **TDE**: first, core concepts and the way they are used to build a knowledge base; second, knowledge-base modification; and finally debugging support. Each section shows how **TDE** design and development was facilitated by features of the **TEST** architecture.

2. Core Concepts

Knowledge acquisition is largely a matter of mapping the knowledge which supports expert decision making into the representations required by a problem-solving system. When there is a conceptual correspondence between these representational units and the terms with which experts understand their task and domain, mapping becomes a straightforward operation. In addition, conceptual correspondence makes direct-manipulation techniques readily available, and permits the system to easily guide users engaged in knowledge base development.

In the following, it is shown that **TEST** concepts, as well as conceptual relations, are familiar within the troubleshooting domain, and, as a result, communication between **TEST** and system developers is facilitated.

2.1. TEST Concepts

Unlike rule-based diagnostic systems that have evolved from the Mycin perspective, **TEST** uses a semantic network of schematic objects, or frames, to represent its key concepts. An attempt has been made to match these concepts to the ones operative for most troubleshooting technicians and many design engineers. Most critical is the **failure-mode**. A failure-mode represents a deviation of the unit under test from its standard of correct performance. Failure-modes are arranged in a hierarchy. At the top of the hierarchy are observable failures of the entire unit, e.g. hot air out of

an air conditioner. At the bottom of the hierarchy are failure-modes of individual components, e.g. a broken cooling unit. Intermediate failure-modes typically represent functional failures which are causal consequences of component failures. Many levels of intermediate failure-modes are common.

Failure-modes can be confirmed or rejected based on particular outcomes of **tests**. Tests are distinct nodes in the network. Other conceptual objects within **TEST** are **decision-points, test-procedures, repair-procedures, rules**, and **parts**, among others. [5] Each of these concepts has an obvious mapping into the troubleshooting domain. Rules in **TEST** represent a variety of contingent actions rather than evidence/belief propositions alone, as is typical in Mycin-like diagnostic systems.

2.2. Using Direct Manipulation

TDE provides a straightforward mechanism for creating knowledge bases. Within a workbench environment, icons representing **TEST**'s domain-familiar objects are easily moved into a representation of the knowledge base. A failure-mode icon, for instance, may be manually linked to another indicating that it is a cause of the latter problem.

Networks built in this way may also be edited as copy, move, and delete operations are available. Icon placement invokes prompts from **TDE** for information that requires keyboard entry, such as failure-modes names. Icons representing failure-modes and other key objects are located along the right edge of editing windows [see figure 2-1] through which the knowledge base may be viewed. While experienced users tend to use direct manipulation of graphic items as the preferred method, novice users of **TDE** rely heavily on system-directed interrogation techniques, described further below. Developers may move freely between system- and user-directed modes of operation.

Since the mapping between information provided and the underlying representation is direct, the system avoids the difficult problem of interpreting user intent when a mapping is required between inputs and internal representations. As the objects manipulated are the very objects that will appear in the knowledge base, there is similarly no issue of the developer being mystified as to the results of providing information to **TDE**.

A key problem for direct-manipulation techniques is that of determining and displaying relevant objects and object relations in a manner that's focused around the needs of the users task. Within **TDE**, **TEST** provides a natural tripartite representational structure for constraining the display and manipulation of the knowledge base.

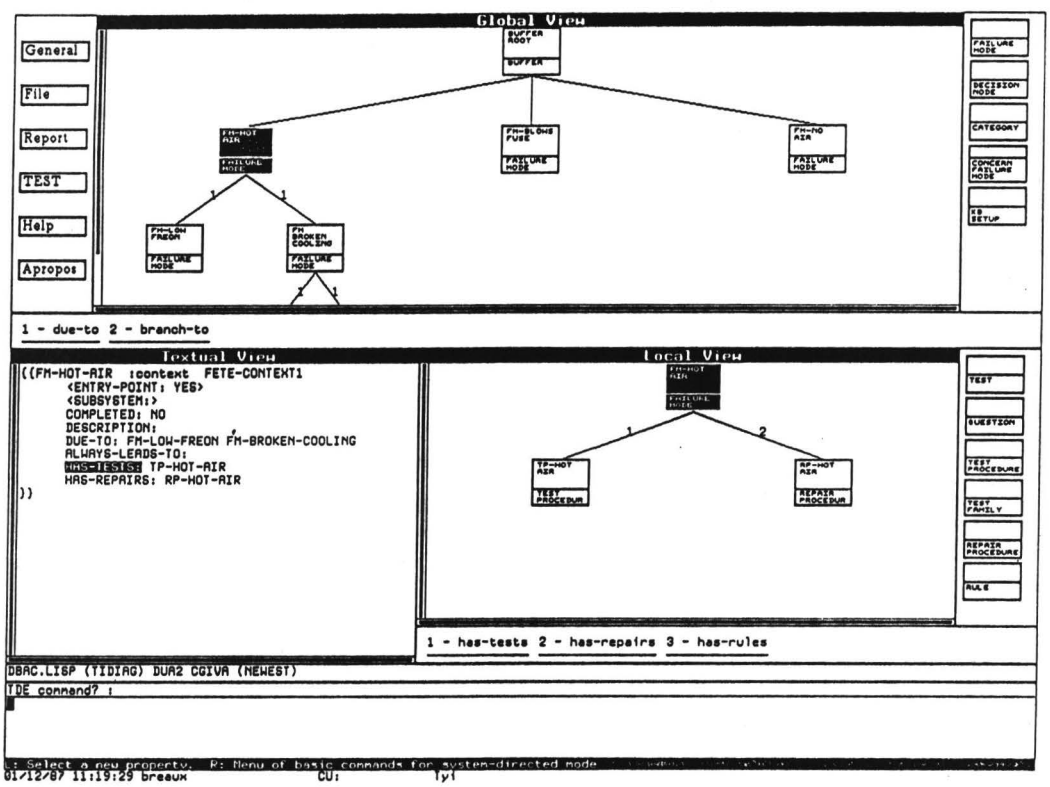

Figure 2-1: Screen Layout for **TDE**

Primary, secondary, and tertiary information can be distinguished within a **TEST** knowledge base. The primary core of a **TEST** knowledge base is a failure-mode tree, as described above. Other secondary information in the knowledge base -- including tests, repairs, and rules associated with the selected failure-mode -- is clustered around each failure-mode. At the tertiary level, each of these objects has itself numerous attributes providing further descriptive and control information.

TDE takes advantage of this tripartite division. A global view or window provides a filtered perspective of the knowledge base. Here failure-modes and the causal relations between them are displayed. A second window gives the user a local perspective into the secondary nodes clustered around a selected failure-mode. Finally, objects selected within these windows can be displayed with their full attributive detail in a third window. (See figure 2-1)

Figure 2-1 shows three views into a knowledge base regarding air condition failures. In the global view, the failure-mode hierarchy is displayed. The *hot-air* failure-mode is shown to be *due-to low-freon* or a *broken-cooling-unit*. The *hot-air* failure-mode has been selected for further examination. Consequently, secondary information local to it is displayed in the lower right. It is shown to have an associated *test*. If other secondary information, such as repairs or rules, had been associated with the failure-mode they would be displayed here as well. All attributes of the selected object are displayed in a structured text-editing window to the lower left. The knowledge base is extended simply by moving more icons into the appropriate window and/or by responding to prompts associated with attribute fields displayed in the text-editing object window.

2.3. Guiding Novice Users

TDE also exploits the inherent tripartite structure of a **TEST** knowledge base to provide guidance to developers. Each perspective delineates a natural line of interrogation. Before asking detailed questions, **TDE** will first flesh out the overall structure of the knowledge base as reflected in the global window. It will then seek to fill in detail at the level reflected in the local view. Finally, as necessary, **TDE** will cycle through the attribute slots of each object, completing the full picture. For example, **TDE** would have asked first about potentially observable air conditioner problems that would signify some kind of malfunction. For each of these, it would attempt to find out their possible causes, as they would be considered by a technician engaged in troubleshooting the initial concern. For each of these failure-modes **TDE** would inquire about associated diagnostic tests and repairs. It would then attempt to get additional attributive information on the reported failure-modes, repairs, and tests, as required.

As information is provided in response to prompts, **TDE** represents it graphically. Thus, the user can both see the system's interpretation of the information provided and follow the developing context in which further questions are asked.

3. Knowledge-Base Modification

In addition to facilities for building up a knowledge base, as described above, **TDE** offers further editing support. As developers make enhancements to pre-existing systems, they need special tools to help avoid errors which frequently lead to flawed knowledge bases. Typical errors include redundancy, incompleteness, and inconsistency. **TDE** helps by providing the user with the ability to browse quickly for information already in the knowledge base, and by detecting certain kinds of hard or suspected errors.

The explicit representation of causal and classificatory information in **TEST** enables approaches that would be precluded by classical rule-based diagnostic systems.

3.1. Browsing

Object classification, relational networks, and the maintenance of back-pointers within **TEST** enables multiple search techniques as well as efficient retrieval. As a result, **TDE** can offer developers string search, network browsing, and schematic pattern matching. Browsing facilities, such as these, are required when the user wants to confirm that a new (to be added) failure-mode is not already in the knowledge base, find a similar failure-mode which can be copied and edited to create the new object, or simply understand the content of what is already there. In a large knowledge base, it is impractical to browse without a means of directing attention to relevant segments of data.

Within **TDE**, string search is used to match against the names of knowledge base objects. Search of this kind is constrained by allowing the user to specify the type (class) of object (failure-mode, test, etc.). Network browsing is used to examine objects within the knowledge base that are in a specified relation to a specified object (e.g. 'has-test 'voltage-meter). Pattern matching is used to find objects with attribute values specified in the search template. In addition, inverse links maintained in the knowledge base enable **TDE** to quickly point users to parts of the knowledge base that may be impacted by their modifications.

3.2. Error Warnings

When developing a knowledge base, the user may unknowingly duplicate a failure-mode by using a different name to refer to one already known. While this can't be entirely prevented, the availability of richly structured information in a **TEST** knowledge base enables **TDE** to use heuristics to identify suspects. **TDE**, for instance, monitors for failure-modes which have similar causes and consequents, or those that share the same test and repair procedure. Isomorphic nodes within the knowledge-base network are typically unexpected and thus constitute grounds for suspicion.

TDE monitors for several other kinds of errors, as well. Of particular interest are objects that are not properly linked into the knowledge base. In addition, failure-modes without causal, test, or repair information are noticed. Violations of type restrictions associated with the attribute slots of each **TEST** object are also flagged.

4. Debugging Support

The **TEST** knowledge base is used directly by the diagnostic problem-solver (the TEST interpreter). As a result, debugging is facilitated. For one, it is easy to move between editing and execution, as it is unnecessary to compile the knowledge base into a special form. Secondly, it is easier to step, trace, modify and explain the program in respect to the use of the knowledge base than it would be if compilation into a runtime representation had been required.

In order to ease debugging by non-programmers, the **TEST** problem-solver is designed to use the knowledge base in much the way an informed technician would proceed with fault-isolation. This again is made possible by structuring problem-solving around the high-order concepts represented in the knowledge base. The troubleshooting task proceeds by focusing on an observed or suspected failure-mode. An attempt is made to determine whether the failure-mode has occurred in the unit under test. If the failure-mode has occurred, or if its status remains unknown, then the possible causes of the failure-mode are investigated to see if they have occurred. The search process is guided by an underlying representation of the order in which diagnostic experts explore possible causes for identified failure-modes. Heuristic rules may be inserted in the knowledge base to modify search behavior as runtime information is acquired.

4.1. Debugging Techniques

In debugging knowledge-based systems, it is desirable to have access during execution to everything the system has already concluded or come to know, as well as to its current hypotheses, planned tests, and queries. This allows the developer to notice when the system failed to conclude inferable information, is preparing to determine needless information, or acted incorrectly on known information. The **TEST** interpreter can provide this information on demand as the entire knowledge base is at its disposal.

In order to get at information at the right time, it is necessary to allow breakpoints at conceptually relevant junctures. Within the **TDE** environment, it is desirable to pause as particular failure-modes become the focus of attention. Since the problem-solver uses the knowledge base directly, developers can set breakpoints simply by toggling desired failure-modes.

5. Conclusion

The **TEST** architecture greatly aided the development of **TDE**. In particular, the choice of a schematic as opposed to rule-based representation led to a knowledge base characterized by the use of domain-familiar concepts, and sufficient conceptual structure to facilitate several **TDE** features. While much of **TDE** has been implemented, there is still much to do. During its continued development, users will be evaluating evolving prototypes. Because knowledge-engineering needs cannot be fully predicted, we expect to learn of many new requirements for **TDE** and perhaps **TEST**.

6. Acknowledgments

Edwin H. Breaux, Peter DeKlerk, Robert L. Joseph, Al Kepner, and Jeff Pepper contributed to the development of ideas presented in this paper. Many others have also been involved in the implementation of **TEST** and **TDE**.

REFERENCES

1. Bennett, James S. "Roget: A Knowledge-Based System for Acquiring the Conceptual Structure of a Diagnostic Expert System". *Journal of Automated Reasoning 1(1)* (1985).

2. Boose, J. Personal Construct Theory and the Transfer of Human Expertise. Proceedings of the National Conference on Artificial Intelligence, Austin, Texas, 1984.

3. Clancey, William J. "From Guidon to Neomycin and Heracles in Twenty Short Lessons". *AI Magazine 7(3)* (Summer 1986).

4. Kahn, G.S., Nowlan, S., McDermott, J. MORE: An Intelligent Knowledge Acquisition Tool. Proceedings of International Joint Conference on Artificial Intelligence, 1985.

5. Kahn, G.S. TEST: A Model-Driven Application Shell. Proceedings of AAAI-87, 1987.

6. Marcus, S., McDermott, J., Wang, J. Knowledge Acquisition for Constructive Systems. Proceedings of International Joint Conference on Artificial Intelligence, 1985.