FANSYS: A Computer Model of Text Comprehension and Question Answering for Failure Analysis*

Sergio J. Alvarado
Ronald K. Braun
Kenrick J. Mock

Technical Report CSE-93-4

*Funds for the support of this study have been allocated by the NASA-Ames Research Center, Moffett Field, California, under Interchange No. NCA2-721.

Table of Contents

Contents **Page**

List of Figures iv
Abstract vi
1. Introduction 1
2. Issues Addressed in FANSYS 2
3. System Architecture 5
    3.1 Overview of System Modules 5
    3.2 Indexing Hierarchies 5
4. Domain Knowledge Representation 13
    4.1 The DMS Fault Domain 13
        4.1.1 DMS Entities 14
        4.1.2 DMS States 16
        4.1.3 DMS Actions 19
        4.1.4 DMS Events 21
        4.1.5 DMS Procedures 22
    4.2 Case Representation 28
5. Comprehension of Input Text and Questions 36
    5.1 Case-Based Parsing 39
    5.2 Parsing Input Text 41
        5.2.1 Marker Passing 43
        5.2.2 The Parsing Algorithm 45
        5.2.3 Domain Specific Inference Strategies 47
        5.2.4 An Annotated Example 49
    5.3 Parsing Input Questions 76
        5.3.1 Question Patterns 77
        5.3.2 An Annotated Question Example 78
    5.4 Generating Answers to Input Questions 80
6. Memory Creation and Retrieval 84
    6.1 MOP Attributes 84
    6.2 Linking MOPs 86
    6.3 Comparing Cases 88
    6.4 Memory Creation Process 93
    6.5 Memory Retrieval Process 98
7. User Interface 109
8. Current Status 125
9. Future Work 126
    9.1 User Interface 126
    9.2 Integration with Other Tools 126
    9.3 Integration of Memory Structures 126
    9.4 Implementation Scope 127
    9.5 Knowledge Acquisition 127
    9.6 Model and Functional Based Reasoning 128
    9.7 Subjective Comprehension of Input Text 128
    9.8 Belief Inferences Based on Past Failure Analysis 129

10. Conclusion 130
11. References.....131
Appendix A: Case Descriptions
Appendix B: Detailed I/O Examples
Appendix C: Code Listings

List of Figures

Figure Title **Page**

Figure 2.1 Input Failure Description : Ring Concentrator Case 1 (RC.1) 3
Figure 2.2 Question Answering Session for Ring Concentrator Case 1 (RC.1) 4
Figure 3.1 FANSYS System Modules 6
Figure 3.2 FANSYS Flow of Control 7
Figure 3.3 FANSYS Link Types for the Power-Up Event 9
Figure 3.4 Sample Abstraction Hierarchy - Indexing of Concepts within Memory 11
Figure 3.5 Partial Memory Abstraction Hierarchy 12
Figure 4.1 Knowledge Representation in FANSYS 13
Figure 4.2 DMS Fault Domain Classes 14
Figure 4.3 DMS Entities 15
Figure 4.4 Entities: Memory Organization Example 16
Figure 4.5 Entities: Textual Mapping Example 17
Figure 4.6 DMS States 18
Figure 4.7 States: Memory Organization Example 18
Figure 4.8 States: Textual Mapping Example 19
Figure 4.9 DMS Actions 20
Figure 4.10 Actions: Memory Organization Example 20
Figure 4.11 Actions: Instance Example 21
Figure 4.12 DMS Events 21
Figure 4.13 Events: Memory Organization Example 23
Figure 4.14 Events: Textual Mapping Example 24
Figure 4.15 DMS Procedures 25
Figure 4.16 Procedures: Memory Organization Example 26
Figure 4.17 Procedures: Textual Mapping Example 27
Figure 4.18 Representing a Case in FANSYS 28
Figure 4.19 Failed Component Mapping for RC.1 29
Figure 4.20 Failure Mode Mapping for RC.1 29
Figure 4.21 Failure Cause Mapping for RC.1 29
Figure 4.22 Failure Detection Mapping for RC.1 30
Figure 4.23 Failure Correction Mapping for RC.1 (Short Term) 31
Figure 4.24 Failure Correction Mapping for RC.1 (Long Term) 33
Figure 4.25 Failure Effects Mapping for RC.1 34
Figure 4.26 Case Representation for RC.1 35
Figure 5.1 Example Mapping of Text for the Power-Up Event 36
Figure 5.2 Index Pattern Structure 43
Figure 5.3 Activation Marker Structure 44
Figure 5.4 Prediction Marker Structure 44
Figure 5.5 The Two Question Formats 78
Figure 5.6 Sample Hierarchy for I-M-Time-Out-Event.144.507 82
Figure 6.1 Sample GI Memory Hierarchy 87
Figure 6.2 Graphical Representation of Sample Query 90
Figure 6.3 Graphical Representation of Sample Case 90
Figure 6.4 Comparing Case and Query 92

Figure 6.5 GI Hierarchy After Adding RC Case to Memory..... 96
Figure 6.6 GI Hierarchy After Adding RC and GW Case to Memory..... 97
Figure 6.7 Traversal Paths for Example Query - Direct Retrieval 101
Figure 6.8 Traversal Paths for Example Query - Alternate Entry MOP 103
Figure 6.9 Traversal Paths for Example Query with Elaboration 106
Figure 7.1 FANSYS User Interface 110
Figure 7.2 Trace Window: Parsing RC.1 111
Figure 7.3 Query Construction via Menu: Selection of Requested Items 113
Figure 7.4 Query Construction via Menu: Selection of Failed Component.....114
Figure 7.5 Query Construction via Menu: Selection of Given Information 115
Figure 7.6 Query Construction via Menu: Selection of Failure Mode 116
Figure 7.7 Query Construction via Menu: Selection of Failure Cause.....117
Figure 7.8 Query Construction via Menu: Selection of Failure Detection 118
Figure 7.9 Query Construction via Menu: Selection of Failure Correction.....119
Figure 7.10 FANSYS Help Window Showing Query Format.....121
Figure 7.11 Sample of the Question Shell 122
Figure 7.12 Memory Inspector: Expansion of Ring Concentrator Case 1 124

Abstract

The research described in this technical report is aimed at extending previous theories of natural language processing and memory search and retrieval. In particular, this research has been concerned with the development of a model of text comprehension and question answering that uses case-based reasoning techniques to acquire knowledge of failure analysis from input text, and answer questions regarding diagnosis and repair of failures in complex systems, such as NASA's Space Station Freedom. The major goal has been to develop a process model that accounts for (a) how to build a knowledge base of system failures and repair procedures from textual descriptions that appear in NASA's FMEA (failure modes and effects analysis) manuals, and (b) how to use that knowledge base to evaluate causes and effects of failures, and provide diagnosis and repair procedures. This process model has been implemented in an experimental computer program called *FANSYS* (Failure ANalysis SYStem), which is a text comprehension and question answering system for the analysis of failures occurring within the data management system (DMS) of NASA's Space Station Freedom. This project received funds from NASA Ames Research Center to support three major tasks in a period of a year: (1) analysis of conceptual content of system-failure manuals; (2) characterization of the knowledge structures and processes underlying the computational analysis of failures; and (3) implementation of an experimental version of FANSYS that reads a few segments of the manuals, and answers a number of questions involving failure diagnosis and repair.

1. Introduction

A goal of artificial intelligence is to develop computer systems that exhibit skills similar to those used by humans during language comprehension, language generation, planning, reasoning, and argumentation (Alvarado, 1992). In the case of intelligent systems designed to aid in complex decision making and the generation of expert advice, it is necessary that they be able to evaluate given goal situations, present their beliefs on possible plans of action, justify their beliefs, understand opposing beliefs, and argue for/against given plans. These skills are needed because it is unlikely that any individual seeking expert advice would unquestioningly trust or follow the suggestions of systems that can propose a course of action, but cannot justify such proposals, and cannot understand or counter those objections that may be raised to their suggestions.

A step towards modeling the interrelationships that exist among reasoning, planning, and language comprehension has been taken by Alvarado (1990) with the development of a prototype editorial comprehension and question answering system called *OpEd*. In this system, the process of text comprehension is viewed as one of managing many different knowledge sources, such as causal chains of reasoning, goals, plans, actions, characters, beliefs, belief relationships, and argument units. Associated with each knowledge construct are one or more processing strategies that are invoked by a conceptual parser (Dyer, 1983) in order to build a network of beliefs (Flowers et al., 1982) that represents the content of the input text. During question answering, the system uses a question-categorization scheme (Lehnert, 1978) and retrieval heuristics (Dyer and Lehnert, 1982) to analyze each input question and select specific strategies of search and retrieval.

The research described in this technical report is aimed at extending the theories of natural language processing and memory search and retrieval underlying OpEd's design. In particular, this research has been concerned with the development of a model of text comprehension and question answering that uses case-based reasoning techniques (Riesbeck and Schank, 1989) to acquire knowledge of failure analysis from input text, and answer questions regarding diagnosis and repair of failures in complex systems, such as NASA's Space Station Freedom. The major goal has been to develop a process model that accounts for (a) how to build a knowledge base of system failures and repair procedures from textual descriptions that appear in NASA's FMEA (failure modes and effects analysis) manuals, and (b) how to use that knowledge base to evaluate causes and effects of failures, and provide diagnosis and repair procedures. This process model has been implemented in an experimental computer program called *FANSYS* (Failure ANalysis SYStem), which is a text comprehension and question answering system for the analysis of failures occurring within the data management system (DMS) of NASA's Space Station Freedom*. This project received funds from NASA Ames Research Center to support three major tasks in a period of a year: (1) analysis of conceptual content of system-failure manuals; (2) characterization of the knowledge structures and processes underlying the computational analysis of failures; and (3) implementation of an experimental version of FANSYS that reads a few segments of the manuals, and answers a number of questions involving failure diagnosis and repair.

---

*See Barlett et. al (1992), McDonnell Douglas (July, 1990), and IBM (July, 1990) for complete descriptions of the DMS.

2. Issues Addressed in FANSYS

The problem of analyzing and diagnosing failures that occur in mechanical or electronic devices has been addressed by other researchers in artificial intelligence and computer science. Some of their work has been concerned with the development of truth-maintenance systems (de Kleer and Williams, 1987; Struss, 1988), expert systems (Reed and Johnson, 1990), connectionist expert systems (Liu et al., 1991), case-based reasoning systems (Hammond and Hurwitz, 1988), and digraph-based analysis systems (Iverson and Patterson-Hine, 1990; Patterson-Hine and Iverson, 1990; Stevenson, Miller, and Austin, 1991). However, these researchers have not addressed the task of comprehension of natural language in descriptions of failures, which requires dynamically constructing a knowledge base of the failure cases from textual input. In contrast, the FANSYS project has been concerned with the domain knowledge, conceptual representations, natural language processing strategies, and search and retrieval strategies involved in understanding textual descriptions of failure cases and answering questions about such cases. FANSYS involves the use of techniques for parsing input failure descriptions into a network of cases that maintains the context for subsequent question answering.

Text comprehension and question answering in FANSYS involve four major tasks: (1) applying domain-specific knowledge (i.e., DMS fault domain); (2) mapping input text into conceptual structures that compose the internal representation of failure-analysis cases; (3) representing and indexing failure-analysis cases in memory by creating a library of cases; and (4) using the library of failure-analysis cases to answer questions regarding failure causes and effects, as well as failure detection and correction procedures. Input failure descriptions correspond to segments of FMEA manuals, and contain the essential wording of the original descriptions. Here "essential" means that the original descriptions have been edited to remove those parts which involve addressing issues that fall outside the scope of FANSYS (e.g., references to specific part numbers, references to supporting documentation, and descriptions of functions of DMS components). Figures 2.1 and 2.2 illustrate the current input/output behavior of FANSYS. The input failure case shown in Figure 2.1 is a fragment of a failure description appearing in the FMEA document number MDC H4563A (McDonnell Douglas, March 1990). A question answering session using this case is shown in figure 2.2. The computational resources needed to support this type of input/output behavior will be described in the rest of this technical report.

Insert Figure 2.1

Insert Figure 2.2

3. System Architecture

At a high level, FANSYS may be decomposed into five major modules: *user interface*, *case-based parser*, *memory update and retrieval*, *main memory,* and the *natural language generator*. An overview of these modules and their interactions is presented below and in figure 3.1.

3.1 Overview of System Modules

User interaction with FANSYS is handled through the user interface module. The interface has been designed to ease user tasks by accepting both textual and menu-driven input. The case-based parsing module is responsible for reading input case descriptions and questions, and building an internal memory representation for what has been read. The memory update and retrieval module determines where cases should be stored in memory and how those cases will be retrieved during question answering. Once cases have been retrieved during question answering, the natural language generator converts the internal representation into English text.

Although the design of FANSYS involves a modular architecture, its modules work in a highly interactive manner. This interaction is illustrated by the flow of control diagram shown in figure 3.2. This diagram indicates that during knowledge acquisition, control flows from the user interface to the parser, and then to the memory module as each case description is parsed and indexed in memory. The diagram also shows that, during question answering, control flows either (1) from the user interface to the parser and then to the retrieval and generation modules, or (2) from the user interface to the retrieval and generation modules. Furthermore, the diagram indicates that both phases of processing in FANSYS require repeated access to main memory to store or retrieve cases.

Figure 3.2 also shows that user interaction with FANSYS occurs primarily during the question answering phase. The user interface supports a variety of options, including the ability to browse through the representations created in memory, obtain help or system information, and ask the system questions about what it has read via natural language or point-and-click buttons. If a question is given in natural language, then the system will parse the question into a memory search query. Conversely, the user may construct the search query by clicking on buttons which represent known objects or procedures. Once a search query has been created, the memory retrieval module will search memory for cases which answer the query. These cases will then be output in English by the natural language generator. If desired, the user may instruct the system to search for additional cases that may be relevant to the search query.

3.2 Indexing Hierarchies

The most elaborate components of FANSYS are the parsing module, update and retrieval module, and the indices which both of these modules use to access memory. Both modules use a set of hierarchical indices to access and organize memory. Memory is organized in terms of Memory Organization Packets (MOP) (Schank, 1982; Riesbeck and Schank, 1989), where each MOP represents a concept. Each MOP is a frame (Charniak, 1977, 1978; Minsky, 1975, 1977) of slots and fillers, where the slots and fillers are defining attributes of the concept. The most

Insert Figure 3.1

Insert Figure 3.2

general MOPs occupy the topmost levels of the hierarchy. More specific concepts are indexed below their more general parents. This indexing scheme continues throughout the hierarchy, with the most specific instances of concepts ultimately indexed at the bottom. These specific concepts are called Instance MOPs; they are denoted I-MOP. All MOPs which are not instances (i.e., they are abstractions of some instance) are denoted M-MOP.

Although concepts in a MOP hierarchy are organized from the general to the specific, there are other relationships among MOPs used in FANSYS's memory. These relationships determine four memory hierarchies associated with the following types of links: ISA links, Slot-Filler links, Lexical links, and Generalization/Indexing links. For example, in figure 3.3 all four of these index types are depicted in relation to the event of powering up a DMS component. In this figure, the system's memory contains two instances of powering up an object. One of the instances involves a software entity called System-Management (SM) powering up a DMS component (I-M-Power-Up-Event.1), and the other instance involves the space-station crew powering up another DMS component (I-M-Power-Up-Event.2). These instances are connected to the abstraction of a general power-up-event (M-Power-Up-Event) via ISA links, which are depicted by arrowheads. These arrowheads do not indicate indexing or memory access direction, but serve only to differentiate the ISA links from other types of links. All links, including the ISA links, are bi-directional. ISA links connect sub-concepts to their parent concepts. In this example, a "System-Management powering up an object" is a type of "DMS-Entity powering up an object."

The Slot-Filler links connect concepts which are related to attributes of another concept. In figure 3.3, slots (i.e., attributes) of the Power-Up-Event include an actor performing the power-up action, an object being powered-up, preconditions for the event to occur, an action which is performed, and a state resulting from the action. All of these slots are constrained by their fillers (i.e., values). Only MOPs which are specializations of the filler may occupy that slot. In the Power-Up-Event example, only DMS-Entities may be the filler for the actor slot in a power-up event. Similarly, the Power-Off state is a precondition for the event, the Power-On state is a resulting postcondition, etc.

Lexical links connect all lexical information used to understand text with their respective concepts and index patterns. In the Power-Up-Event example, the word "power" is linked to one index pattern that contains "power" via lexical activation links. The figure shows two index patterns, one where an actor powers up an object, and the other where an actor powers down an object. The power-up pattern is linked to the power-up event through a lexical reference link. By traversing these links, the concept of power-up-event can be activated when the single word "power" is encountered while reading text. These indices are discussed in detail within the parsing section.

Finally, Generalization/Indexing (GI) links function similarly to both the ISA and slot-filler links, except they organize generalizations across cases and organize cases by their differences. GI indices are depicted in figure 3.3 by a triangle and dashed lines. In this example, the two power-up instances are indexed by the ways in which they differ. I-M-Power-Up-Event.1 is only accessible through the index of Actor=SM, while I-M-Power-Up-Event.2 is only accessible through the index of Actor=Crew. This simplified figure only shows one type of GI index; there may also be other indices for the object, precondition, action, and result slots. Furthermore, the

Insert Figure 3.3

GI indices are only depicted here by a triangle; however, the GI indices are not simple indices, but may be comprised of an entire hierarchy of MOPs. This GI hierarchy is very useful for quickly accessing memory during question/answering, and is discussed in detail within the memory retrieval section.

The previous example showed how the different links interact with one another to index and represent concepts in memory. However, figure 3.3 does not show how concepts are represented in memory as a whole. In contrast, figure 3.4 is a more comprehensive diagram which shows how the Power-Up-Event is represented in relation to M-Root, the conceptual abstraction of all concepts. In this figure, the dashed line encircles the example shown previously in figure 3.3. The Power-Up-Event is indexed under the more abstract category of Power-Events, which is a type of Event, which ultimately links to the Root. Similarly, a hierarchical organization is shown for index patterns, words, and the lexicon. There are also connections between ISA links and Slot-Filler links; i.e., an abstraction hierarchy also exists along the Slot-Filler links. These are depicted in the figure by the ISA links from I-M-SM.1, an instance which is the filler for a slot of I-M-Power-Up-Event.1. I-M-SM.1 generalizes up the ISA hierarchy to the abstraction of M-DMS-Entity, which is a a filler for a slot of the more general Power-Up-Event. While not shown in the figure, similar links exist for the other fillers between abstractions and instances.

Figures 3.3 and 3.4 illustrate the different types of indices and how these indices organize concepts in memory. The same memory organization scheme is used for organizing entire failure cases, where each failure case contains the failure mode, correction procedure, detection procedure, failure causes, and failure effects for a given device. Failure cases are organized hierarchically through the ISA and GI links. These link hierarchies are shown for a portion of memory cases in figure 3.5. The GI index hierarchy is shown in dashed lines and triangles while the ISA hierarchy is shown with solid arrows; Lexical and Slot-Filler links are not shown. This figure shows how multiple instances of the System Management, I-M-SM.1 and I-M-SM.2, are indexed under the System Management MOP M-SM. An instance of the Power-On-Self-Test, I-M-POST.1, is indexed in an identical manner under the MOP M-POST. Similarly, a failure case regarding the Gateway, instance I-M-GW.1, is indexed under the Gateway MOP, M-GW. Entire cases are indexed in an identical manner under the Case MOP M-Case. Figure 3.5 also shows one other case, an instance of a ring concentrator failure case called I-M-RC.1.

Insert Figure 3.4

Insert Figure 3.5

4. Domain Knowledge Representation

One of the first tasks confronting a researcher in constructing a natural language understanding system is to characterize the domain in which the understanding system is to operate (Schank, 1975, 1978). Once this domain has been characterized, a representational scheme must be adopted that will allow the researcher to encode that domain knowledge in a manner that permits the system to perform the required understanding task (Alvarado, 1990; Carbonell, 1981; Schank and Carbonell, 1979; Wilensky, 1986). In FANSYS, the model of domain knowledge used during text comprehension and question answering consists of two types of representational classes: one encompassing the specifics of the domain of space-station DMS faults, and one characterizing the fault cases which are described in the FMEA manuals (see figure 4.1). Although these classes differ in organizational function, they are each represented using the same frame-based representational structure of the MOP. The class that organizes DMS-fault knowledge characterizes the entities and relationships found within the DMS fault manuals, and enables FANSYS to parse case descriptions from such manuals, and perform limited inferences and reasoning within the domain of the space-station DMS. In contrast, the class that encodes the fault cases organizes a library of fault descriptions that is built up during parsing and queried during question answering. Both of these classes are indexed under M-ROOT, the top-most abstraction MOP which organizes memory. This section will consider these two classes of domain knowledge in detail.

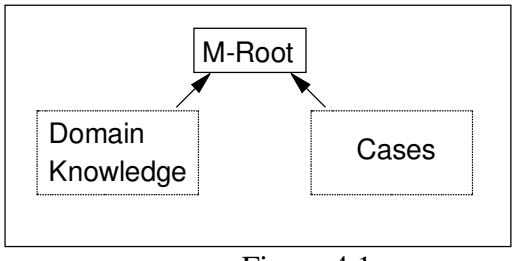

Figure 4.1
Knowledge Representation in FANSYS

4.1 The DMS Fault Domain

After having analyzed the domain of failure effects and procedures described in the FMEA manuals, it was determined that the DMS fault domain could be effectively modeled by five conceptual types of knowledge structures: entities, states, actions, events, and procedures (figure 4.2). These classes provide a detailed enough representational scheme of the domain to encode the events, relationships, and procedures underlying the textual descriptions of fault cases in the manuals. The case descriptions that are built by FANSYS are in turn founded upon these five types of domain knowledge. These types will be considered individually below.

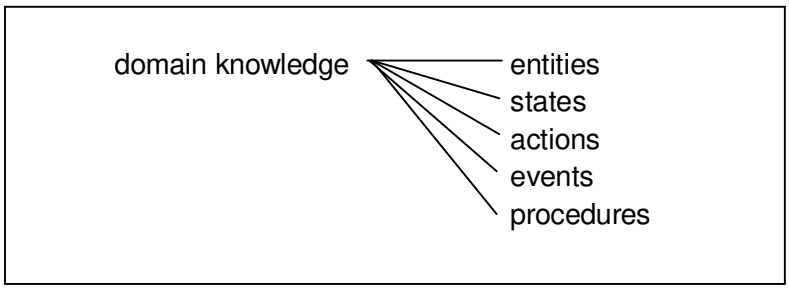

Figure 4.2
DMS Fault Domain Classes

4.1.1 DMS Entities

Entities are defined as the hardware and software components of the DMS system. There are two broad categories of DMS entities: systems and messages (figure 4.3). Systems are the physical devices of the DMS, such as an ORU (Orbital Replacement Unit), a token ring network, or an ORU's local system management software. Systems may be composed of hardware and software components, which may themselves be systems, to whatever granularity is required to model any of the textual descriptions.

As figure 4.3 shows, FANSYS recognizes several categories of systems. The mission components are the four systems that are effected by failures described within the FMEA manuals (for example, see the "Failure Effects" section, figure 2.1). The different networks of the DMS, as well as the DMS itself, are explicitly represented as two other categories of systems. The hardware components are the physical devices that comprise the DMS and its subcomponents, such as individual ORUs (e.g., ring concentrators and gateways) and the power systems. Finally, the software components of the DMS represent the software support systems, including system management software, error correcting procedures, and the like. These categories of systems effectively model the physical components of the DMS.

The second entity described in figure 4.3 is that of the message. Messages are the entities used in communication between or within systems. A token and a heartbeat message in a token ring network represent this type of entities. Messages are typically passed between systems within the DMS, carrying status or other information, and are differentiated from software systems in that they are not active agents within the DMS network. Rather, they represent passive constructs that are manipulated by systems.

Figure 4.4 shows a part of the ISA and Slot-Filler hierarchies for some sample entities. Notice that hardware components may be recursively defined as consisting of other hardware and software subcomponents. For example, a ring concentrator (a hardware component) may consist of a transmitter (a hardware component) and a local system management program, as well as other hardware and software subcomponents. Likewise, software components may consist of several other software subcomponents. (System management software typically has application management software as a subcomponent, for example, although this is not reflected in figure 4.4.) Such systems are defined to the level of granularity required to parse the case manuals; i.e. there is no need to represent a transmitter at the level of the atoms that compose it, since those atoms are not referenced within the FMEA manuals.

Insert Figure 4.3

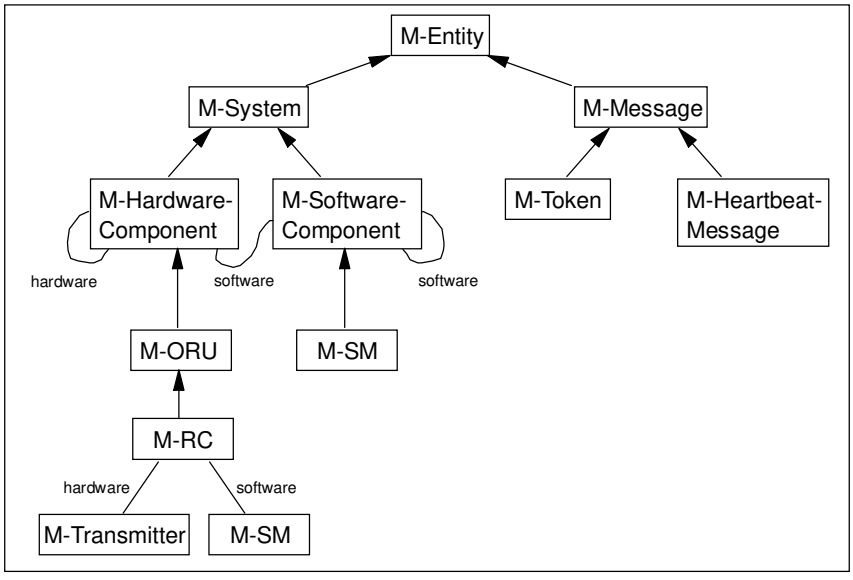

Figure 4.4
Entities: Memory Organization Example

To see how text is mapped into the class of entities, consider the sentence "system management reaches a time out limit for receipt of the network token from the ring concentrator." This text references two systems and one message: the ring concentrator and system management, and the network token, respectively. Figure 4.5 shows a concept instance for each of these entities from the text, and the actual text that maps into those instances. (The exact mechanism by which text is parsed into memory will be detailed later in section five on parsing.) Notice that as the ring concentrator is instantiated in memory, its subcomponents may be instantiated as well by inference. Thus, by explicitly representing concepts and their components in memory, the domain representation allows FANSYS to make inferences and connections not explicitly found within the text.

4.1.2 DMS States

A state represents a property of a DMS component. It may apply only to an individual component, or it may express some relationship between components. As shown in figure 4.6, there are eight states that are pertinent to the DMS domain. The first six encode physical properties for a specific DMS entity, including its structural integrity, whether it is operating within an acceptable temperature range, whether it has been contaminated, whether it is powered on or not, whether it is operable or faulty, and whether it has passed or failed a given diagnostic that has been performed on it. The other two states encode relationships between two entities of the DMS, such as a description of the other components to which a given device is logically or physically connected, and the current state of any communications between two entities. These states have been found to sufficiently characterize the state of the DMS system during the understanding process of the FMEA manuals.

Insert Figure 4.5

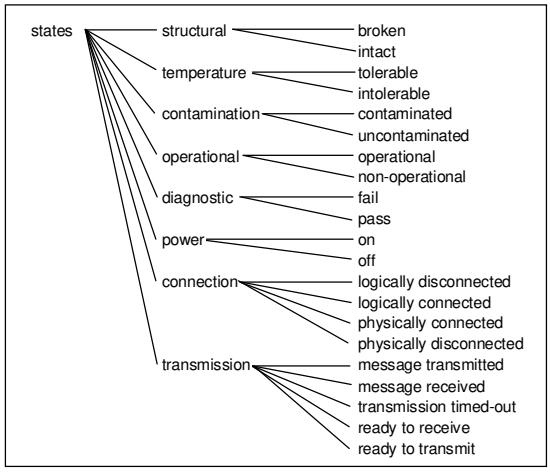

Figure 4.6
DMS States

Figure 4.7 illustrates a fragment of the ISA and Slot-Filler hierarchies for a few states. Notice that each of the states involving a single entity has a single slot detailing the object to which the state applies (e.g., m-power-on), while each of the states involving relationships between multiple entities has a couple of slots detailing the entities to which that state applies (e.g., m-physically-connected). These slots are expressed at an abstract level for each state, i.e., since slots are inherited down the ISA hierarchy, each of the subMOPs for that class will also inherit those slots.

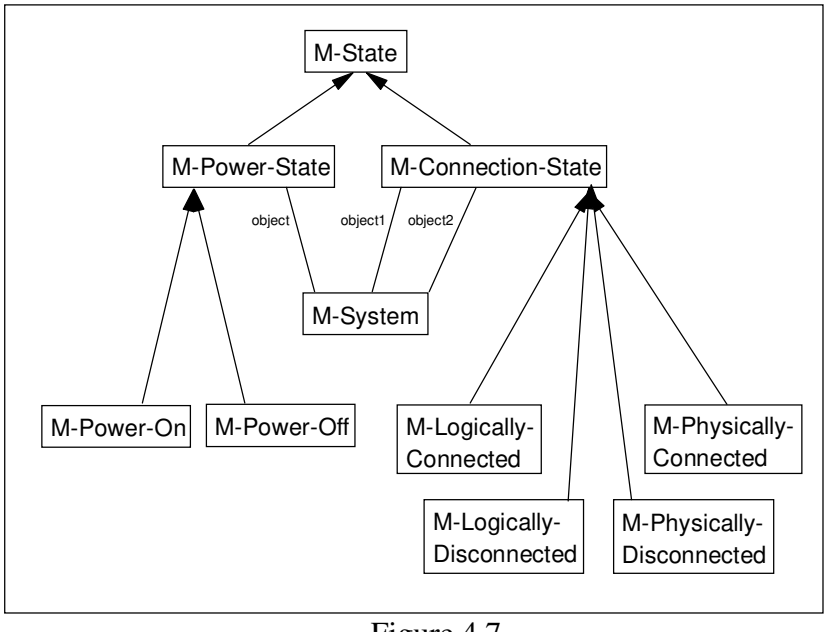

Figure 4.7
States: Memory Organization Example

The mapping of text into state structures is exemplified by figure 4.8, which shows the instances created for the parsing of two sentence fragments: "the power for the standard data processor is on" and "the ring concentrator is connected to the primary network." Notice that the

devices mentioned in the fragments map into entities, while the textual state assertions map into the states themselves. The text will generally specify the relationships of various subcomponents for a MOP to that MOP. For example, the first fragment specifies that the standard data processor fills the object slot of the m-power-on state.

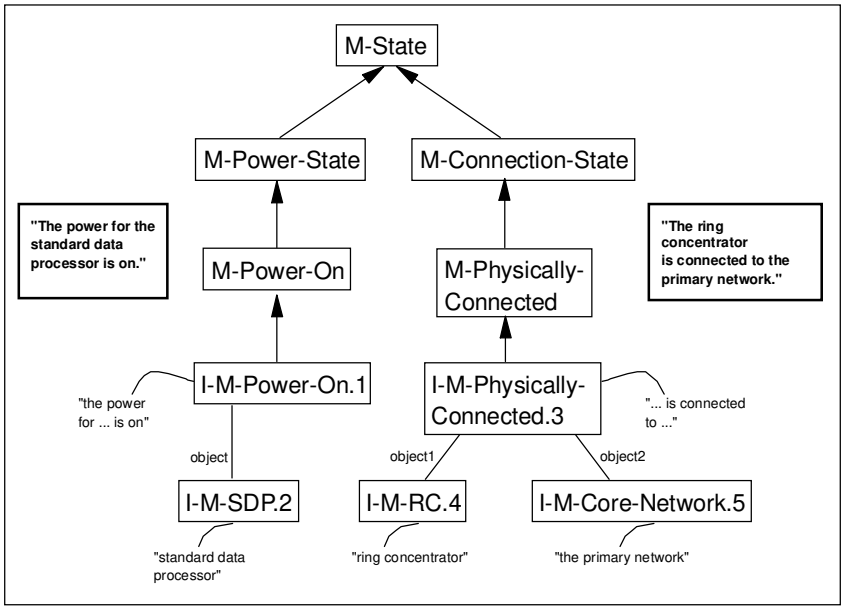

Figure 4.8
States: Textual Mapping Example

In practice, states are rarely mentioned explicitly in the manuals that were studied for the FANSYS project. Rather, they are usually left implicit as the results or preconditions of the events which are described within the manuals. Although left implicit in the text, these states must be explicitly represented in memory. For example, the result of powering on a ring concentrator is that the ring concentrator is in a state called m-power-on (perhaps with a precondition of being in the state of m-power-off). This relationship between states and events will be clarified shortly when the representation for events is discussed.

4.1.3 DMS Actions

Actions may be defined as acts performed by or on a DMS component. As with states, actions fall into a few well defined types, including powering actions, diagnostic actions, connection actions, and transmission actions (see figure 4.9). Powering actions involve turning on or off components, whereas diagnostic actions typically involve the performance of some sort of diagnostic test on a device. (For example, there is a diagnostic called the Power-On Self Test that is executed at powering-up time for some components, which determines if the component has become successfully operable.) Connection actions deal with establishing physical or logical connections between different devices. Transmission actions cover the transmission and reception of messages, and occurrences during communication such as the timing out of a device.

In addition to these types of actions, there are also some individual actions that describe operations on DMS systems. One of such actions represents the moving of something from one physical location to another, such as from the ground to the Space Station. Another such action

encodes the physical verification that a system is in the state that it is supposed to be in, e.g., that the connectors on a unit are properly connected. The last such action represents the looking-up of a backup unit in system tables by the system management for selection during the switchover to a redundant unit.

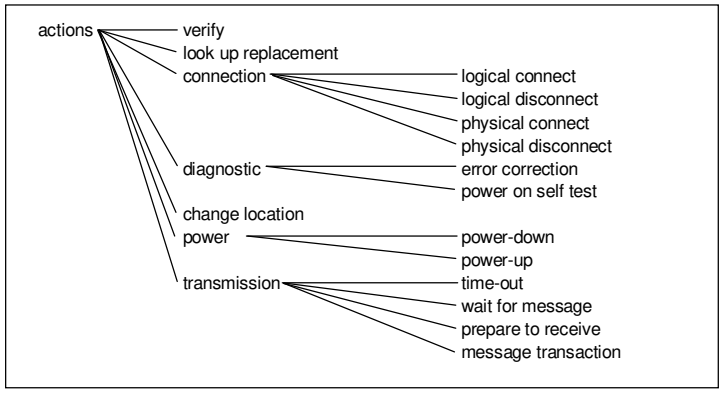

Figure 4.9
DMS Actions

Actions always involve an actor, and one or more objects that are acted upon. See figure 4.10 for some sample action MOPs, and the ISA and slot-filler hierarchies which characterize them. Note that while the powering actions have the usual object and actor slots that are inherited by all actions, the transmission actions have two additional slots detailing the sender and receiver of the transmission events.

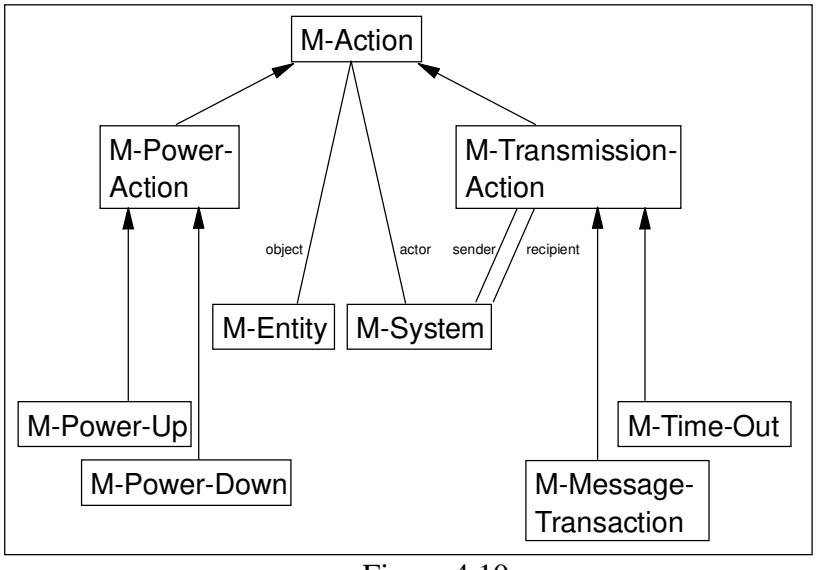

Figure 4.10
Actions: Memory Organization Example

In contrast to the prior types of domain representation that have been considered, actions are never mapped to directly by the parser. Rather, they are used as components of individual events; the events themselves are mapped to from the text. Actions thus represent atomic building blocks upon which events are founded. Figure 4.11 shows some sample action instances

that might be created during the recognition of the events which encompass them. Events will be considered in detail next.

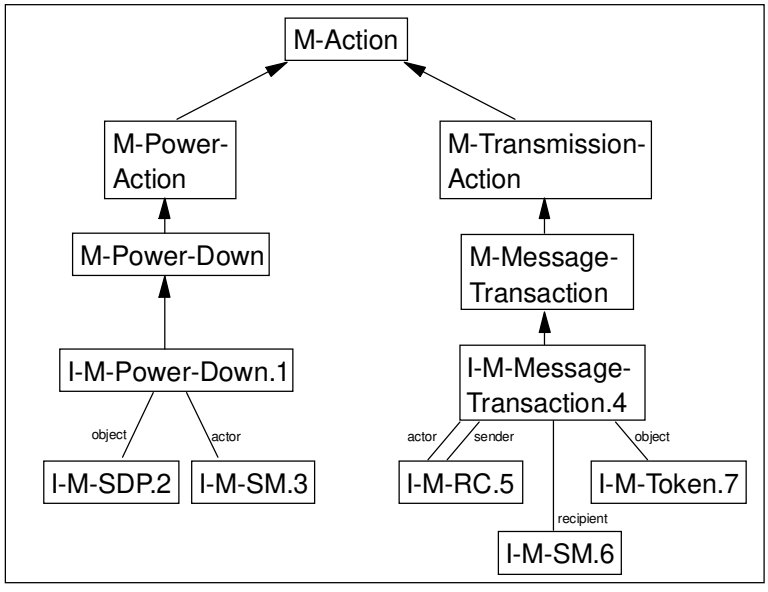

Figure 4.11
Actions: Instance Example

4.1.4 DMS Events

An event represents the occurrence of some action and the resulting state change that takes place because of that action. Each of the actions detailed in figure 4.9 has a corresponding event that describes the state transitions for that action; the categories of events are thus identical to the categories of actions (see figure 4.12). To illustrate this, consider the class consisting of powering events. Powering events involve transitions between power states based on the powering actions. For example, the event m-power-down-event packages the m-power-down action together with the precondition that the power be on initially (the state m-power-on) and the result that the power be off afterwards (the state m-power-off). The other classes of events likewise specify state transitions based on the actions to which they correspond. Since the precondition and results of events are seldom mentioned explicitly in a text, they must be explicitly represented in memory. This facilitates reasoning based on world knowledge and the generation of inferences regarding the current state of the DMS systems.

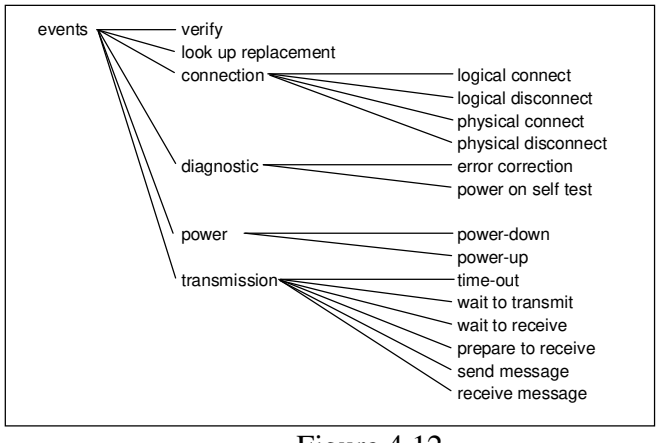

Figure 4.12
DMS Events

Since an event characterizes state transitions based on an action, an event may be defined as an action, that action's preconditions (which are each states), and the resulting states that follow from that action. Figure 4.13 depicts this slot-filler relationship, as well as other sample parts of the ISA and slot-filler hierarchies for the event class. Note that every event has one (or more) preconditions and results associated with an action at the abstract level. As the classes of event MOPs become more specified, the individual filler constraints also become more specified as to exactly which states and actions are involved in characterizing the associated MOP.

Text from the FMEA manuals that describe events map directly into the event MOPs, as demonstrated by figure 4.14. The sentence "the system management powers down the standard data processor" causes an instance of the m-power-down MOP to be created. The action, preconditions, and results associated with that MOP are inferred by virtue of the fact that the event was recognized to have occurred. FANSYS is thus able to track state changes during the parsing of text, a necessary step to the understanding task for which the system was designed.

4.1.5 DMS Procedures

The final and more complex conceptual type required to represent the DMS fault domain is that of the procedure. A procedure represents an ordered sequence of events or other procedures involved in some task described by the FMEA manuals. Figure 4.15 details the different categories of procedures (also sometimes called sequences-of-events within the FANSYS representation) defined within the system. Failure causes encode the events that lead to the failure of an ORU. These events are not actually specified within the manuals, save by named references like "piece-part failures" (see the "Failure Causes" section of the sample case manual in figure 2.1). For this reason, the actual set of steps for each of the failure causes is currently left undefined in the system. These steps could be fleshed out if more documentation that identified them became available.

The failure modes describe the events that occur when a failure is first detected. For example, a device may fail to become operable during the startup procedure for that device. Like the failure causes, the actual specifics of the procedure are left largely undefined, since they are not well specified within the FMEA manuals.

The failure detection procedures encode the specific steps used to detect the failure of a device within the DMS. For example, the m-next-node-detection procedure describes the case where a token circulating around a token ring network is not received by a specific ring concentrator; the implication is that the ring concentrator immediately upstream of it has failed. The last class of procedures is that of the failure corrections. These procedures describe the steps that need to be taken in order for a failure to be repaired. In the case of the failed ring concentrator just described, the correction procedure might be m-bypass-node, which consists of the automatic selectover to a redundant ring concentrator by the DMS system management.

Insert Figure 4.13

Insert Figure 4.14

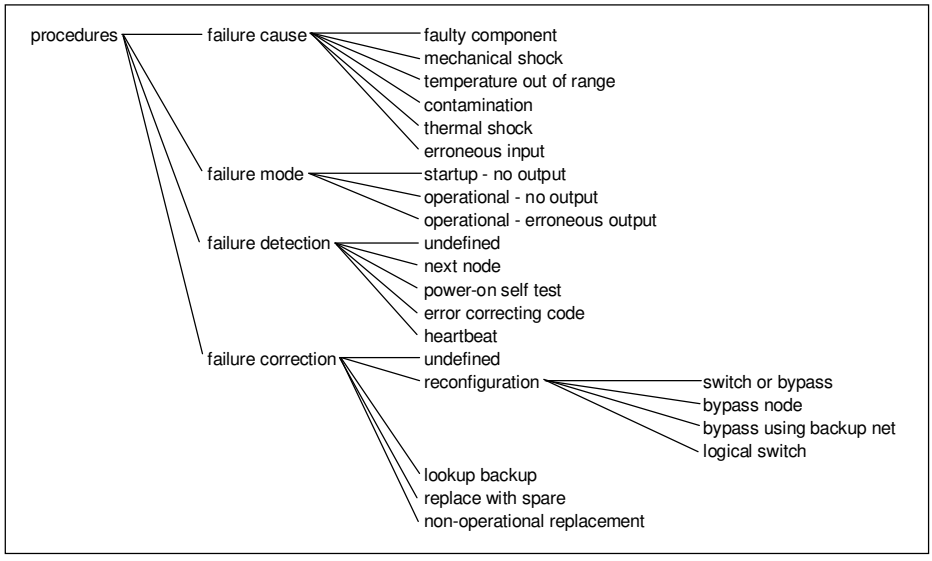

Figure 4.15
DMS Procedures

Figure 4.16 shows some simplified fragments of the ISA and slot-filler hierarchies for procedures. Like events, procedures have preconditions and results that are specified as state assertions; a procedure is in effect a complex series of state transitions (made by individual events) that lead from some starting set of states to some final set of states. For example, the preconditions for the m-next-node-detection procedure include the fact that the devices involved in the procedure are actually connected to the network physically and logically, while the result is that a failure has occurred. The sequence of steps details exactly what group of events (or other procedures) occurs in the procedure. Notice that a detection procedure actually consists of three subprocedures: the events in the detection process, the events necessary to verify that the failure has indeed occurred, and the final notification events to inform the DMS management of the failure. (Because only the detection steps are detailed in the manual cases with which the system currently deals, the notification and verification procedures are left undeveloped.)

One example of the detection process is illustrated in figure 4.16 by the steps for m-next-node-detection, which detail how a ring concentrator waits to receive a token from another ring concentrator, before eventually timing out. This process indicates a failure may have occurred in one of the ring concentrators. In addition to the slots specified in the figure, there are other slots not shown. For example, failure detection procedures have slots describing what component failed and which component actually detected that failure. These slots have been omitted from the figure due to spatial considerations; the interested reader is invited to examine the domain representation definitions included in Appendix C for complete details.

In order to illustrate how the representation of a detection procedure can be used to encode text, consider figure 4.17. The detection procedure for the first ring concentrator case consists of the following two sentences: "Indication of a RC failure is first detected by the next active node on the network. System management will reach a time out limit for receipt of the network token." As the figure shows, the first sentence is essentially a pointer to the class of detection procedure described by the text; in this case, it is a next-node detection procedure, in

Insert Figure 4.16

Insert Figure 4.17

which the node located downstream from a device in a network detects the failure of another device by timing out for the receipt of some message from that device. The second sentence references an actual step in the detection process: the timing out of the next node by failing to receive the token. Once an instance of the next node detection procedure has been instantiated, the rest of the representation for that procedure can be inferred, including the other steps in the detection process, the preconditions, the results, and so on (this is not shown in figure 4.17). The actual process of parsing text with procedural descriptions will be discussed in the section on Parsing.

4.2 Case Representation

Cases are the top-level organizing structures that package together entities, states, actions, events, and procedures into a representation of each failure case descriptions detailed within the FMEA manuals. A case description has slots corresponding to each of the main sections specified within a FMEA manual entry (see figure 2.1 for an example of such a text). Figure 4.18 shows the slot-filler hierarchy for a case description, including all of the slots that characterize a case (note that in practice there may be more than one of each type of slot in the case description). These include the failed component (an ORU entity), the failure causes and mode (procedures), the detection and correction procedures, and the failure effects (represented as a type of correction procedure). This case representation was determined by examining and characterizing the content of the FMEA manuals that the system is designed to parse.

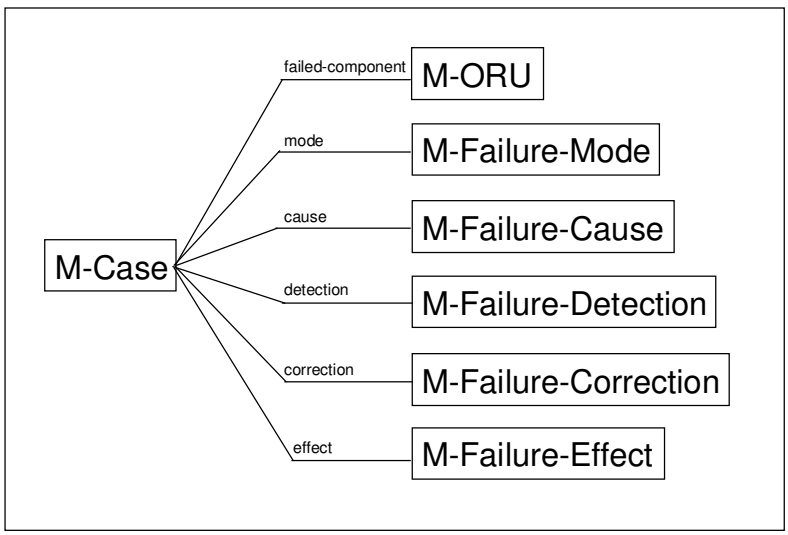

Figure 4.18
Representing a Case in FANSYS

Consider the text describing the first failure case for a ring concentrator given in figure 2.1. The next series of figures shows how each of the pieces of that manual text map into the memory structures that organize them. Figure 4.19 shows the mapping of the failed item name. The "item name:" tag is the stereotypical method in the FMEA manuals of specifying that what follows will be the name of the item that failed. When parsed, that tag provides a context which generates expectations to the system about what will occur next in the text. This is similarly the case for other tags like "failure mode:" and "failure effect on:". This process will be detailed in the section on Parsing.

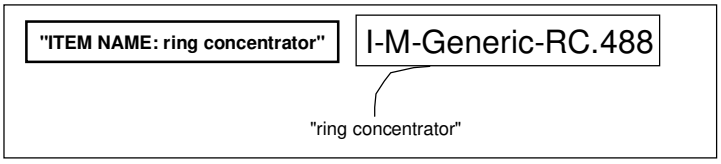

Figure 4.19
Failed Component Mapping for RC.1

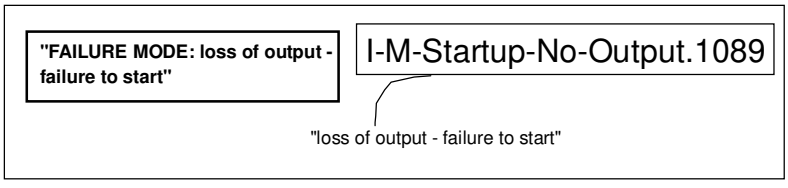

Figure 4.20
Failure Mode Mapping for RC.1

Figures 4.20 and 4.21 demonstrate the mappings of the failure modes and causes. These mappings are also fairly stereotypical in format; the text maps straight into the procedures which underlie it. However, the representation contains more information than the text for each of the modes and causes, and such an information is inferred when the individual MOPs are instantiated at parse time.

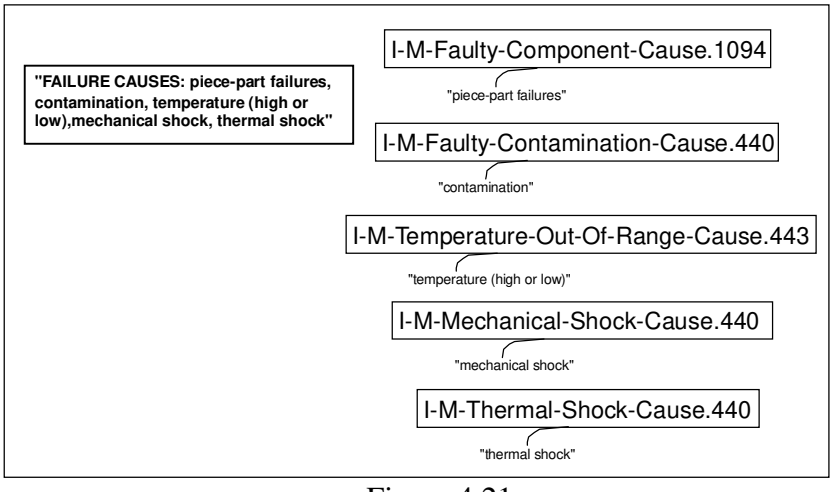

Figure 4.21
Failure Cause Mapping for RC.1

Figure 4.22 is a recapitulation of the figure used in the domain representation section on procedures, demonstrating the mapping of the detection procedure m-next-node-detection. As mentioned previously, the first sentence acts as a pointer to the type of detection procedure being described, while the second sentence provides information for instantiating the specific procedure.

Figure 4.23 is an example of the representation of the textual description of a procedure that has no steps explicitly specified; instead, the text just talks about the procedure in abstract terms, leaving the system to infer the actual steps. In this figure, the first sentence maps into a

Insert Figure 4.22

Insert Figure 4.23

very broad category of correction procedures involving the reconfiguration of a network. The second sentence specifies which procedure in that class is being referenced.

Figure 4.24 provides an example of a procedure wherein only the steps of the procedure are referenced, since both sentences describe individual steps of the m-non-operational-replacement correction procedure. As the last three figures suggest, the parser must be able to construct and recognize a procedure when any combination of pointers and individual steps occur within the text.

Figure 4.25 shows the mapping of the failure effects. In each of the manual entries examined, there was generally no effect on the mission components (this mapped to an instance of the null-procedure). In some cases, a justification is given that the correction procedure alleviated any possible effects (the first effect of figure 4.25, for example). In such cases, the text is mapped to the correction procedure which is used as that justification.

Figure 4.26 ties all of the preceding figures together into a single case representation. This is the complete characterization of the first ring concentrator case that is built by the system during the parsing of that case. The other cases have similarly been represented by hand-coding them using the same representational scheme (see Appendix A for details), although the parsing process should be largely the same as the example that is handled by FANSYS. The actual logistics of the parsing process are described in the next section.

Insert Figure 4.24

Insert Figure 4.25

Insert Figure 4.26

5. Comprehension of Input Text and Questions

In order to understand input segments of failure analysis manuals, FANSYS requires some means of building a conceptual representation of natural language text in the domain of failure detection, analysis, and repair. This task is performed by the parsing module of FANSYS, which uses case-based parsing techniques (Riesbeck and Martin, 1986; Riesbeck and Schank, 1989) to map input text into the underlying memory structures that characterize the conceptual content of that text. As described in System Architecture (Section 3), failure cases are stored as frames called MOPs within a multi-linked, hierarchical semantic network. This network encodes the DMS fault domain, as described in Domain Knowledge Representation (Section 4). The parser must create a mapping between a piece of text from the Failure Modes Effects Analysis (FMEA) manuals and those MOPs that represent the concepts underlying that text. Such MOPs may already exist within memory, in which case parsing is the recognition of those concepts and relationships in a text that the system has already encountered. Alternatively, new MOPs may be instantiated representing new instances of concepts that the system knows about but has not yet encountered. The parser must also index these MOPs in such a manner as to make them available during future text processing, and during question-answering sessions.

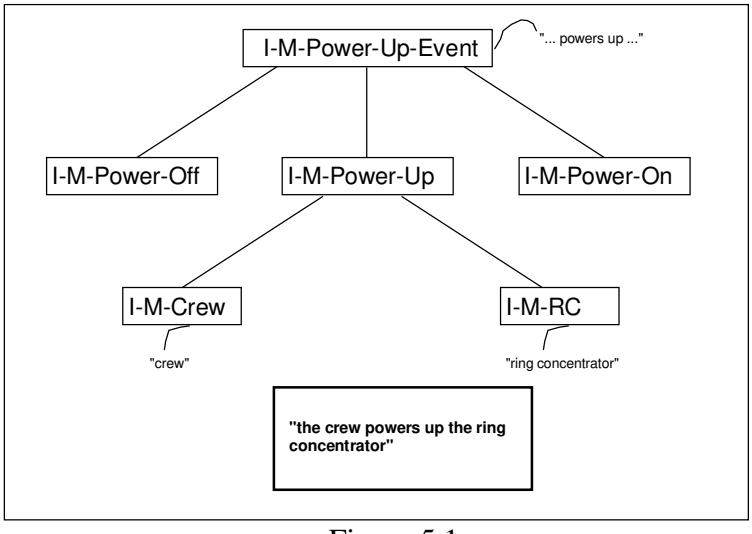

Figure 5.1
Example mapping of text to conceptual representation for the power-up event

To illustrate the process of mapping text into its conceptual representation, consider figure 5.1 which shows the representation of the following sentence fragment: "the crew powers up the ring concentrator." Conceptually, this text fragment describes an event wherein the crew members of the space station apply power to one of the components of the DMS. As described in the Domain Knowledge Representation Section, there are MOPs representing both powering events and systems within the DMS. The MOPs corresponding to those entities from the text fragment would be M-POWER-UP-EVENT, M-CREW, and M-RC. The task of the parser is to recognize each of these events and systems from the text, and to create (or find) the MOPs in memory which encode them. Figure 5.1 suggests the MOP representation that might be referenced by the system for the powering-up text fragment. Note that different pieces of the text map into different segments of the representation, and that there are parts of the representation that are not referenced explicitly in the text. For example, the state transition between the power

being off and the power being on (as represented by the precondition and result slots of I-M-POWER-UP-EVENT) is implicit in the text, and is inferred from domain knowledge.

One approach to creating this mapping between text and its underlying conceptual structures is to specify lexical patterns which capture the stereotypical use of language use. A lexical pattern can form a link to the MOP which characterizes the content of that pattern. For example, a pattern consisting of the words "ring concentrator" might map to the MOP M-RC, since that is how ring concentrators are specified in English. The parser also needs to know that "powering up" (when used in the context of an actor powering up a physical system) maps to the M-POWER-UP-EVENT, and that by virtue of the rules governing English use, the actor of the event will precede the words "powers up" and the object will follow. A general pattern that the system could use to capture this relationship might be "(actor) powers up (object)," where the actor is a concept representing an entity capable of performing a power up action, and the object is a device capable of being powered up. In this scheme, the phrase "ring concentrator" would map straight into M-RC and "the crew" would map into M-CREW. Note that this pattern is general enough to also be used in the following case: "the system management powers up the gateway," with "the system management" mapping into M-SM and "the gateway" mapping into M-GW. The generality of the phrase "(actor) powers up (object)" captures a broad class of mapping instances, which is a reflection of stereotypical language use.

Another point to be made about the parser is its language independence: the conceptual representation mapped into by the parser should be independent of the surface form of the source language. For example, the same MOP built to characterize the sentence "the crew powers on the ring concentrator" should also be created for the sentence "the power system of the RC is activated by the crew members." Processing this sentence entails using a different lexical pattern to capture the notion of an event where an object "is activated by" someone, but the underlying conceptual structures would be the same.

In addition to identifying the memory structures represented within the input text, the parser must perform a number of other tasks to cohesively organize those structures. For example, causal relationships between statements within the text must be recognized (e.g. through application of inferential domain knowledge) and characterized. The salient components of a case description, such as the failure causes, effects, and correction procedures must be recognized and organized for a case instance. Procedures that are composed of multiple steps across several sentences within a text must be constructed and organized. In each of these tasks, domain knowledge, in the form of expectations as to what comprises a procedure, case, or abstract relationship must be applied to the input text to further elucidate those relationships within the text that are often left implicit.

Once the case has been constructed by the parser, it must be stored within memory in a manner which insures that it is available during future parsings of text. This is facilitated by indexing case descriptions with the ISA and slot-filler links described in System Architecture (Section 3). As a case library is built up by the reading of successive texts, generalizations across cases and across the components of a case (procedures, entities, events, etc.) should be drawn. The cases are indexed with the generalization indexing hierarchical scheme to facilitate the generation of such generalizations. This process of drawing these generalizations from text will

be further described in the Memory Search and Retrieval (Section 6). The case library, built incrementally during the processing of manuals, should then be available for question answering tasks. These tasks rely on both the parsing process (to understand natural language queries) and the search and retrieval process (to find answers to the queries).

Since the parser interacts with the Search and Retrieval Module during question-answering sessions, it must work in a tightly coupled manner with memory search and retrieval processes. To this end, the MOPs that are created and recognized by the parser are the same ones used during memory search and retrieval; only the indexing hierarchies and the MOPs created as generalizations are different across the two modules. The natural language question answering shell described in User Interfaces (Section 7) is built directly on top of the Parsing Module. Questions about cases are parsed with the same parser that is used to process cases; functional knowledge associated with the memory structures activated by parsing a question are responsible for generating the query used by the memory search and retrieval processes. This tightly-knit integration of the two modules provides a flexible natural language query environment.

Parsing in the restricted domain of fault detection and correction for the DMS has some ramifications for the parsing process. As figure 2.1 suggests, the text for the FMEA manuals is highly stereotypical in format. Case descriptions are always given with sections for each of the major components of a case, including the causes, effects, correction procedures, detection procedures, and effects. The parser may take advantage of the standard format for a manual to generate expectations about what it will see next in a text. For example, the text following the tag phrase "Correction Procedure:" is likely to be a procedural description of how to correct a failure in a DMS component. These expectations generated after recognizing a word or phrase allow the parser to apply domain knowledge to help process the text at appropriate times.

Another feature of the restricted domain over which the texts deal is the manner in which the task of disambiguation is made easier. The meaning of a word in any natural language can be highly context dependent; words have multiple meanings, and the appropriate meaning is often determined by context. In a restricted domain like that of the DMS, word meanings are highly constrained. For example, the word "network" is likely to deal with a configuration of communicating devices, as in a token ring network, rather than, say, a television network. Likewise the word "ring" might appear as part of the name of a ring concentrator, or as part of a token ring network, but would not be used in the sense of a wedding band. However, domain restrictions do not to imply that all words are easily disambiguated within the domain. For instance, the word "failure" is used within many different contexts; disambiguation typically occurs by waiting to see what the subsequent word is, such as in "failure correction" versus "failure mode."

The parser currently parses the case described previously for the failure of the ring concentrator (see figure 2.1). The actual representation for that case was shown in the Domain Knowledge Representation Section (figure 4.26). Both the theoretical and implemented processes used to parse this case will be discussed in the rest of this section, including a detailed, annotated trace of the parser in operation. The processes involved in question-answering and text generation will also be considered. Although only the one case is currently parsed by the system, the same processes developed for the first case are applicable to any of the other cases the parser

may encounter. For example, the first gateway case, GW.1 (loss of output - failure to start) involves the same modes, causes, and effects of the ring concentrator case shown in figure 2.1. The same processes that construct the correction and detection procedures in the ring concentrator will apply to the gateway; only the individual steps that describe the procedure and the type of procedure are different. The methodology used by the parser and other components of the system are designed to generalize across the whole corpus of FMEA manuals, creating a system capable of understanding and operating within the DMS fault domain.

5.1 Case-Based Parsing

In FANSYS, comprehension of input text and questions is performed using the case-based parsing techniques provided by DMAP, a Direct Memory Access Parser (Riesbeck and Martin, 1986; Riesbeck and Schank, 1989). In DMAP, parsing is viewed as a recognition process, i.e., the goal of the parser is to determine which memory structures best organize the input based upon what the parser has already been exposed to. In reading the case describing a gateway failure, the parser will automatically find many of the other case representations it has already seen and use them to help understand the new input. The output of such a parser is the set of memory structures that have been referenced in the understanding of a new text (such as part of some other case), the new structures added to memory during the parse, and the set of expectations about what will be seen next based on what was just read. In this view, parsing is tied directly to the contents of memory; there is no notion of "outputting" some representation. Exactly what occurs when something is referenced in memory is itself guided by the goals of the understanding system, which are also represented in memory. In the case of FANSYS, the goal is to organize the input in ways useful to characterizing the DMS fault domain, and to permit natural language queries over the case library.

Since case-based parsing is seen as a recognition process, memory organization and search are central to any case-based paradigm. MOP structures (Schank, 1982; Riesbeck and Schank, 1989) are used in FANSYS to represent linguistic, domain, and world knowledge. A MOP may be used to represent a word, a concept, or any other semantically useful item within memory. The Domain Knowledge Representation section already detailed several classes of MOPs used to represent domain knowledge within FANSYS. In addition to those MOPs, there are MOPs for each word that the system knows about, as well as MOPs called *index patterns* (Riesbeck and Martin, 1986; Riesbeck and Schank, 1989) which represent stereotypical mappings of natural language into their corresponding concepts. These index patterns drive the parsing process. Other MOPs represent processing knowledge the system has to aid in constructing case descriptions. MOPs are used to implement all knowledge constructs at every level of abstraction and functionality within FANSYS.

MOPs represent nodes in a semantic network, and are organized hierarchically along two parameters that are crucial to the parsing process: the slot-filler hierarchy and the ISA hierarchy. The slot-filler hierarchy links together a MOP with its conceptual subcomponents; for example, the MOP representing an action would organize at least two MOPs in the slot-filler hierarchy: one for the MOP representing the actor, and one for the MOP representing the object being operated upon. MOPs organized within the slot-filler hierarchy thus implement a frame-based representation of knowledge. The ISA hierarchy organizes MOPs based on their conceptual

abstractions, from most general concept to most specific. Thus, the MOP for the power-up action would have as an abstraction the MOP representing power actions in general, which in turn would have an abstraction representing generic actions, and so on. Other specific action MOPs, such as transmission and connection actions would also have the generic action MOP as their abstraction.

MOP-specific processing strategies can also be associated with any given MOP in memory. The processing strategies (called associated functions) are executed whenever a MOP is referenced during parsing or during other processing. Similarly, contextual constraints may be associated with a MOP, which inhibit its activation during a parse if contextual requirements are not met. Thus, domain specific processing strategies, knowledge specific functionality, and contextual requirements are represented and stored with the MOP to which they pertain, and are invoked only when needed. This approach gives the system flexibility in specifying the precise processing characteristics required for different knowledge constructs, and insures that processing proceeds in an integrated manner.

The actual parsing of a text involves the recognition of slot-filler relationships between concepts in a given text. To specify these relationships, FANSYS employs index patterns, which represent stereotypical uses of a target natural language. For example, the pattern { (actor) powers up (object) } => M-POWER-UP-EVENT specifies that a power-up event can be recognized if an actor concept followed immediately by the words "powers up" followed by some object concept is recognized from the input text. The actor and object concepts are slots in the power-up event representation; the index pattern represents the linguistic or conceptual relationship between those slots as expressed in English. Patterns are associated with the MOP that they reference and the MOP with which they are stored by lexical links.

These index patterns also specify lexical and (implicitly) semantic constraints. In the example of the powering-up event, the words describing the actor mop are required to be adjacent to the words "powers up" in order for the pattern to be recognized. Patterns thus implicitly encode syntactic features of a target language. Concepts specified within an index pattern are further constrained by filler restrictions specified within the MOP to which the slot pertains. In the powering-up example, the object in the pattern is constrained to be some mechanical device (e.g. a system) capable of being powered up.

Index patterns are useful for capturing lexical relationships between concepts within text. There are, however, more loosely constrained conceptual relationships which may occur in a text that are not lexically based. In a case description within a FMEA manual, for example, the system needs to know that there are certain components to a case that it should expect within the text: the failure mode, the correction procedures, the effects, etc. These may occur in any order in general (although as mentioned earlier, the FMEA manuals are quite stereotypical in the ordering of the case components -- this may not be the case in other manuals). High level index patterns (HLIPS) exist that capture these kinds of relationships. One example of this type of pattern is { failed-component mode cause detection correction effect } => M-CASE. This pattern indicates that the system should collect any instances of failed components, modes, causes, etc. together that it sees in a text, and use them as slot fillers for a case MOP. These patterns describe the high level organization of a target text. One characteristic of these patterns that contrasts to lexical

index patterns is that all components of the pattern need not be present in the text; the HLIPs just encode predictions about what may or may not be seen in the text.

In addition to lexical patterns and HLIPs, other types of processing strategies are needed to apply domain knowledge in building a coherent representation of a case. These types of strategies implement a form of inference generation. Inferences need to be drawn in many cases because much is often left implicit in a text that needs to be made explicit in order for the system to tie together relationships and concepts within a text. Consider for example the sentence: "Indication of a RC failure is first detected by the next active node on the network." The phrase "next active node on the network" is enough to recognize the relationship of two functioning nodes on the network, one of which is in some sense the next node to the other (in the sense of the ordered passing of a token in a token ring network, for example). The fact that the next node is itself a ring concentrator, and that it is waiting for a token from the failed node, are all left implicit, and must be inferred from the understander's domain knowledge about token ring networks.

Another type of inferential strategy is that which pulls together steps from a procedures, determines which procedure is being referenced, and then recognizes that procedure. Consider the following correction procedure from the first case description: "The crewmen check for applied power and check for connector tightness. If the RC cannot then be placed in operation, it is removed and replaced with an ORU logistics spare." The fragment details essentially three steps to a procedure: checking for power, checking the connectors, and replacing a unit with a spare. These three steps form a procedure for correcting a failure. The parser must be able to recognize that these events are steps in a procedure, and then to recognize the procedure in memory that is being referenced. In this case, it is the procedure called M-NON-OPERATIONAL-REPLACEMENT, described earlier in figure 4.24.

These inferential strategies are generally attached to specific nodes in memory that get referenced during a parse. For example, the procedure building strategy is attached to the MOP for an abstract event, so that any event that is referenced during a parse will execute it. Inferences are thus localized to the MOPs to which they apply, and are activated naturally in the course of the recognition of those MOPs. The mechanisms for implementing these strategies, and for the parsing of text in general, are described next. An annotated trace will also exemplify these processes.

5.2 Parsing Input Text

As mentioned in the previous section, the stereotypical use of natural language is captured through the use of lexical patterns (called index patterns) which map text into the memory structures underlying the conceptual content of that text. Index patterns are themselves MOPs, indexed via lexical activation links with the MOP that is most predictive of the pattern, and with the MOP to which the pattern refers via lexical reference links. In the example of IP1: { (actor M-SYSTEM) powers up (object M-ENTITY) } => M-POWER-UP-EVENT, the pattern would be stored with the lexical MOP for the word "powers," since that is the most relevant element in the pattern to its recognition. The pattern should not be stored with the M-SYSTEM MOP (which is the filler for the actor slot), for example, because actors appear in many different

patterns and contexts. IP1 would also have a link to the MOP M-POWER-UP-EVENT, since that is the MOP characterized by the pattern. If a MOP is referenced in the text or through subsequent processing, any index patterns that are associated with that node are activated. For example, anytime the MOP for the word "powers" is referenced in the text, pattern IP1 becomes activated. If the elements that appear prior to that MOP in the pattern have occurred in the text in the proper lexical order (i.e. lexical adjacency requirements are met), then the pattern is considered active and will be tracked. In the example if IP1, the M-SYSTEM MOP would have had to have been referenced just prior to the MOP for the word "powers" in the previous pattern. Once a pattern is tracked, a prediction is generated about what will appear next in the text based on that pattern. To continue the example, the parser would predict that the MOP for the word "up" will appear next in the text. If this prediction fails, the pattern is abandoned. If the prediction is confirmed, then the pattern will be further advanced, and the next element of the phrase predicted (in the example of IP1, it would be the node representing the object slot filler, M-ENTITY). When all of the elements of a pattern have been recognized in the text, the target of the pattern is recognized with its slots filled based on the text. In the example "the crew powers up the ring concentrator," an instance of the M-POWER-UP-EVENT MOP would be instantiated with the slots of (actor I-M-CREW) and (object I-M-GENERIC-RC).

The application of index patterns involves a mixture of bottom-up and top-down processing. Bottom-up processing occurs in the recognition of lexical units (words, usually) from the input text. These words may trigger the activation of index patterns, which then spawn predictions about what will be seen next in the text. Subsequent words or concepts may fulfill top-down expectations as generated by index patterns, as in the prediction of the word "up" or the concept of M-ENTITY in IP1. Patterns which are fully recognized lead to the activation of new MOPs, which may themselves have index patterns associated with them. The output of the parser after reading a text is thus the memory structures that have been created or recognized to organize the input, and the new predictions that are pending based on that reading.

Index patterns are defined as in figure 5.2. The parts of the definition in square brackets are optional. The syntax-category attribute is a simple method of encoding syntactic constraints to be used during generation. (See the section on Generation for more details on the use of syntactic categories; they do not effect parsing.) The target is the MOP to be activated if all of the elements of a phrase are recognized; the target thus specifies the lexical activation link. Elements (elem1 ... elemn) may be either words (lexical nodes) or concepts in memory. The :terminator specifies when the pattern should be destroyed if it hasn't been recognized. This is typically set to be M-HARD-PUNCTUATION (e.g. when a period is parsed), and is used in the cleaning up of abandoned patterns. The :store node specifies which of the elements of the pattern should be used as the lexical activation link; that is, which element is most predictive of the pattern. The :gen and :nogen fields are used during generation; see the Generation subsection for more details, as they do not influence parsing directly. Finally, the :opt allows the system to specify optional elements of a pattern. These elements need not appear in the text in order for the whole pattern to be recognized. If they are in the text, they are parsed as part of the pattern, but they are ignored otherwise. The definition specifiers given in figure 5.2 allow index patterns to be defined with a fair degree of generality. See appendix C for a listing of the pattern definitions used by the system.

```
(defphrase [syntax-category] target elem1 [elem2 ... elem n]
  [:termination termmop] [:start elem-number] [:gen] [:nogen] [:opt optlist])
```

Figure 5.2
Index Pattern Structure

In addition to index patterns, there is also a class of closely related structures called higher-level index patterns (HLIPS). These are similar to normal index patterns, except that they do not require that any adjacency requirements be met. HLIPS typically capture relationships that occur across large sections of a text that are not order dependent. For example, case descriptions have a number of features including the correction and detection procedures, the failure causes and modes, the failure effects, and so on, that could in principle appear in any order within a manual description. There is thus a HLIP defined that looks for instances of each of those features and collects them together to create a case MOP. Appendix C also details those patterns as used to parse the first ring concentrator case.

5.2.1 Marker Passing

Mention has been made of the generation and fulfillment of predictions from index patterns, as well as the recognition (or activation) of structures within memory. These features have been implemented in the FANSYS parser using a marker passing algorithm suggested (but not implemented) in microversion of DMAP appearing in (Riesbeck and Schank, 1989). Marker passing may be defined as the propagation of structured objects over some representational network. The intersection of markers during this propagation typically allows useful work to be performed. (See (Hendler, 1988) and (Norvig, 1989) for an introduction to marker passing methods in general.)

In FANSYS, two types of markers are propagated: prediction markers and activation markers. Prediction markers are passed to nodes in memory which are predicted by advancing index patterns. These markers contain backpointers to the index pattern (or other MOP processing structure) which generated it. Activation markers are passed to MOPs and their abstractions which have been referenced in input text, or have been activated by other means (e.g. by activating the target node of a recognized phrase). The collision of these two types of markers at a MOP in memory represents a fulfilled expectation. The node that generated the prediction is subsequently informed of fact that the prediction was satisfied. Activation markers thus keep track of the state of activation within memory; that is, which MOPs have been recognized during the parsing of text. Prediction markers keep track of the state of expectations within the system. Together, these markers implement an efficient scheme for generating and fulfilling predictions within FANSYS.

Figure 5.3 shows the structure of an activation marker. The parent of an activation marker is a backpointer to the node which was initially activated (since markers may be passed around the ISA hierarchy, a backpointer to the original node is necessary). The owners attribute is a list of all of the MOPs which have been passed the activation marker; this is just a device to aid in the eventual removal of the markers from the system. The start and finish attributes are used by index patterns to enforce adjacency requirements. When an index pattern is first

instantiated, it is given the "start" value of a counter which is incremented by one every time a word is read in the text. When the pattern is fulfilled, the value of "finish" is set to the current value of the counter. When a node is referenced by an index pattern, it in turn inherits the start and finish values of the pattern for its activation marker. These values are then used within other index patterns to enforce adjacency of concepts and words in the pattern. Adjacency may be determined by noting that two nodes are adjacent if and only if the finish attribute on the marker of the first node is one less than the start attribute of the marker on the second node.

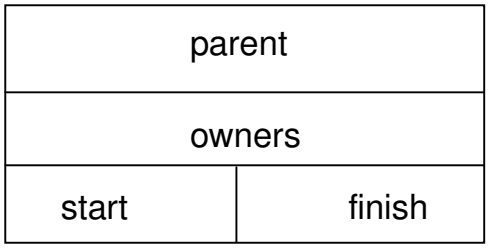

Figure 5.3
Activation Marker Structure

The structure of a prediction marker is demonstrated by figure 5.4. The only information stored with a prediction marker is a backpointer to the MOP which spawned the prediction. This pointer is used to inform the predicting node of the fact that the prediction was fulfilled.

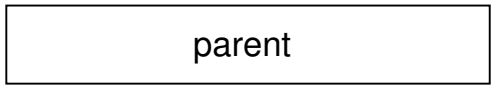

Figure 5.4
Prediction Marker Structure

The basic marker passing algorithm which implements the prediction/activation mechanism is an adaptation of the one presented in (Riesbeck and Schank, 1989). It consists of the following four rules for passing markers:

Activation Markers

- When a word is read from the input text, an activation marker is given to the corresponding lexical node.
- When every element of a phrase receives an activation marker, the target concept attached to the phrase receives a marker.
- When a node receives a marker, it sends copies to all of its abstractions up the ISA links.

Prediction Markers

- When an element of a phrase in an index pattern receives an activation marker, the next element in the phrase gets a prediction marker.

The collision of an activation and a prediction marker causes the parent node of the prediction to be activated. Associated functions for that MOP may then guide processing to deal with the fulfilled expectation. Fore example, an index pattern node that is activated by a fulfilled prediction has functional knowledge that would advance the pattern one element, filling in any appropriate slots, or that would activate the target node if all elements had been recognized.

One final note concerning the marker passing algorithm is worth making. This marker passing scheme is independent of the configuration of memory, and thus is independent of the domain in which it operates and source language with which it is concerned. It is a general mechanism for implementing activation, expectations, and fulfilled predictions. Note on the other hand that the actual comprehension of text is dependent on the memory configuration, since different memory structures and domain knowledge will cause a different pattern of activation throughout memory.

5.2.2 The Parsing Algorithm

The actual process of parsing text into the internal representation that characterizes it is founded upon a simple algorithm:

1. Get the next word from the text.
2. a) If a lexical node exists in memory corresponding to that word, activate that node.
   b) Otherwise, create a lexical node for that word and index that MOP under the MOP M-UNKNOWN-WORD.

Each word that FANSYS knows has a corresponding MOP in memory. Words are parsed from left to right, with each MOP corresponding to a word being activated when that word is read (the concept of activation will be discussed in a moment). Words that are not recognized are added to memory as lexical nodes, but are indexed as unknown words. This is allows the administrator of the system to notice new words and add any corresponding processing knowledge to the system to handle that word. (In future implementations of the system, it may be possible to have the system implement learning strategies for unknown words automatically, based on the context in which the word occurs; such strategies could be stored as functional knowledge with the MOP M-UNKNOWN-WORD.)

Activation is the fundamental memory operation for the parser. A node is activated if it has some relevance to the processing of the input text. (The notions of the recognition of a MOP and the activation of a MOP are used interchangeably within this report.) The activation operation consists of the following six steps:

1. The mop is specialized to an appropriate instance, if there are any specific slots passed to the activation function. This may involve finding a prior instance in memory that matches the MOP and its slots, or it may involve creating a new instance of the MOP in memory.
2. An activation marker is created for the node. If the node is activated with an activation marker passed to it, it gets a copy of that marker. Otherwise, a new marker is created for this node.
3. Any functions associated with that node are executed. These functions typically encode the structure-specific (semantic) content of that node (i.e. how to process it).
4. All of the index patterns which are stored with this MOP via lexical activation links are activated.
5. If this node was predicted (because prediction markers were passed to it previously), then the predictions are fulfilled by activating the MOP or MOPs that predicted it.
6. All of the abstractions of this node are activated, and are passed this node's activation marker.

Whenever a concept in memory is activated, it is specialized to the most specific instance of that MOP that matches the input. Typically, an abstract MOP is activated with a set of slots and fillers specified. Consider the example presented earlier, where the M-POWER-UP-EVENT MOP is activated with the slots of (actor I-M-CREW) and (object I-M-GENERIC-RC). If the parser has already seen an instance of M-POWER-EVENT with the specified slots, it would find that instance and activate it. Otherwise, the parser is seeing novel text, and must create a new instance of the MOP with the appropriate slots. (This new instance may cause some reorganization of memory, since the Memory Creation and Retrieval module may use it to draw generalizations about the text; see the section on Memory Creation and Retrieval for more information.) This process of specialization demonstrates how the parser uses a case-based approach: new input text is mapped onto already existing (i.e. previously experienced) memory structures, if such structures are available. Memory content (experience) guides parsing. Since new MOPs are built during parsing if no suitable MOPs exist in memory, parsing changes memory. If a text is read twice, most of the processing (the creation of all new MOPs) occurs during the first reading; the second time around, the text is mapped onto the same structures used or created during the first reading. Thus, FANSYS learns from reading a text, making subsequent processing of similar material more efficient.

When a MOP is activated, any functions associated with that MOP are executed. This allows MOP-specific processing strategies to be executed only as needed. For example, the code that handles the processing of index patterns is stored with the abstract node M-INDEX-PATTERN. Whenever an index pattern MOP is activated, this code gets executed, and the pattern is processed. Thus, the parsing process involves domain- and MOP-specific processing in addition to the domain-independent processes described in the previous algorithms. FANSYS' parser can therefore be viewed as a distributed parser, since domain and knowledge specific parsing strategies are stored throughout memory as needed, and come into play only when such associated MOPs are activated. This also provides an adaptable and extendible environment,

since new domains can be added to the model by defining the MOPs required for that domain, and by specifying any special processing requirements with those MOPs.

Another event which occurs when a MOP is activated is the activation of all of that MOP's abstractions. This captures the notion that when one is talking about M-POWER-UP-EVENT, one is also talking about powering events in general (e.g. M-POWER-EVENT), as well as generic events (M-EVENT). Since index patterns can define textual relationships at varying levels of abstraction, this mechanism allows the parsing of input text to also occur on multiple levels of generality simultaneously. For example, the M-POWER-UP event may have index patterns associated with it, as may the M-EVENT MOP; these patterns would vary in terms of their generality, but each would become activated when a power down event is processed in the text.

Since the activation of a MOP usually leads to the subsequent activation of several other MOPs, including abstractions, index patterns, predicting nodes, and so on, the parsing of a word typically involves the spreading of activation over several parts of the memory of FANSYS. This spreading activation is constrained by the algorithm, but allows processing to flow to those structures which are required to process the text. This provides an efficient and domain-independent method for understanding text, but one which is still guided by the actual content of the domain representation.

5.2.3 Domain-Specific Inference Strategies

In addition to the normal flow of control detailed by the parsing algorithm and the tracking of index patterns, some domain specific inferencing strategies are implemented which also effect the processing of an input text. These inferencing strategies are generally attached to specific processing MOPs as associated functions. Such processing MOPs may become activated by higher level patterns or through explicit activation by other associated functions. These strategies represent domain-specific parsing knowledge, and are inferential in that they typically involve processing that makes relationships and concepts that are implicit within a text explicit.

One example of such an inferencing strategy is the functionality which constructs and recognizes procedures within textual descriptions. Procedures occur within the text in two primary contexts, that of the detection procedure and the correction procedure. Procedures are usually specified in one of two ways (as has been determined by a study the input manuals). In the first case, a pointer to the class of procedure is given as the first sentence of the textual description, followed by one or more events which serve to disambiguate which instance of that class is being recognized. For example, consider the failure detection procedure of the first ring concentrator case, RC.1 (see figure 2.1). The first sentence of this description, "Indication of a RC failure is first detected by the next active node on the network," is a pointer to the class of detection procedures in which a failure is detected by another node noticing that some expected communication with it has failed to occur. The next sentence, "Local system management will reach a time-out limit for receipt of the network token," is an event which disambiguates the detection procedure. In this case, since the event details a token circulating around a token ring which fails to be passed to a node, the parser may determine that the procedure in question is the M-NEXT-NODE-DETECTION procedure.

The second class of procedural recognition strategies occurs when a text elaborates two or more steps of a procedure. In this case, the parser must recognize each step, pull them together, and infer which procedure in memory is being referenced by the text. As an example of this class, consider the long-term corrective procedure of RC.1. In this example, three steps are mentioned in the text: checking for applied power, checking for connector tightness, and replacing the spare. These are three steps of the procedure M-NON-OPERATIONAL-REPLACEMENT, which must be recognized and activated during the parse of RC.1.

Each of these two cases requires a specific strategy implementation to determine the procedure being referenced (and thus to ultimately activate that procedure in memory), although the fundamental functionality is the same in both cases. In the first case, a pointer is given to the class of procedures involved, which significantly reduces the space of procedures which much be searched for the target procedure. In the second case, the library of all procedures (correction or detection, depending on context) must be searched. Once the search space is determined by the strategy type, the next step is to collect together the subsequent events in the text, and determine which procedure they comprise. It should be noted that usually not all of the events in a procedure are specified in a text; many are left implicit, with the assumption that the reader's domain knowledge will be sufficient to fill in the missing details. Since all of the events must be collected together, a processing MOP is activated by the first event to occur in the text which will keep track of all of the events that are subsequently referenced (this activation occurs by virtue of an associated function attached to M-EVENT). This MOP subsequently predicts that it will see more events by passing a prediction marker to M-EVENT each time it is activated. If that prediction is fulfilled, the processing MOP will again be activated, and it can deal with the new event, and predict that it will see yet more events. Once the procedural context is left by parsing a new section of the text, the processing MOP considers all of the events it collected and determines which procedure was referenced.

The algorithm for determining which procedure is to be inferred from the collected events is a simple one. Each event step contains a back pointer to each procedure in which it occurs. (In fact, all MOPs contain a backpointer to any other MOPs in which it occurs as a slot filler.) Since an event may occur in more than one procedure, there may be several such backpointers. Once all of the steps are collected together, the processing MOP intersects all of the backpointers together from all of the events; this intersection is likely to be just one MOP, the procedure which packages all of the steps. If more than one MOP is in the intersection set, than some other method may be required to further disambiguate the procedure. As our corpus of procedures is limited by the number of cases that are dealt with, this has not proven necessary -- the process of intersecting the backpointers has proven sufficient to uniquely determine all procedures tested. It is recognized however that with a more extensive library of procedure descriptions, a more generalized approach to procedural referencing may be necessary. Once the procedure has been identified by whatever means, it is activated and the processing MOP created for the recognition task is terminated.

There are several other miscellaneous inferencing strategies that are employed to process the text for RC.1 (see figure 2.1). One example of this is the disambiguation of the word "none" in the failure effects procedures. There are two practical meanings of the word "none" in this context. The first is exemplified by the failure effects on the crew in RC.1. Although the effect is

listed as none, there actually is one effect on the crew; they must implement the correction procedure described by the text after the word "none". This text is almost always a pointer back to one of the correction procedures described earlier in the case. The task of the parser here is to infer that the real effect is the correction procedure described next in the text. The inferencing function (associated with the word "none" when it appears in the context of the effects section of the text) must therefore search back over the correction procedures to find the one which best matches the text in the given effects text. In the case of RC.1, the phrase "the conditions associated with the replacement of the RC" is enough to disambiguate between the two correction procedures, since the procedure M-NON-OPERATIONAL-REPLACEMENT is the one that involves an event with the crew replacing a spare RC. For the other effects, where there is no subsequent text after the word "none", the parser must infer that the default procedure M-NULL-PROCEDURE is the actual effect (i.e. that there is indeed no effect). This is the second practical meaning of "none" in this context.

Some other inferencing procedures implement contextual constraints and miscellaneous disambiguation functions. These will not be described here in any detail. The interested reader is invited to examine the code for more details. It should be noted that although most of the inferencing strategies are founded upon a generalized theory of what should constitute a reasonable inference, it is not claimed that all of the inferencing strategies implemented to parse the first case are of a general nature. Some strategies will undoubtedly generalize as more cases are processed by the system.

5.2.4 An Annotated Example

To illustrate the processes that have been discussed in the prior subsections, consider the actual trace output of the parser as it parses the first ring concentrator case, RC.1. The trace is highly edited due to spatial considerations; only features which illustrate relevant processing will be left intact. Interested readers should consult Appendix B for a complete copy of the trace. Note also that details will be omitted once they have been covered in the trace; this will have the effect of creating a sparser trace as this subsection progresses.

```
>(parse rc.1)

Parsing (ITEM NAME *COLON* RING CONCENTRATOR FAILURE MODE *COLON* LOSS
        OF OUTPUT - FAILURE TO START FAILURE CAUSES *COLON*
        PIECE-PART FAILURES *COMMA* CONTAMINATION *COMMA*
        TEMPERATURE *LEFT-PAREN* HIGH OR LOW *RIGHT-PAREN*
        *COMMA* MECHANICAL SHOCK *COMMA* THERMAL SHOCK FAILURE
        DETECTION *SLASH* VERIFICATION *COLON* INDICATION OF A RC
        FAILURE IS FIRST DETECTED BY THE *QUOTE* NEXT *QUOTE*
        ACTIVE NODE ON THE NETWORK *PERIOD* SYSTEM MANAGEMENT
        WILL REACH A TIME-OUT LIMIT FOR RECEIPT OF THE NETWORK
        TOKEN *PERIOD* CORRECTIVE ACTION *COLON* *LEFT-PAREN* A
        *RIGHT-PAREN* SHORT TERM *COLON* NETWORK RECONFIGURATION
        IS EFFECTED AUTOMATICALLY *PERIOD* THE DMS NETWORK
        REMAINS IN OPERATION IN A RECONFIGURED STATE WITH THE
        FAILED RC BYPASSED *PERIOD* *LEFT-PAREN* B *RIGHT-PAREN*
        LONG TERM *COLON* THE CREWMEN CHECK FOR *QUOTE* APPLIED
        POWER *QUOTE* AND THE CREWMEN CHECK FOR *QUOTE* CONNECTOR
```

```
TIGHTNESS *QUOTE* *PERIOD* IF THE RC CANNOT THEN BE
PLACED IN OPERATION *COMMA* THE RC IS REMOVED AND
REPLACED WITH AN ORU LOGISTICS SPARE *PERIOD* FAILURE
EFFECT ON *COLON* *LEFT-PAREN* A *RIGHT-PAREN* CREW
*SLASH* SSPE *COLON* NONE *PERIOD* THE CONDITIONS
ASSOCIATED WITH THE REPLACEMENT OF A RC WITHIN THE
NETWORK CONFIGURATION ARE SUCH THAT THE NETWORK IS
ALREADY IN A RECONFIGURED STATE SUPPLYING FULL SERVICES
*PERIOD* *LEFT-PAREN* B *RIGHT-PAREN* MISSION SUPPORT
*COLON* NONE *PERIOD* *LEFT-PAREN* C *RIGHT-PAREN* SYSTEM
*COLON* NONE *PERIOD* *LEFT-PAREN* D *RIGHT-PAREN*
INTERFACES *COLON* NONE *PERIOD* *EOC*).
```

The parser begins by processing the text in a word-for-word manner. The first word, "item", is activated as described in the activation section. This includes activating the word, its abstractions, any index patterns associated with it, and so on.

```
Reading ITEM
    Recognizing: M-IP.388
    Creating: I-M-IP.388.417
    Specializing: I-M-IP.388.417
    Recognizing: I-M-TERMINATOR.323
  (ITEM * NAME *COLON*) = M-ITEM-CONTEXT
```

In activating the word "item", one index pattern has been found which is stored with that word. This pattern is activated. As usual in the activation process, the concept is recognized, and an instance is created for it. FANSYS then determines whether it has seen this input before by trying to specialize the input into an existing memory structure. If it is successful, the parser will throw away the new instance and activate the prior occurrence. Otherwise, it will store this new instance in memory. In this case, the index pattern is a new one, and so is added to memory. The I-M-TERMINATOR.323 MOP is a bookkeeping MOP which controls when this pattern will be killed. The designer of the lexical phrases may specify when a given phrase should be terminated; the default is at the end of a sentence. The MOP M-ITEM-CONTEXT is a MOP which specifies that the current context is that of the failure item. Within this context, any devices that are specified are interpreted as the failed component(s) of the case. In order for this context to be activated, the full tag of "item name:" must be encountered in the text. The model is thus relying on the stereotypical format of the case manuals to determine the context in which it is processing. Notice that if the phrase "item name:" is encountered anywhere within the text, the parser will activate this context. This has the potential of confusing the system, but the manuals are generally structured such that this is not a problem.

```
Reading NAME
  (ITEM NAME * *COLON*) = M-ITEM-CONTEXT
```

Here the pattern continues to be tracked by the system.

```
Reading *COLON*
  (ITEM NAME *COLON* *) = M-ITEM-CONTEXT referenced
    Recognizing: M-ITEM-CONTEXT
    Creating: I-M-ITEM-CONTEXT.418
    Specializing: I-M-ITEM-CONTEXT.418
```

```
    Recognizing: M-CONTEXT
    Creating: I-M-CONTEXT.420
    Specializing: I-M-CONTEXT.420
    Removing: I-M-IP.388.417
```

Since the whole phrase has been recognized, the target of the phrase, M-ITEM-CONTEXT is activated. The functional knowledge associated with this MOP activates the general M-CONTEXT MOP, which is tasked with the responsibility of maintaining the current context. The M-CONTEXT MOP in turn makes note of the new context. This context inhibits the activation of some higher level patterns and allows the activation of others. Specifically, it will allow the activation of those high level patterns which keep track of the failed-component slot for a case, and inhibit the activation of patterns that are used in the other contexts. The last bit of processing involves removing the fulfilled index pattern from the system.

```
Reading RING
  (RING * CONCENTRATOR) = M-GENERIC-RC
```

For index patterns, the recognize/create/specialize/termination sequence will be edited out from here on and only the pattern will be shown as it is tracked.

```
Reading CONCENTRATOR
  (RING CONCENTRATOR *) = M-GENERIC-RC referenced
    Recognizing: M-GENERIC-RC
    Creating: I-M-GENERIC-RC.422
    Specializing: I-M-GENERIC-RC.422
    Recognizing: M-HL-PATTERN.387
    Creating: I-M-HL-PATTERN.387.423
    Specializing: I-M-HL-PATTERN.387.423
    Removing: I-M-IP.327.421
```

With this word, the pattern recognizing a generic instance of a ring concentrator is fulfilled, and that instance is activated. This causes a high level pattern to be activated; this pattern collects any references to ORUs within this context, so that the appropriate failed-component slots may be added to the case.

```
Reading FAILURE
```

Note that no phrases are associated with the word failure, since it is not particularly predictive in this domain.

```
Reading MODE
  (FAILURE MODE * *COLON*) = M-MODE-CONTEXT
```

Here is an example of a phrase that is stored with a concept other than the first element of the phrase. In this case, the parser checks to verify that the concepts in the phrase prior to the word "mode" are present in the proper lexical order. This is accomplished by checking the activation markers associated with the words or concepts in the phrase to see if they were activated at the appropriate time. Had they not been, the phrase would be terminated at this point. This particular phrase survives the test, and so is tracked. Note that it is a phrase that may establish a new parsing context.

```
Reading *COLON*
  (FAILURE MODE *COLON* *) = M-MODE-CONTEXT referenced
    Specializing: I-M-MODE-CONTEXT.425
    Removing: I-M-CONTEXT.420
    Specializing: I-M-CONTEXT.427
    Removing: I-M-IP.389.424
```

The recognition/creation of ordinary concepts is now abbreviated from this point on. The M-MODE-CONTEXT establishes a new context for the parser; there is now an expectation that failure modes will be processed next. With this new context, the old context is destroyed. At this point, the failure components that have been collected together (in this case, just M-GENERIC-RC) are added to the case (this is not reflected in the trace). Every time a new context is encountered, any processing in the prior context is resolved and terminated. This typically involves the addition of new slots to the case description. It is necessary to wait for the new context to finish processing, since the parser has no other means of determining when a section of the text is finished (people might rely on white space between sections, for example).

```
Reading LOSS
  (LOSS * OF OUTPUT - FAILURE DURING OPERATION) = M-OPERATIONAL-NO-OUTPUT
  (LOSS * OF OUTPUT - FAILURE TO START) = M-STARTUP-NO-OUTPUT
```

Two phrases are attached to the word "loss". Both will be tracked simultaneously until the patterns are either fulfilled, or terminated due to a failed lexical prediction.

```
Reading OF
  (LOSS OF * OUTPUT - FAILURE TO START) = M-STARTUP-NO-OUTPUT
  (LOSS OF * OUTPUT - FAILURE DURING OPERATION) = M-OPERATIONAL-NO-OUTPUT

Reading OUTPUT
  (LOSS OF OUTPUT * - FAILURE DURING OPERATION) = M-OPERATIONAL-NO-OUTPUT
  (LOSS OF OUTPUT * - FAILURE TO START) = M-STARTUP-NO-OUTPUT

Reading -
  (LOSS OF OUTPUT - * FAILURE TO START) = M-STARTUP-NO-OUTPUT
  (LOSS OF OUTPUT - * FAILURE DURING OPERATION) = M-OPERATIONAL-NO-OUTPUT

Reading FAILURE
  (LOSS OF OUTPUT - FAILURE * DURING OPERATION) = M-OPERATIONAL-NO-OUTPUT
  (LOSS OF OUTPUT - FAILURE * TO START) = M-STARTUP-NO-OUTPUT

Reading TO
  (LOSS OF OUTPUT - FAILURE TO * START) = M-STARTUP-NO-OUTPUT
```

At this point, one of the phrases fails to advance because of a failed lexical prediction. This phrase is no longer tracked by the system and will be removed at the appropriate time (i.e. when the end of a sentence is encountered).

```
Reading START
  (LOSS OF OUTPUT - FAILURE TO START *) = M-STARTUP-NO-OUTPUT referenced
    Specializing: I-M-STARTUP-NO-OUTPUT.430
     Specializing: I-M-HL-PATTERN.387.431
    Removing: I-M-IP.370.429
```

A high level pattern collecting together failure modes is now activated. At this point in the trace, the removal of index patterns will be omitted, except in noteworthy circumstances.

```
Reading FAILURE

Reading CAUSES
   (FAILURE CAUSES * *COLON*) = M-CAUSE-CONTEXT

Reading *COLON*
  (FAILURE CAUSES *COLON* *) = M-CAUSE-CONTEXT referenced
    Specializing: I-M-CAUSE-CONTEXT.433
    Specializing: I-M-CONTEXT.435
```

A new context is established in which the parser has expectations that failure causes will be processed next. The processing for failure modes is now wrapped up, with a slot for the failure mode I-M-STARTUP-NO-OUTPUT.430 being added to a case (again, this is not reflected in the trace).

```
Reading PIECE-PART
   (PIECE-PART * FAILURES) = M-FAULTY-COMPONENT-CAUSE

Reading FAILURES
  (PIECE-PART FAILURES *) = M-FAULTY-COMPONENT-CAUSE referenced
    Specializing: I-M-FAULTY-COMPONENT-CAUSE.437
    Specializing: I-M-HL-PATTERN.387.438

I-M-HL-PATTERN.387.438 is a pattern which collects together failure causes.

Reading *COMMA*

Reading CONTAMINATION
   (CONTAMINATION *) = M-CONTAMINATION-CAUSE referenced
    Specializing: I-M-CONTAMINATION-CAUSE.440
    Specializing: I-M-HL-PATTERN.387.441
```

Note that "contamination" is a fairly common word, and it activates the MOP M-CONTAMINATION-CAUSE each time it is encountered. However, such activation is irrelevant in any context outside of the failure cause context, since the high level pattern which collects such causes is inhibited in any other context. If this were in any context except the failure cause context, this would be an example in which something is activated within the system which causes no useful processing to occur.

```
Reading *COMMA*
```

```
Reading TEMPERATURE
  (TEMPERATURE * *LEFT-PAREN* HIGH OR LOW *RIGHT-PAREN*) =
      M-TEMPERATURE-OUT-OF-RANGE-CAUSE

Reading *LEFT-PAREN*
  (TEMPERATURE *LEFT-PAREN* * HIGH OR LOW *RIGHT-PAREN*) =
      M-TEMPERATURE-OUT-OF-RANGE-CAUSE

Reading HIGH
  (TEMPERATURE *LEFT-PAREN* HIGH * OR LOW *RIGHT-PAREN*) =
      M-TEMPERATURE-OUT-OF-RANGE-CAUSE

Reading OR
  (TEMPERATURE *LEFT-PAREN* HIGH OR * LOW *RIGHT-PAREN*) =
      M-TEMPERATURE-OUT-OF-RANGE-CAUSE

Reading LOW
  (TEMPERATURE *LEFT-PAREN* HIGH OR LOW * *RIGHT-PAREN*) =
      M-TEMPERATURE-OUT-OF-RANGE-CAUSE

Reading *RIGHT-PAREN*
  (TEMPERATURE *LEFT-PAREN* HIGH OR LOW *RIGHT-PAREN* *) =
      M-TEMPERATURE-OUT-OF-RANGE-CAUSE referenced
    Specializing: I-M-TEMPERATURE-OUT-OF-RANGE-CAUSE.443
    Specializing: I-M-HL-PATTERN.387.444
```

Note that parentheses are not handled in any general sense. They are merely incorporated within specific phrases. Since the text in the manuals is so stereotypical in format, this approach handles most cases that are encountered. A more general approach to processing parentheses remains to be developed.

```
Reading *COMMA*

Reading MECHANICAL
   (MECHANICAL * SHOCK) = M-MECHANICAL-SHOCK-CAUSE

Reading SHOCK
  (MECHANICAL SHOCK *) = M-MECHANICAL-SHOCK-CAUSE referenced
    Specializing: I-M-MECHANICAL-SHOCK-CAUSE.446
    Specializing: I-M-HL-PATTERN.387.447

Reading *COMMA*

Reading THERMAL
   (THERMAL * SHOCK) = M-THERMAL-SHOCK-CAUSE

Reading SHOCK
  (THERMAL SHOCK *) = M-THERMAL-SHOCK-CAUSE referenced
    Specializing: I-M-THERMAL-SHOCK-CAUSE.449
    Specializing: I-M-HL-PATTERN.387.450

Reading FAILURE
```

```
Reading DETECTION
  (FAILURE DETECTION * *SLASH* VERIFICATION *COLON*
       (SEQUENCE-OF-STEPS M-DETECTION-AND-GROUP)) = M-FAILURE-DETECTION
  (FAILURE DETECTION * *SLASH* VERIFICATION *COLON*) = M-DETECTION-CONTEXT
  (DETECTION * PROCEDURE *COLON*
       (SEQUENCE-OF-STEPS M-DETECTION-AND-GROUP)) = M-FAILURE-DETECTION

Reading *SLASH*
  (FAILURE DETECTION *SLASH* * VERIFICATION *COLON*) = M-DETECTION-CONTEXT
  (FAILURE DETECTION *SLASH* * VERIFICATION *COLON*
       (SEQUENCE-OF-STEPS M-DETECTION-AND-GROUP)) = M-FAILURE-DETECTION

Reading VERIFICATION
  (FAILURE DETECTION *SLASH* VERIFICATION * *COLON*
       (SEQUENCE-OF-STEPS M-DETECTION-AND-GROUP)) = M-FAILURE-DETECTION
  (FAILURE DETECTION *SLASH* VERIFICATION * *COLON*) = M-DETECTION-CONTEXT

Reading *COLON*
  (FAILURE DETECTION *SLASH* VERIFICATION *COLON* *) =
       M-DETECTION-CONTEXT referenced
     Specializing: I-M-DETECTION-CONTEXT.454
     Specializing: I-M-CONTEXT.456
   (FAILURE DETECTION *SLASH* VERIFICATION *COLON* *
       (SEQUENCE-OF-STEPS M-DETECTION-AND-GROUP)) = M-FAILURE-DETECTION
```

A new context is created for detection procedures. At this point, the failure causes have been added to the case representation.

```
Reading INDICATION
   (INDICATION * OF A (FAILED-COMPONENT M-ORU) FAILURE IS FIRST
       DETECTED BY THE (DETECTION-COMPONENT M-ORU) *PERIOD*) =
       M-NEXT-NODE-DETECTION

Reading OF
  (INDICATION OF * A (FAILED-COMPONENT M-ORU) FAILURE IS FIRST
     DETECTED BY THE (DETECTION-COMPONENT M-ORU) *PERIOD*) = M-NEXT-NODE-
DETECTION

Reading A
  (*LEFT-PAREN* A * *RIGHT-PAREN*) = M-ENUMERATION
  (INDICATION OF A * (FAILED-COMPONENT M-ORU) FAILURE IS FIRST
       DETECTED BY THE (DETECTION-COMPONENT M-ORU) *PERIOD*) =
       M-NEXT-NODE-DETECTION

Reading RC
   (RC *) = M-RC referenced
     Specializing: I-M-RC.460
  (INDICATION OF A (FAILED-COMPONENT M-ORU) * FAILURE IS FIRST
       DETECTED BY THE (DETECTION-COMPONENT M-ORU) *PERIOD*) =
       M-NEXT-NODE-DETECTION
```

The ring concentrator fulfills the requirement of finding an ORU at this point in the phrase, and so it is advanced. If and when M-NEXT-NODE-DETECTION is recognized, it will be instantiated with the slot and filler (FAILED-COMPONENT I-M-RC.460).

```
Reading FAILURE
  (INDICATION OF A (FAILED-COMPONENT I-M-RC.460) FAILURE * IS FIRST
      DETECTED BY THE (DETECTION-COMPONENT M-ORU) *PERIOD*) =
      M-NEXT-NODE-DETECTION

Reading IS
  (INDICATION OF A (FAILED-COMPONENT I-M-RC.460) FAILURE IS * FIRST
      DETECTED BY THE (DETECTION-COMPONENT M-ORU) *PERIOD*) =
      M-NEXT-NODE-DETECTION

Reading FIRST
  (INDICATION OF A (FAILED-COMPONENT I-M-RC.460) FAILURE IS FIRST *
      DETECTED BY THE (DETECTION-COMPONENT M-ORU) *PERIOD*) =
      M-NEXT-NODE-DETECTION

Reading DETECTED
  (INDICATION OF A (FAILED-COMPONENT I-M-RC.460) FAILURE IS FIRST
      DETECTED * BY THE (DETECTION-COMPONENT M-ORU) *PERIOD*) =
      M-NEXT-NODE-DETECTION

Reading BY
  (INDICATION OF A (FAILED-COMPONENT I-M-RC.460) FAILURE IS FIRST
      DETECTED BY * THE (DETECTION-COMPONENT M-ORU) *PERIOD*) =
      M-NEXT-NODE-DETECTION

Reading THE
  (INDICATION OF A (FAILED-COMPONENT I-M-RC.460) FAILURE IS FIRST
      DETECTED BY THE * (DETECTION-COMPONENT M-ORU) *PERIOD*) =
      M-NEXT-NODE-DETECTION

Reading *QUOTE*

Reading NEXT
  (*QUOTE* NEXT * *QUOTE* ACTIVE NODE ON THE NETWORK) = M-NEXT-RC
  (NEXT * ACTIVE RING CONCENTRATOR IN THE NETWORK) = M-NEXT-RC
```

Just as with parentheses, quotes are not yet handled in a systematic, general manner.

```
Reading *QUOTE*
  (*QUOTE* NEXT *QUOTE* * ACTIVE NODE ON THE NETWORK) = M-NEXT-RC

Reading ACTIVE
  (*QUOTE* NEXT *QUOTE* ACTIVE * NODE ON THE NETWORK) = M-NEXT-RC

Reading NODE
  (*QUOTE* NEXT *QUOTE* ACTIVE NODE * ON THE NETWORK) = M-NEXT-RC

Reading ON
  (*QUOTE* NEXT *QUOTE* ACTIVE NODE ON * THE NETWORK) = M-NEXT-RC
```

```
Reading THE
  (*QUOTE* NEXT *QUOTE* ACTIVE NODE ON THE * NETWORK) = M-NEXT-RC

Reading NETWORK
  (DMS NETWORK *) = M-CORE-NETWORK referenced
    Specializing: I-M-CORE-NETWORK.471
  (*QUOTE* NEXT *QUOTE* ACTIVE NODE ON THE NETWORK *) = M-NEXT-RC referenced
    Specializing: I-M-NEXT-RC.472
  (INDICATION OF A (FAILED-COMPONENT I-M-RC.460) FAILURE IS FIRST
      DETECTED BY THE (DETECTION-COMPONENT M-ORU) * *PERIOD*) =
      M-NEXT-NODE-DETECTION

Reading *PERIOD*
  (INDICATION OF A (FAILED-COMPONENT I-M-RC.460) FAILURE IS FIRST
      DETECTED BY THE (DETECTION-COMPONENT I-M-NEXT-RC.472)
      *PERIOD* *) = M-NEXT-NODE-DETECTION referenced
    Specializing: I-M-NEXT-NODE-DETECTION.474
  ((PROCEDURE M-FAILURE-DETECTION) * (EVENT M-EVENT-STEP)) =
      M-DISAMBIGUATE-DETECTION-EVENT
    Specializing: I-M-HL-PATTERN.409.476
    Specializing: I-M-HL-PATTERN.387.477
    Specializing: I-M-HL-PATTERN.416.478
    Removing: I-M-IP.407.457
    Removing: I-M-IP.328.467
    Removing: I-M-IP.395.458
    Removing: I-M-IP.380.453
    Removing: I-M-IP.371.428
```

Here a piece of the detection procedure from the text is recognized by activating a detection procedure called M-NEXT-NODE-DETECTION detailing the failed and detection components. Over the course of parsing the detection section, several such detection procedures may be instantiated as different parts of the procedure are recognized. All of these procedural fragments will ultimately be unified into one detection procedure when the context changes. The first sentence is in effect acting as a pointer to the type of procedure that is being described; the class of M-NEXT-NODE-DETECTION organizes procedures which involve one node noticing that another node has failed to communicate with it as expected, indicating a possible failure of that node. The first high level pattern I-M-HL-PATTERN.409.476 is the unification pattern which collects together instances of detection procedures for unification at a later time. The second pattern I-M-HL-PATTERN.387.477 is the one which gathers detection procedures together for inclusion in the case description. The last high level pattern is one which deals with building procedures from individual events in a procedure. The latter one is not relevant to continued processing. Notice that several index patterns are purged from the system at this time. Since the "*period*" was processed, the system assumes that the end of a sentence was encountered, and purges any useless patterns that were designed (perhaps by default) to be killed at sentence boundaries.

```
Reading SYSTEM
  (SYSTEM *) = M-SYSTEMS referenced
    Specializing: I-M-SYSTEMS.480
  ((EFFECTED-COMPONENT M-MISSION-COMPONENT) * *COLON*
      (SEQUENCE-OF-STEPS M-PROCEDURE)) = M-FAILURE-EFFECT
  (SYSTEM * MANAGEMENT) = M-SM

Reading MANAGEMENT
  (SYSTEM MANAGEMENT *) = M-SM referenced
    Recognizing: M-SM
    Creating: I-M-SM.483
    Removing: I-M-SM.483
    Specializing: I-M-SYSTEM-SM
```

Here is an example where an activated MOP specialized into a MOP that is already resident in memory. In this case, I-M-SYSTEM-SM happened to be predefined within the system. The parser makes use of pre-existing memory structures whenever possible.

```
Reading WILL

Reading REACH

Reading A
  (*LEFT-PAREN* A * *RIGHT-PAREN*) = M-ENUMERATION

Reading TIME-OUT
  ((RECIPIENT I-M-SYSTEM-SM) WILL REACH A TIME-OUT * LIMIT FOR RECEIPT
      OF THE (MESSAGE M-MESSAGE) *PERIOD*) = M-TIME-OUT-EVENT

Reading LIMIT
  ((RECIPIENT I-M-SYSTEM-SM) WILL REACH A TIME-OUT LIMIT * FOR RECEIPT
      OF THE (MESSAGE M-MESSAGE) *PERIOD*) = M-TIME-OUT-EVENT

Reading FOR
  ((RECIPIENT I-M-SYSTEM-SM) WILL REACH A TIME-OUT LIMIT FOR * RECEIPT
      OF THE (MESSAGE M-MESSAGE) *PERIOD*) = M-TIME-OUT-EVENT

Reading RECEIPT
  ((RECIPIENT I-M-SYSTEM-SM) WILL REACH A TIME-OUT LIMIT FOR RECEIPT *
      OF THE (MESSAGE M-MESSAGE) *PERIOD*) = M-TIME-OUT-EVENT

Reading OF
  ((RECIPIENT I-M-SYSTEM-SM) WILL REACH A TIME-OUT LIMIT FOR RECEIPT
      OF * THE (MESSAGE M-MESSAGE) *PERIOD*) = M-TIME-OUT-EVENT

Reading THE
  ((RECIPIENT I-M-SYSTEM-SM) WILL REACH A TIME-OUT LIMIT FOR RECEIPT
      OF THE * (MESSAGE M-MESSAGE) *PERIOD*) = M-TIME-OUT-EVENT

Reading NETWORK
  (DMS NETWORK *) = M-CORE-NETWORK referenced
    Specializing: I-M-CORE-NETWORK.471
```

Although a pattern causes M-CORE-NETWORK to be recognized in the text (the "DMS" part of the phrase is optional), no useful processing occurs, since there were no expectations for that MOP.

```
Reading TOKEN
  (NETWORK TOKEN *) = M-TOKEN referenced
    Specializing: I-M-TOKEN.491
  ((RECIPIENT I-M-SYSTEM-SM) WILL REACH A TIME-OUT LIMIT FOR RECEIPT
      OF THE (MESSAGE M-MESSAGE) * *PERIOD*) = M-TIME-OUT-EVENT

Reading *PERIOD*
  ((RECIPIENT I-M-SYSTEM-SM) WILL REACH A TIME-OUT LIMIT FOR RECEIPT
      OF THE (MESSAGE I-M-TOKEN.491) *PERIOD* *) = M-TIME-OUT-EVENT
      referenced
    Specializing: I-M-TIME-OUT-EVENT.492
    Specializing: I-M-HL-PATTERN.416.505
  ((PROCEDURE I-M-NEXT-NODE-DETECTION.474)
      (EVENT I-M-TIME-OUT-EVENT.492) *) =
      M-DISAMBIGUATE-DETECTION-EVENT referenced
    Specializing: I-M-DISAMBIGUATE-DETECTION-EVENT.506
    Creating: I-M-TIME-OUT-EVENT.144.507
    Recognizing: I-M-TIME-OUT-EVENT.144.507
```

Here an event is recognized from the text. Events cause the activation of a high level pattern that collects together those events occurring within a context. When a context switch occurs, the pattern MOP will search for the procedure that best organizes all of the events, and will recognize that procedure as the detection procedure being described. The MOP M-DISAMBIGUATE-DETECTION-EVENT ties together the procedure recognized from the first sentence to this event, such that some of the slots for the event can be fleshed out. Detection procedures are typically described by a pointer to the procedural class followed by an event from the procedure. This index pattern captures that stereotypical pattern of text usage in the FMEA manuals, acting as a processing aid for disambiguating the actual event.

```
Reading CORRECTIVE
  (CORRECTIVE * ACTION *COLON*) = M-CORRECTION-CONTEXT

Reading ACTION
  (CORRECTIVE ACTION * *COLON*) = M-CORRECTION-CONTEXT

Reading *COLON*
  (CORRECTIVE ACTION *COLON* *) = M-CORRECTION-CONTEXT referenced
    Specializing: I-M-CORRECTION-CONTEXT.531
    Specializing: I-M-CONTEXT.603
    Specializing: I-M-BUILD-PROCEDURE.533
    Specializing: I-M-AND-GROUP.145.534
    Specializing: I-M-BUILD-PROCEDURE.557
    Specializing: I-M-DETECTION-PROC.146.558
    Specializing: I-M-BUILD-PROCEDURE.560
    Specializing: I-M-DETECTION-AND-GROUP.147.561
    Specializing: I-M-BUILD-PROCEDURE.588
    Specializing: I-M-NEXT-NODE-DETECTION.589
  ((PROCEDURE M-FAILURE-DETECTION) * (EVENT M-EVENT-STEP)) =
```

```
      M-DISAMBIGUATE-DETECTION-EVENT
    Specializing: I-M-UNIFY-DETECTION-MOPS.595
    Specializing: I-M-NEXT-NODE-DETECTION.596
  ((PROCEDURE M-FAILURE-DETECTION) * (EVENT M-EVENT-STEP)) =
      M-DISAMBIGUATE-DETECTION-EVENT
```

As usual, with the recognition of a new context, the high level processing MOPs of the prior context are resolved and terminated. In this case, there are several high level patterns that were operating in the old context. The first was a pattern that collected together any referenced events (there was only one in this particular detection description). This pattern activates the M-BUILD-PROCEDURE MOP, which searches through memory for procedures which organize those events. Procedures are typically built up recursively; in this example, the I-M-NEXT-NODE-DETECTION.589 procedure is built up over several steps. It is important to note that this process is one of recognition. If there is no procedure in memory to organize the given events, a detection procedure cannot be recognized for the text. Once the events have been handled, another high level pattern is resolved. This one collects together procedures that occur in a context and unifies them (if they are of the same class) into a single procedure, collecting together all of the slots which characterize that MOP. In this case, the procedure created by the first sentence and the procedure built up from the event are unified together into a single detection procedure called I-M-NEXT-NODE-DETECTION.596. After this point, the detection procedure is added to the case description, and the parsing continues.

```
Reading *LEFT-PAREN*

Reading A
  (*LEFT-PAREN* A * *RIGHT-PAREN*) = M-ENUMERATION

Reading *RIGHT-PAREN*
  (*LEFT-PAREN* A *RIGHT-PAREN* *) = M-ENUMERATION referenced
    Specializing: I-M-ENUMERATION.605
    Specializing: I-M-BUILD-CONTEXT.606
    Specializing: I-M-CONTEXT.608
```

An enumerated list is processed by the parser by creating new subcontexts each time an enumeration tag is specified. The parser currently recognizes the tags of (a), (b), ..., as an enumerated sequence. The strategy to handle this is simple: a context switch is made to wrap up any of the higher level patterns for the prior context. Then that context is reasserted. Thus if an enumerated list with two enumerations occurs in the correction procedures section, the correction context will be asserted twice. This has the effect of wrapping up any processing that needs to be resolved, while still maintaining a given context.

```
Reading SHORT
  (SHORT * TERM) = M-SHORT-TERM

Reading TERM
  (SHORT TERM *) = M-SHORT-TERM referenced
    Specializing: I-M-SHORT-TERM.610
  ((FRAMEWORK M-FRAMEWORK) * *COLON*) = M-FAILURE-CORRECTION
```

Each correction procedure has a framework specifier of either short-term or long-term.

```
Reading *COLON*
  ((FRAMEWORK I-M-SHORT-TERM.610) *COLON* *) = M-FAILURE-CORRECTION referenced
     Specializing: I-M-FAILURE-CORRECTION.612
     Specializing: I-M-HL-PATTERN.404.613
     Specializing: I-M-HL-PATTERN.387.614
     Specializing: I-M-HL-PATTERN.416.615
```

A failure correction MOP is instantiated here with the framework recorded as a slot filler. Just as with the detection procedure, multiple correction procedures may be activated, each of which records some information about the correction procedure being described in the text. These will be unified at the next context switch into the final correction procedure. The high level patterns are those that are concerned with building up procedures (which, recall, may be composed of events and other procedures), with unification, and with building the case.

```
Reading NETWORK
  (DMS NETWORK *) = M-CORE-NETWORK referenced
     Specializing: I-M-CORE-NETWORK.471

Reading RECONFIGURATION
  ((CONFIGURATION I-M-CORE-NETWORK.471) RECONFIGURATION * IS EFFECTED
        AUTOMATICALLY *PERIOD*) = M-AUTO-RECONFIG

Reading IS
  ((CONFIGURATION I-M-CORE-NETWORK.471) RECONFIGURATION IS * EFFECTED
        AUTOMATICALLY *PERIOD*) = M-AUTO-RECONFIG

Reading EFFECTED
  ((CONFIGURATION I-M-CORE-NETWORK.471) RECONFIGURATION IS EFFECTED *
        AUTOMATICALLY *PERIOD*) = M-AUTO-RECONFIG

Reading AUTOMATICALLY
  ((CONFIGURATION I-M-CORE-NETWORK.471) RECONFIGURATION IS EFFECTED
        AUTOMATICALLY * *PERIOD*) = M-AUTO-RECONFIG

Reading *PERIOD*
  ((CONFIGURATION I-M-CORE-NETWORK.471) RECONFIGURATION IS EFFECTED
        AUTOMATICALLY *PERIOD* *) = M-AUTO-RECONFIG referenced
     Specializing: I-M-AUTO-RECONFIG.648
    Inferring--> M-RECONFIGURATION
     Specializing: I-M-RECONFIGURATION.649
     Specializing: I-M-HL-PATTERN.404.650
     Specializing: I-M-HL-PATTERN.387.651
```

The MOP M-AUTO-RECONFIG characterizes the process of a network reconfiguring itself under software control. The major part of this act is described by the M-RECONFIGURATION class of correction procedures, which organizes those procedures that reconfigure the network in some manner; the functionality of the M-AUTO-RECONFIG MOP therefore infers the M-RECONFIGURATION procedure. Since this is a type of correction procedure, it will eventually be unified along with the other correction procedure fragments.

```
Reading THE
```

```
Reading DMS

Reading NETWORK
  (DMS NETWORK *) = M-CORE-NETWORK referenced
    Specializing: I-M-CORE-NETWORK.471

Reading REMAINS
  (THE (OBJECT I-M-CORE-NETWORK.471) REMAINS * IN OPERATION) = M-OPERATIONAL

Reading IN
  (THE (OBJECT I-M-CORE-NETWORK.471) REMAINS IN * OPERATION) = M-OPERATIONAL

Reading OPERATION
  (THE (OBJECT I-M-CORE-NETWORK.471) REMAINS IN OPERATION *) =
      M-OPERATIONAL referenced
    Specializing: I-M-OPERATIONAL.670
```

This is the first example of a state that has been explicitly mentioned in the text. Other states have already been implicitly instantiated, however, as preconditions and results to the events and procedures that have been recognized previously in the text.

```
Reading IN

Reading A
  (*LEFT-PAREN* A * *RIGHT-PAREN*) = M-ENUMERATION

Reading RECONFIGURED
  ((RESULT I-M-OPERATIONAL.670) IN A RECONFIGURED * STATE WITH THE
      FAILED (FAILED-COMPONENT M-ORU) BYPASSED) = M-BYPASS-NODE

Reading STATE
  ((RESULT I-M-OPERATIONAL.670) IN A RECONFIGURED STATE * WITH THE
      FAILED (FAILED-COMPONENT M-ORU) BYPASSED) = M-BYPASS-NODE

Reading WITH
  ((RESULT I-M-OPERATIONAL.670) IN A RECONFIGURED STATE WITH * THE
      FAILED (FAILED-COMPONENT M-ORU) BYPASSED) = M-BYPASS-NODE

Reading THE
  ((RESULT I-M-OPERATIONAL.670) IN A RECONFIGURED STATE WITH THE *
      FAILED (FAILED-COMPONENT M-ORU) BYPASSED) = M-BYPASS-NODE

Reading FAILED
  ((RESULT I-M-OPERATIONAL.670) IN A RECONFIGURED STATE WITH THE
      FAILED * (FAILED-COMPONENT M-ORU) BYPASSED) = M-BYPASS-NODE

Reading RC
  (RC *) = M-RC referenced
    Specializing: I-M-RC.460
  ((RESULT I-M-OPERATIONAL.670) IN A RECONFIGURED STATE WITH THE
      FAILED (FAILED-COMPONENT M-ORU) * BYPASSED) = M-BYPASS-NODE
```

```
Reading BYPASSED
  ((RESULT I-M-OPERATIONAL.670) IN A RECONFIGURED STATE WITH THE
      FAILED (FAILED-COMPONENT I-M-RC.460) BYPASSED *) =
      M-BYPASS-NODE referenced
    Specializing: I-M-BYPASS-NODE.681
    Specializing: I-M-HL-PATTERN.404.682
    Specializing: I-M-HL-PATTERN.387.683
    Specializing: I-M-HL-PATTERN.416.684
```

Here a more specific instance of the M-RECONFIGURATION class has been activated. The high level patterns are the usual ones activated with a procedure.

```
Reading *PERIOD*
    Removing: I-M-IP.395.672

Reading *LEFT-PAREN*

Reading B
  (*LEFT-PAREN* B * *RIGHT-PAREN*) = M-ENUMERATION

Reading *RIGHT-PAREN*
  (*LEFT-PAREN* B *RIGHT-PAREN* *) = M-ENUMERATION referenced
    Specializing: I-M-ENUMERATION.605
    Specializing: I-M-BUILD-CONTEXT.606
    Specializing: I-M-CONTEXT.697
    Removing: I-M-HL-PATTERN.416.684
    Removing: I-M-HL-PATTERN.416.652
    Removing: I-M-HL-PATTERN.416.615
    Specializing: I-M-BUILD-PROCEDURE.689
    Removing: I-M-HL-PATTERN.404.682
    Removing: I-M-HL-PATTERN.404.650
    Removing: I-M-HL-PATTERN.404.613
    Specializing: I-M-UNIFY-CORRECTION-MOPS.690
    Specializing: I-M-BYPASS-NODE.691
    Specializing: I-M-UNIFY-CORRECTION-MOPS.693
    Specializing: I-M-HL-PATTERN.387.694
    Specializing: I-M-BUILD-PROCEDURE.696
```

With the enumeration tag, we have a context switch again. The M-BUILD-PROCEDURE MOP attempts to find a procedure in memory which organizes any of the correction procedures that have been activated, since procedures may be composed of subprocedures. This task fails in this instance, since there are no such procedures involving these correction MOPs. The unification then occurs, creating the final instance of I-M-BYPASS-NODE.691. At this point, the correction procedure is added to the case description.

```
Reading LONG
  (LONG * TERM) = M-LONG-TERM

Reading TERM
  (LONG TERM *) = M-LONG-TERM referenced
    Specializing: I-M-LONG-TERM.699
  ((FRAMEWORK M-FRAMEWORK) * *COLON*) = M-FAILURE-CORRECTION
```

```
Reading *COLON*
  ((FRAMEWORK I-M-LONG-TERM.699) *COLON* *) = M-FAILURE-CORRECTION referenced
     Specializing: I-M-FAILURE-CORRECTION.701
     Specializing: I-M-HL-PATTERN.404.702
     Specializing: I-M-HL-PATTERN.387.703
     Specializing: I-M-HL-PATTERN.416.704

Reading THE

Reading CREWMEN
  (THE CREWMEN *) = M-CREW referenced
     Specializing: I-M-CREW.737
  ((EFFECTED-COMPONENT M-MISSION-COMPONENT) * *COLON*
        (SEQUENCE-OF-STEPS M-PROCEDURE)) = M-FAILURE-EFFECT

Reading CHECK
  ((ACTOR I-M-CREW.737) CHECK * FOR (OBJECT M-SYSTEM)) = M-VERIFY-EVENT

Reading FOR
  ((ACTOR I-M-CREW.737) CHECK FOR * (OBJECT M-SYSTEM)) = M-VERIFY-EVENT

Reading *QUOTE*

Reading APPLIED
  (*QUOTE* APPLIED * POWER *QUOTE*) = M-POWER-SYSTEM

Reading POWER
  (*QUOTE* APPLIED POWER * *QUOTE*) = M-POWER-SYSTEM

Reading *QUOTE*
  (*QUOTE* APPLIED POWER *QUOTE* *) = M-POWER-SYSTEM referenced
     Specializing: I-M-POWER-SYSTEM.308
  ((ACTOR I-M-CREW.737) CHECK FOR (OBJECT I-M-POWER-SYSTEM.308) *) =
        M-VERIFY-EVENT referenced
     Specializing: I-M-VERIFY-EVENT.742
     Specializing: I-M-HL-PATTERN.416.745
```

Note that although the event describing the verification of the power system has been activated, it is not clear what the power system is a component of from this sentence. A context for this event will eventually be provided when the correction procedure which organizes it is recognized. At that time, the device under consideration (a ring concentrator) will be referenced along with this event, thereby providing the explicit connection between the power system and the ring concentrator.

```
Reading AND

Reading THE

Reading CREWMEN
  (THE CREWMEN *) = M-CREW referenced
     Specializing: I-M-CREW.737
  ((EFFECTED-COMPONENT M-MISSION-COMPONENT) * *COLON*
        (SEQUENCE-OF-STEPS M-PROCEDURE)) = M-FAILURE-EFFECT
```

```
Reading CHECK
  ((ACTOR I-M-CREW.737) CHECK * FOR (OBJECT M-SYSTEM)) = M-VERIFY-EVENT

Reading FOR
  ((ACTOR I-M-CREW.737) CHECK FOR * (OBJECT M-SYSTEM)) = M-VERIFY-EVENT

Reading *QUOTE*

Reading CONNECTOR
  (*QUOTE* CONNECTOR * TIGHTNESS *QUOTE*) = M-CONNECTORS

Reading TIGHTNESS
  (*QUOTE* CONNECTOR TIGHTNESS * *QUOTE*) = M-CONNECTORS

Reading *QUOTE*
  (*QUOTE* CONNECTOR TIGHTNESS *QUOTE* *) = M-CONNECTORS referenced
    Specializing: I-M-CONNECTORS.310
  ((ACTOR I-M-CREW.737) CHECK FOR (OBJECT I-M-CONNECTORS.310) *) =
      M-VERIFY-EVENT referenced
    Specializing: I-M-VERIFY-EVENT.754
    Specializing: I-M-HL-PATTERN.416.757
```

Here is a second event from the current correction procedure. This illustrates how multiple events from a procedure may be specified within a context.

```
Reading *PERIOD*

Reading IF

Reading THE

Reading RC
  (RC *) = M-RC referenced
    Specializing: I-M-RC.460

Reading CANNOT

Reading THEN

Reading BE

Reading PLACED

Reading IN

Reading OPERATION

Reading *COMMA*

Reading THE
```

```
Reading RC
  (RC *) = M-RC referenced
    Specializing: I-M-RC.460

Reading IS

Reading REMOVED

Reading AND

Reading REPLACED
  (THE (FAILED-COMPONENT I-M-RC.460) IS REMOVED AND REPLACED * WITH
       AN ORU LOGISTICS SPARE *PERIOD*) = M-REPLACE-WITH-SPARE

Reading WITH
  (THE (FAILED-COMPONENT I-M-RC.460) IS REMOVED AND REPLACED WITH *
       AN ORU LOGISTICS SPARE *PERIOD*) = M-REPLACE-WITH-SPARE

Reading AN
  (THE (FAILED-COMPONENT I-M-RC.460) IS REMOVED AND REPLACED WITH AN
       * ORU LOGISTICS SPARE *PERIOD*) = M-REPLACE-WITH-SPARE

Reading ORU
  (THE (FAILED-COMPONENT I-M-RC.460) IS REMOVED AND REPLACED WITH AN
       ORU * LOGISTICS SPARE *PERIOD*) = M-REPLACE-WITH-SPARE

Reading LOGISTICS
  (THE (FAILED-COMPONENT I-M-RC.460) IS REMOVED AND REPLACED WITH AN
       ORU LOGISTICS * SPARE *PERIOD*) = M-REPLACE-WITH-SPARE

Reading SPARE
  (THE (FAILED-COMPONENT I-M-RC.460) IS REMOVED AND REPLACED WITH AN
       ORU LOGISTICS SPARE * *PERIOD*) = M-REPLACE-WITH-SPARE

Reading *PERIOD*
  (THE (FAILED-COMPONENT I-M-RC.460) IS REMOVED AND REPLACED WITH AN
       ORU LOGISTICS SPARE *PERIOD* *) = M-REPLACE-WITH-SPARE referenced
    Specializing: I-M-REPLACE-WITH-SPARE.778
    Specializing: I-M-HL-PATTERN.404.779
    Specializing: I-M-HL-PATTERN.387.780
    Specializing: I-M-HL-PATTERN.416.781
```

Note that this sentence begins with an "if" conditional clause. This is ignored by the current system, since conditionals are not represented within the knowledge representation. This decision was motivated by the desire to simplify some of the issues involved in parsing text; the addition of conditionals would have significantly complicated processing.

```
Reading FAILURE

Reading EFFECT
  (FAILURE EFFECT * ON *COLON*) = M-EFFECT-CONTEXT
  (FAILURE EFFECT * ON THE (EFFECTED-COMPONENT M-MISSION-COMPONENT)
       *COLON* (SEQUENCE-OF-STEPS M-GROUP)) = M-FAILURE-EFFECT
```

```
Reading ON
  (FAILURE EFFECT ON * THE (EFFECTED-COMPONENT M-MISSION-COMPONENT)
       *COLON* (SEQUENCE-OF-STEPS M-GROUP)) = M-FAILURE-EFFECT
  (FAILURE EFFECT ON * *COLON*) = M-EFFECT-CONTEXT

Reading *COLON*
  (FAILURE EFFECT ON *COLON* *) = M-EFFECT-CONTEXT referenced
    Specializing: I-M-EFFECT-CONTEXT.784
    Specializing: I-M-BUILD-PROCEDURE.786
    Specializing: I-M-AND-GROUP.313.787
    Specializing: I-M-BUILD-PROCEDURE.826
    Specializing: I-M-AND-GROUP.314.827
    Specializing: I-M-BUILD-PROCEDURE.885
    Specializing: I-M-CORRECTION-PROC.315.886
    Specializing: I-M-BUILD-PROCEDURE.888
    Specializing: I-M-CORRECTION-AND-GROUP.316.889
    Specializing: I-M-BUILD-PROCEDURE.931
    Specializing: I-M-NON-OPERATIONAL-REPLACEMENT.932
    Specializing: I-M-HL-PATTERN.404.933
    Specializing: I-M-HL-PATTERN.387.934
    Specializing: I-M-BUILD-PROCEDURE.936
    Specializing: I-M-UNIFY-CORRECTION-MOPS.937
    Specializing: I-M-FAILURE-CORRECTION.938
    Specializing: I-M-CONTEXT.956
```

Once again, the build procedure and unification processes kick in and create the correction procedure I-M-NON-OPERATIONAL-REPLACEMENT.932 described in the text. In this instance, the parser has to recognize a correction procedure that organizes three events mentioned in the text. The M-BUILD-PROCEDURE is general enough to find a candidate procedure that best characterizes any number of events. At this point in the parsing of the text, both of the correction procedures have been added to the case description.

```
Reading *LEFT-PAREN*

Reading A
  (*LEFT-PAREN* A * *RIGHT-PAREN*) = M-ENUMERATION

Reading *RIGHT-PAREN*
  (*LEFT-PAREN* A *RIGHT-PAREN* *) = M-ENUMERATION referenced
    Specializing: I-M-ENUMERATION.605
    Specializing: I-M-BUILD-CONTEXT.959
    Specializing: I-M-CONTEXT.961

Reading CREW
  (CREW *) = M-CREW referenced
    Specializing: I-M-CREW.737
  ((EFFECTED-COMPONENT M-MISSION-COMPONENT) * *COLON*
       (SEQUENCE-OF-STEPS M-PROCEDURE)) = M-FAILURE-EFFECT

Reading *SLASH*
```

```
Reading SSPE
  (CREW *SLASH* SSPE *) = M-CREW referenced
    Specializing: I-M-CREW.737
  ((EFFECTED-COMPONENT M-MISSION-COMPONENT) * *COLON*
      (SEQUENCE-OF-STEPS M-PROCEDURE)) = M-FAILURE-EFFECT

Reading *COLON*
  ((EFFECTED-COMPONENT I-M-CREW.737) *COLON* *
      (SEQUENCE-OF-STEPS M-PROCEDURE)) = M-FAILURE-EFFECT
  ((EFFECTED-COMPONENT I-M-CREW.737) *COLON* *
      (SEQUENCE-OF-STEPS M-PROCEDURE)) = M-FAILURE-EFFECT

Reading NONE
  (NONE * *PERIOD*) = M-DISAMBIGUATE-NONE

Reading *PERIOD*
  (NONE *PERIOD* *) = M-DISAMBIGUATE-NONE referenced
    Specializing: I-M-DISAMBIGUATE-NONE.969
    Recognizing: I-M-NULL-PROCEDURE
  ((EFFECTED-COMPONENT I-M-CREW.737) *COLON*
      (SEQUENCE-OF-STEPS I-M-NULL-PROCEDURE) *) = M-FAILURE-EFFECT
      referenced
    Specializing: I-M-FAILURE-EFFECT.970
  ((SEQUENCE-OF-STEPS M-FAILURE-EFFECT) *
      (JUSTIFICATION M-FAILURE-CORRECTION)) = M-SEARCH-FOR-EFFECT-REFERENT
    Specializing: I-M-HL-PATTERN.387.972
    Specializing: I-M-HL-PATTERN.416.973
```

The word "none" when used in the context of a failure effect usually indicates that there is no effect. That same word in another context might mean something entirely different. The MOP M-DISAMBIGUATE-NONE checks for the current context to disambiguate the meaning of that word. There are certainly other (possibly more general) methods of disambiguating a word within a given context; this one was chosen for ease of implementation. "None" in this context therefore causes the activation of I-M-NULL-PROCEDURE, the concept that is used to represent the lack of a procedure.

```
Reading THE

Reading CONDITIONS
  (THE CONDITIONS * ASSOCIATED WITH THE REPLACEMENT OF A RC WITHIN
      THE NETWORK CONFIGURATION ARE SUCH THAT THE NETWORK IS
      ALREADY IN A RECONFIGURED STATE SUPPLYING FULL SERVICES
      *PERIOD*) = M-BYPASS-NODE
```

This pattern demonstrates that index patterns can be quite specific and therefore not of general utility if not defined properly. This pattern is effective only for a sentence that follows this precise format. A better definition of the pattern would be to generalize it. Minimally, each of the entities mentioned could be replaced by an ORU concept filler, which would allow the phrase to be used in conjunction with any of the ORU devices. Since this is such a long phrase, it will be omitted from the trace until it is actually recognized.

```
Reading ASSOCIATED
```

Reading WITH

Reading THE

Reading REPLACEMENT

Reading OF

Reading A
  (*LEFT-PAREN* A * *RIGHT-PAREN*) = M-ENUMERATION

Reading RC
  (RC *) = M-RC referenced
    Specializing: I-M-RC.460

Reading WITHIN

Reading THE

Reading NETWORK
  (DMS NETWORK *) = M-CORE-NETWORK referenced
    Specializing: I-M-CORE-NETWORK.471

Reading CONFIGURATION

Reading ARE

Reading SUCH

Reading THAT

Reading THE

Reading NETWORK
  (DMS NETWORK *) = M-CORE-NETWORK referenced
    Specializing: I-M-CORE-NETWORK.471

Reading IS

Reading ALREADY

Reading IN

Reading A
  (*LEFT-PAREN* A * *RIGHT-PAREN*) = M-ENUMERATION

Reading RECONFIGURED
  ((RESULT I-M-OPERATIONAL.670) IN A RECONFIGURED * STATE WITH THE
      FAILED (FAILED-COMPONENT M-ORU) BYPASSED) = M-BYPASS-NODE

Reading STATE
  ((RESULT I-M-OPERATIONAL.670) IN A RECONFIGURED STATE * WITH THE
      FAILED (FAILED-COMPONENT M-ORU) BYPASSED) = M-BYPASS-NODE

```
Reading SUPPLYING

Reading FULL

Reading SERVICES

Reading *PERIOD*
  (THE CONDITIONS ASSOCIATED WITH THE REPLACEMENT OF A RC WITHIN THE
      NETWORK CONFIGURATION ARE SUCH THAT THE NETWORK IS ALREADY IN
      A RECONFIGURED STATE SUPPLYING FULL SERVICES *PERIOD* *) =
      M-BYPASS-NODE referenced
    Specializing: I-M-BYPASS-NODE.996
    Specializing: I-M-HL-PATTERN.404.997
  ((SEQUENCE-OF-STEPS I-M-FAILURE-EFFECT.970)
      (JUSTIFICATION I-M-BYPASS-NODE.996) *) =
      M-SEARCH-FOR-EFFECT-REFERENT referenced
    Specializing: I-M-SEARCH-FOR-EFFECT-REFERENT.999
    Specializing: I-M-HL-PATTERN.416.1000
```

With failure effects, the FMEA manuals generally follow one of two patterns. They either detail a procedure that represents the effect on a system component (this may be the null procedure), or they specify a procedure and give a justification for those effects which references one of the correction procedures in the same case. The latter instance occurs with this first failure effect. The justification detailed is the M-BYPASS-NODE procedure from the failure corrections section (the argument is essentially that there is no effect because the failed component is automatically bypassed). When such a justification is given, the parser must search back over the failure correction procedures to determine which one is being used as the justification. The M-BYPASS-NODE procedure instantiated by the justification is used to find the matching M-BYPASS-PROCEDURE from the corrections section; in this case, I-M-BYPASS-NODE.691 is found (although this is not reflected in the trace). This procedure is then used as the filler for the failure effect slot.

```
Reading *LEFT-PAREN*

Reading B
  (*LEFT-PAREN* B * *RIGHT-PAREN*) = M-ENUMERATION

Reading *RIGHT-PAREN*
  (*LEFT-PAREN* B *RIGHT-PAREN* *) = M-ENUMERATION referenced
    Specializing: I-M-ENUMERATION.605
    Specializing: I-M-BUILD-CONTEXT.959
    Specializing: I-M-CONTEXT.1007
```

With the new failure effect context being established, the first failure effect is added to the case description.

```
Reading MISSION
  (MISSION * SUPPORT) = M-MISSION-SUPPORT
```

```
Reading SUPPORT
  (MISSION SUPPORT *) = M-MISSION-SUPPORT referenced
    Specializing: I-M-MISSION-SUPPORT.1009
  ((EFFECTED-COMPONENT M-MISSION-COMPONENT) * *COLON*
       (SEQUENCE-OF-STEPS M-PROCEDURE)) = M-FAILURE-EFFECT

Reading *COLON*
  ((EFFECTED-COMPONENT I-M-MISSION-SUPPORT.1009) *COLON* *
       (SEQUENCE-OF-STEPS M-PROCEDURE)) = M-FAILURE-EFFECT

Reading NONE
  (NONE * *PERIOD*) = M-DISAMBIGUATE-NONE

Reading *PERIOD*
  (NONE *PERIOD* *) = M-DISAMBIGUATE-NONE referenced
    Specializing: I-M-DISAMBIGUATE-NONE.969
    Recognizing: I-M-NULL-PROCEDURE
  ((EFFECTED-COMPONENT I-M-MISSION-SUPPORT.1009) *COLON*
       (SEQUENCE-OF-STEPS I-M-NULL-PROCEDURE) *) = M-FAILURE-EFFECT
       referenced
    Specializing: I-M-FAILURE-EFFECT.1013
  ((SEQUENCE-OF-STEPS M-FAILURE-EFFECT) *
       (JUSTIFICATION M-FAILURE-CORRECTION)) = M-SEARCH-FOR-EFFECT-REFERENT
    Specializing: I-M-HL-PATTERN.387.1015
    Specializing: I-M-HL-PATTERN.416.1016

Reading *LEFT-PAREN*

Reading C
  (*LEFT-PAREN* C * *RIGHT-PAREN*) = M-ENUMERATION

Reading *RIGHT-PAREN*
  (*LEFT-PAREN* C *RIGHT-PAREN* *) = M-ENUMERATION referenced
    Specializing: I-M-ENUMERATION.605
    Specializing: I-M-BUILD-CONTEXT.959
    Specializing: I-M-CONTEXT.1023
```

The second failure effect is now added to the case description. This one has just the null procedure as its filler.

```
Reading SYSTEM
  (SYSTEM *) = M-SYSTEMS referenced
    Specializing: I-M-SYSTEMS.480
  ((EFFECTED-COMPONENT M-MISSION-COMPONENT) * *COLON*
       (SEQUENCE-OF-STEPS M-PROCEDURE)) = M-FAILURE-EFFECT
  (SYSTEM * MANAGEMENT) = M-SM

Reading *COLON*
  ((EFFECTED-COMPONENT I-M-SYSTEMS.480) *COLON* *
       (SEQUENCE-OF-STEPS M-PROCEDURE)) = M-FAILURE-EFFECT

Reading NONE
  (NONE * *PERIOD*) = M-DISAMBIGUATE-NONE
```

```
Reading *PERIOD*
  (NONE *PERIOD* *) = M-DISAMBIGUATE-NONE referenced
    Specializing: I-M-DISAMBIGUATE-NONE.969
    Recognizing: I-M-NULL-PROCEDURE
  ((EFFECTED-COMPONENT I-M-SYSTEMS.480) *COLON*
      (SEQUENCE-OF-STEPS I-M-NULL-PROCEDURE) *) = M-FAILURE-EFFECT
      referenced
    Specializing: I-M-FAILURE-EFFECT.1030
  ((SEQUENCE-OF-STEPS M-FAILURE-EFFECT) *
      (JUSTIFICATION M-FAILURE-CORRECTION)) = M-SEARCH-FOR-EFFECT-REFERENT
    Specializing: I-M-HL-PATTERN.387.1032
    Specializing: I-M-HL-PATTERN.416.1033

Reading *LEFT-PAREN*

Reading D
  (*LEFT-PAREN* D * *RIGHT-PAREN*) = M-ENUMERATION

Reading *RIGHT-PAREN*
  (*LEFT-PAREN* D *RIGHT-PAREN* *) = M-ENUMERATION referenced
    Specializing: I-M-ENUMERATION.605
    Recognizing: M-CONTEXT
    Specializing: I-M-CONTEXT.1040
```

The third failure effect is added to the case description at this point.

```
Reading INTERFACES
  (INTERFACES *) = M-INTERFACES referenced
    Specializing: I-M-INTERFACES.1042
  ((EFFECTED-COMPONENT M-MISSION-COMPONENT) * *COLON*
   (SEQUENCE-OF-STEPS M-PROCEDURE)) = M-FAILURE-EFFECT

Reading *COLON*
  ((EFFECTED-COMPONENT I-M-INTERFACES.1042) *COLON* *
      (SEQUENCE-OF-STEPS M-PROCEDURE)) = M-FAILURE-EFFECT

Reading NONE
  (NONE * *PERIOD*) = M-DISAMBIGUATE-NONE

Reading *PERIOD*
  (NONE *PERIOD* *) = M-DISAMBIGUATE-NONE referenced
    Specializing: I-M-DISAMBIGUATE-NONE.969
    Recognizing: I-M-NULL-PROCEDURE
  ((EFFECTED-COMPONENT I-M-INTERFACES.1042) *COLON*
      (SEQUENCE-OF-STEPS I-M-NULL-PROCEDURE) *) = M-FAILURE-EFFECT
      referenced
    Specializing: I-M-FAILURE-EFFECT.1046
  ((SEQUENCE-OF-STEPS M-FAILURE-EFFECT) *
      (JUSTIFICATION M-FAILURE-CORRECTION)) = M-SEARCH-FOR-EFFECT-REFERENT
    Specializing: I-M-HL-PATTERN.387.1048
    Specializing: I-M-HL-PATTERN.416.1049
```

```
Reading *EOC*
    Specializing: I-M-CONTEXT.1051
    Specializing: I-M-CASE.1053
```

The *EOC* is used to mark the end of a case text. It also causes a context switch, which wraps up the processing of the last effect. Finally, the *EOC* symbol causes the termination of the high level pattern that builds up case descriptions. The final case structure is now instantiated, and the parsing process is complete for the given input.

This is the final case description that is built by the parser:

```
>(dph `I-M-CASE.1053)
(I-M-CASE.1053 (FAILED-COMPONENT I-M-GENERIC-RC.422)
   (MODE I-M-STARTUP-NO-OUTPUT.1089
      (SEQUENCE-OF-STEPS I-M-AND-GROUP.97.1085
         (2 I-M-WAIT-TO-RECEIVE-EVENT.95.1071
          (RESULT I-M-TIME-ELAPSED-FOR-OBJECT.25.1056
                (OBJECT I-M-SYSTEM-SM))
          (RESULT I-M-READY-TO-RECEIVE.22.1059
                (RECIPIENT I-M-SYSTEM-SM)
                (OBJECT I-M-TOKEN.491))
          (ACTION I-M-WAIT-FOR-MESSAGE.23.1064
                (ACTOR I-M-SYSTEM-SM) (RECIPIENT I-M-SYSTEM-SM)
                (OBJECT I-M-TOKEN.491))
          (PRECOND I-M-READY-TO-RECEIVE.22.1059
                 (RECIPIENT I-M-SYSTEM-SM)
                 (OBJECT I-M-TOKEN.491))
          (RECIPIENT I-M-SYSTEM-SM) (MESSAGE I-M-TOKEN.491))
         (3 I-M-TIME-OUT-EVENT.96.1084
          (RESULT I-M-TRANSMISSION-TIMED-OUT.41.502
                (RECIPIENT I-M-SYSTEM-SM)
                (OBJECT I-M-TOKEN.491))
          (ACTION I-M-TIME-OUT.40.499 (ACTOR I-M-SYSTEM-SM)
                (RECIPIENT I-M-SYSTEM-SM)
                (OBJECT I-M-TOKEN.491))
          (PRECOND I-M-READY-TO-RECEIVE.36.495
                 (RECIPIENT I-M-SYSTEM-SM)
                 (OBJECT I-M-TOKEN.491))
          (RECIPIENT I-M-SYSTEM-SM) (MESSAGE I-M-TOKEN.491)))
       (PRECOND I-M-READY-TO-RECEIVE.93.1088
             (RECIPIENT I-M-SYSTEM-SM) (OBJECT I-M-TOKEN.491)))
   (CAUSE I-M-FAULTY-COMPONENT-CAUSE.1094
   (CAUSE I-M-CONTAMINATION-CAUSE.440)
   (CAUSE I-M-TEMPERATURE-OUT-OF-RANGE-CAUSE.443)
   (CAUSE I-M-MECHANICAL-SHOCK-CAUSE.446)
   (CAUSE I-M-THERMAL-SHOCK-CAUSE.449)
   (DETECTION I-M-NEXT-NODE-DETECTION.1150
      (RESULT I-M-NON-OPERATIONAL.148.1100
            (OBJECT I-M-GENERIC-RC.422))
      (PRECOND I-M-POWER-ON.141.1102 (OBJECT I-M-NEXT-RC.472))
      (PRECOND I-M-NEXT-NODE.140.1105 (OBJECT1 I-M-NEXT-RC.472)
            (OBJECT2 I-M-GENERIC-RC.422))
      (PRECOND I-M-LOGICALLY-CONNECTED.137.1107
```

```
        (OBJECT1 I-M-GENERIC-RC.422))
    (PRECOND I-M-PHYSICALLY-CONNECTED.136.1109
        (OBJECT1 I-M-GENERIC-RC.422))
    (FAILED-COMPONENT I-M-GENERIC-RC.422)
    (DETECTION-COMPONENT I-M-NEXT-RC.472)
    (SEQUENCE-OF-STEPS I-M-DETECTION-AND-GROUP.147.1147
      (1 I-M-DETECTION-PROC.146.1144
       (PRECOND I-M-READY-TO-RECEIVE.142.563
           (OBJECT I-M-TOKEN.491))
       (SEQUENCE-OF-STEPS I-M-AND-GROUP.145.1142
         (1 I-M-WAIT-TO-RECEIVE-EVENT.143.1123
          (RESULT I-M-TIME-ELAPSED-FOR-OBJECT.25.1056
              (OBJECT I-M-SYSTEM-SM))
          (RESULT I-M-READY-TO-RECEIVE.22.536
              (OBJECT I-M-TOKEN.491))
          (ACTION I-M-WAIT-FOR-MESSAGE.23.538
              (OBJECT I-M-TOKEN.491))
          (PRECOND I-M-READY-TO-RECEIVE.22.536
              (OBJECT I-M-TOKEN.491))
          (MESSAGE I-M-TOKEN.491))
         (2 I-M-TIME-OUT-EVENT.144.507
          (PRECOND I-M-READY-TO-RECEIVE.36.495
              (RECIPIENT I-M-SYSTEM-SM)
              (OBJECT I-M-TOKEN.491))
          (ACTION I-M-TIME-OUT.40.499 (ACTOR I-M-SYSTEM-SM)
              (RECIPIENT I-M-SYSTEM-SM)
              (OBJECT I-M-TOKEN.491))
          (RESULT I-M-TRANSMISSION-TIMED-OUT.41.502
              (RECIPIENT I-M-SYSTEM-SM)
              (OBJECT I-M-TOKEN.491))
          (RECIPIENT I-M-SYSTEM-SM) (MESSAGE I-M-TOKEN.491))))))
(CORRECTION I-M-BYPASS-NODE.1177
  (SEQUENCE-OF-STEPS I-M-CORRECTION-AND-GROUP.251.1164
    (1 I-M-CORRECTION-PROC.250.1163
     (SEQUENCE-OF-STEPS I-M-AND-GROUP.249.1162
       (1 I-M-LOGICAL-DISCONNECT-EVENT.248.1161
        (ACTOR I-M-SYSTEM-SM) (OBJECT1 I-M-GENERIC-RC.422)
        (OBJECT2 I-M-CORE-NETWORK.471)
        (PRECOND I-M-LOGICALLY-CONNECTED.245.1155
            (OBJECT2 I-M-CORE-NETWORK.471))
        (ACTION I-M-LOGICAL-DISCONNECT.246.1158
            (ACTOR I-M-SYSTEM-SM)
            (OBJECT2 I-M-CORE-NETWORK.471))
        (RESULT I-M-LOGICALLY-DISCONNECTED.247.1160
            (OBJECT2 I-M-CORE-NETWORK.471))))))
  (PRECOND I-M-PHYSICALLY-CONNECTED.202.1167
      (OBJECT1 I-M-GENERIC-RC.422)
      (OBJECT2 I-M-CORE-NETWORK.471))
  (PRECOND I-M-LOGICALLY-CONNECTED.203.1170
      (OBJECT1 I-M-GENERIC-RC.422)
      (OBJECT2 I-M-CORE-NETWORK.471))
  (FRAMEWORK I-M-SHORT-TERM.610)
  (CONFIGURATION I-M-CORE-NETWORK.471)
  (CORRECTION-COMPONENT I-M-SYSTEM-SM)
  (RESULT I-M-OPERATIONAL.302.1175 (OBJECT I-M-CORE-NETWORK.471))
```

```
   (FAILED-COMPONENT I-M-GENERIC-RC.422))
(CORRECTION I-M-UNDEFINED-CORRECTION.1238
   (FRAMEWORK I-M-LONG-TERM.699)
   (FAILED-COMPONENT I-M-GENERIC-RC.422)
   (SEQUENCE-OF-STEPS I-M-CORRECTION-AND-GROUP.316.1237
      (1 I-M-CORRECTION-PROC.315.1236
       (SEQUENCE-OF-STEPS I-M-AND-GROUP.314.1235
          (1 I-M-AND-GROUP.313.1234
           (2 I-M-VERIFY-CONNECTORS-EVENT.311.793
            (ACTION I-M-VERIFY-CONNECTORS.84.790
                  (ACTOR I-M-CREW.737)
                  (OBJECT I-M-CONNECTORS.310))
            (ACTOR I-M-CREW.737) (OBJECT I-M-CONNECTORS.310))
           (1 I-M-VERIFY-POWER-EVENT.309.802
            (ACTION I-M-VERIFY-POWER.81.799 (ACTOR I-M-CREW.737)
                  (OBJECT I-M-POWER-SYSTEM.308))
            (ACTOR I-M-CREW.737) (OBJECT I-M-POWER-SYSTEM.308))
           (3 I-M-REPLACE-WITH-SPARE.312.1229
            (RESULT I-M-LOGICALLY-CONNECTED.300.1193
                  (OBJECT1 I-M-CORE-NETWORK.471))
            (RESULT I-M-OPERATIONAL.302.1175
                  (OBJECT I-M-CORE-NETWORK.471))
            (PRECOND I-M-NON-OPERATIONAL.286.807
                  (OBJECT I-M-GENERIC-RC.422))
            (SEQUENCE-OF-STEPS I-M-CORRECTION-AND-GROUP.299.1226
               (1 I-M-CORRECTION-PROC.298.1225
                (SEQUENCE-OF-STEPS I-M-AND-GROUP.297.1224
                   (4 I-M-LOGICAL-CONNECT-EVENT.296.1199
                    (OBJECT1 I-M-CORE-NETWORK.471))
                   (2 I-M-PHYSICAL-CONNECT-EVENT.288.1207
                    (RESULT I-M-PHYSICALLY-CONNECTED.73.1201
                          (OBJECT1 I-M-CORE-NETWORK.471))
                    (ACTION I-M-PHYSICAL-CONNECT.72.1203
                          (OBJECT1 I-M-CORE-NETWORK.471))
                    (PRECOND I-M-PHYSICALLY-DISCONNECTED.71.1205
                           (OBJECT1 I-M-CORE-NETWORK.471))
                    (OBJECT1 I-M-CORE-NETWORK.471))
                   (1 I-M-PHYSICAL-DISCONNECT-EVENT.287.815
                    (RESULT I-M-PHYSICALLY-DISCONNECTED.67.809
                          (OBJECT1 I-M-GENERIC-RC.422))
                    (ACTION I-M-PHYSICAL-DISCONNECT.66.811
                          (OBJECT1 I-M-GENERIC-RC.422))
                    (PRECOND I-M-PHYSICALLY-CONNECTED.65.813
                           (OBJECT1 I-M-GENERIC-RC.422))
                    (OBJECT1 I-M-GENERIC-RC.422))
                   (3 I-M-LOGICAL-DISCONNECT-EVENT.292.857
                    (RESULT I-M-LOGICALLY-DISCONNECTED.291.851
                          (OBJECT1 I-M-GENERIC-RC.422))
                    (ACTION I-M-LOGICAL-DISCONNECT.290.853
                          (OBJECT1 I-M-GENERIC-RC.422))
                    (PRECOND I-M-LOGICALLY-CONNECTED.289.855
                           (OBJECT1 I-M-GENERIC-RC.422))
                    (OBJECT1 I-M-GENERIC-RC.422)))))
            (RESULT I-M-LOGICALLY-DISCONNECTED.301.822
                  (OBJECT1 I-M-GENERIC-RC.422))))))))
```

```
(EFFECT I-M-FAILURE-EFFECT.1267
      (SEQUENCE-OF-STEPS I-M-BYPASS-NODE.1177
         (SEQUENCE-OF-STEPS I-M-CORRECTION-AND-GROUP.251.1164
            (1 I-M-CORRECTION-PROC.250.1163
             (SEQUENCE-OF-STEPS I-M-AND-GROUP.249.1162
                (1 I-M-LOGICAL-DISCONNECT-EVENT.248.1161
                 (ACTOR I-M-SYSTEM-SM)
                 (OBJECT1 I-M-GENERIC-RC.422)
                 (OBJECT2 I-M-CORE-NETWORK.471)
                 (PRECOND I-M-LOGICALLY-CONNECTED.245.1155
                       (OBJECT2 I-M-CORE-NETWORK.471))
                 (ACTION I-M-LOGICAL-DISCONNECT.246.1158
                       (ACTOR I-M-SYSTEM-SM)
                       (OBJECT2 I-M-CORE-NETWORK.471))
                 (RESULT I-M-LOGICALLY-DISCONNECTED.247.1160
                       (OBJECT2 I-M-CORE-NETWORK.471))))))
         (PRECOND I-M-PHYSICALLY-CONNECTED.202.1167
                (OBJECT1 I-M-GENERIC-RC.422)
                (OBJECT2 I-M-CORE-NETWORK.471))
         (PRECOND I-M-LOGICALLY-CONNECTED.203.1170
                (OBJECT1 I-M-GENERIC-RC.422)
                (OBJECT2 I-M-CORE-NETWORK.471))
         (FRAMEWORK I-M-SHORT-TERM.610)
         (CONFIGURATION I-M-CORE-NETWORK.471)
         (CORRECTION-COMPONENT I-M-SYSTEM-SM)
         (RESULT I-M-OPERATIONAL.302.1175
               (OBJECT I-M-CORE-NETWORK.471))
         (FAILED-COMPONENT I-M-GENERIC-RC.422))
      (EFFECTED-COMPONENT I-M-CREW.737))
(EFFECT I-M-FAILURE-EFFECT.1273
      (EFFECTED-COMPONENT I-M-MISSION-SUPPORT.1009)
      (SEQUENCE-OF-STEPS I-M-PROCEDURE.1272
         (SEQUENCE-OF-STEPS I-M-NULL-AND-GROUP.86
            (1 I-M-NULL-EVENT.85 (ACTION I-M-NULL-ACTION)))))
(EFFECT I-M-FAILURE-EFFECT.1282
      (EFFECTED-COMPONENT I-M-SYSTEMS.480)
      (SEQUENCE-OF-STEPS I-M-PROCEDURE.1272
         (SEQUENCE-OF-STEPS I-M-NULL-AND-GROUP.86
            (1 I-M-NULL-EVENT.85 (ACTION I-M-NULL-ACTION)))))
(EFFECT I-M-FAILURE-EFFECT.1288
      (EFFECTED-COMPONENT I-M-INTERFACES.1042)
      (SEQUENCE-OF-STEPS I-M-PROCEDURE.1272
         (SEQUENCE-OF-STEPS I-M-NULL-AND-GROUP.86
            (1 I-M-NULL-EVENT.85 (ACTION I-M-NULL-ACTION))))))
```

5.3 Parsing Input Questions

In FANSYS, the parser operates in two modes called case mode and question mode. The parser behaves identically in each mode with the exception of the degree of indexing that occurs in each. In case mode, every instance of a concept that is created in memory is fully indexed in the generalization indexing hierarchy (in addition to each of the other hierarchies), as described in the section on Memory Creation and Retrieval. Thus, every new instance of a concept will be generalized with other similar instances. This reflects the fact that during case parsing, all

information is useful and should be used to learn about the domain. Since it is the task of the parser to learn from the case texts that it reads, it is desirable that full indexing occur.

In question mode, instances are not generalized. This reflects the different goal that FANSYS has in parsing a question. In case mode, the goal is to learn new concepts and organize them within memory. In question mode, the goal is to formulate a search query and return the results of that query. Questions that are parsed instantiate concepts in memory just as in case mode, but these concepts need not be generalized over, since they are not part of the actual case text. The degree of indexing is the primary difference between the two modes.

It is possible to specify that certain index patterns be active in only either question mode or in case mode. The motivation behind this involves the processing that occurs when a question is parsed. If parsing a manual and rhetorical question is encountered, the system should not activate the search and retrieval patterns that handle that question, since it is not an appropriate context for search and retrieval. In question mode, on the other hand, the system should activate search and retrieval functions if a question is parsed. This is facilitated by making the question patterns with functional processing knowledge operable in only question mode, rather than case mode. In general however, an index pattern will be active in both modes; the phrase "ring concentrator" should certainly activate the same concept in either mode.

Although this dichotomy between the two modes exists, it should be emphasized that processing is identical in all other ways. The flow of control is the same, as are those hierarchies other than the generalization indexing hierarchy, since the same memory structures are parsed into in both cases.

5.3.1 Question Patterns

In parsing questions, there are two primary indexed patterns used. These are shown in figure 5.5, along with sample questions. The parts of the patterns in square brackets are optional. Questions are highly constrained in the format in which they may be posed, primarily for ease of definition, and to provide a well specified means of generating a query. In principle, many different question patterns could be defined to handle the multitudinous ways in which questions are posed by people; this would simply involve adding more lexical patterns to the system to handle such cases. As it stands, the format for queries has been chosen to be similar to the format used by the User Interface point and click mechanism.

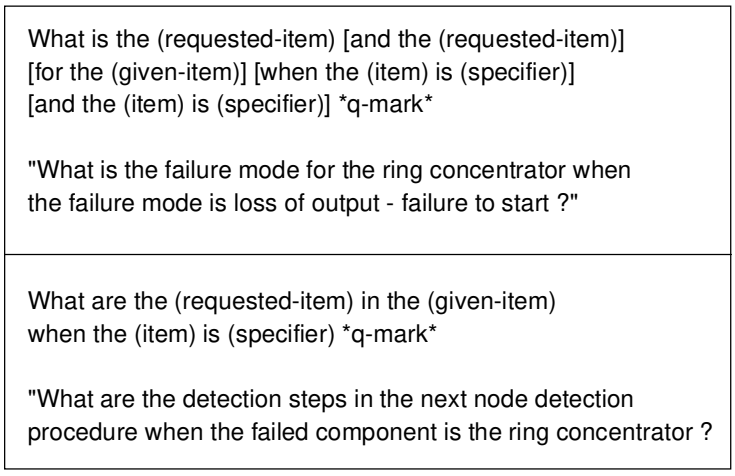

Figure 5.5
The Two Question Formats

Once a question MOP is instantiated and activated by recognizing one of the two question patterns, the information in the slots is extracted by an associated function attached to the question MOP. The memory search functions detailed in the section on Memory Creation and Retrieval are then invoked by the question MOP, and control is relinquished to the search module, which performs the search and generates the result.

5.3.2 An Annotated Question Example

The process of parsing a question will now be illustrated by an annotated trace. This trace is heavily edited; it is assumed that the reader is familiar with the parsing mechanism from the earlier trace of the parsing of case RC.1. Only those features specific to the processing of a question will be highlighted.

```
Parsing
   (WHAT IS THE FAILURE CAUSE FOR THE TIME GENERATION UNIT *Q-MARK*).

Reading WHAT
  (WHAT * IS THE (REQUESTED M-QUESTION-CONSTRAINT) AND THE
     (REQUESTED M-QUESTION-CONSTRAINT) FOR THE (GIVEN M-QUESTION-CONSTRAINT)
     WHEN THE (STATE M-STATE-ASSERTION) AND THE (STATE M-STATE-ASSERTION)
     *Q-MARK*) = M-QUESTION
  (WHAT * ARE THE (REQUESTED M-AND-GROUP) IN THE
     (GIVEN M-QUESTION-CONSTRAINT) WHEN THE (STATE M-STATE-ASSERTION)
     *Q-MARK*) = M-QUESTION2
```

As was discussed earlier, there are two query formats. These are stored with the question word "what", and both become active upon parsing that word.

```
Reading IS
  (WHAT IS * THE (REQUESTED M-QUESTION-CONSTRAINT) AND THE
     (REQUESTED M-QUESTION-CONSTRAINT) FOR THE (GIVEN M-QUESTION-CONSTRAINT)
     WHEN THE (STATE M-STATE-ASSERTION) AND THE (STATE M-STATE-ASSERTION)
     *Q-MARK*) = M-QUESTION
```

```
Reading THE
  (WHAT IS THE * (REQUESTED M-QUESTION-CONSTRAINT) AND THE
    (REQUESTED M-QUESTION-CONSTRAINT) FOR THE (GIVEN M-QUESTION-CONSTRAINT)
    WHEN THE (STATE M-STATE-ASSERTION) AND THE (STATE M-STATE-ASSERTION)
    *Q-MARK*) = M-QUESTION

Reading FAILURE

Reading CAUSE
  (FAILURE CAUSE *) = M-FAILURE-CAUSE referenced
    Specializing: I-M-FAILURE-CAUSE.4416
  (WHAT IS THE (REQUESTED M-QUESTION-CONSTRAINT) * AND THE
    (REQUESTED M-QUESTION-CONSTRAINT) FOR THE (GIVEN M-QUESTION-CONSTRAINT)
    WHEN THE (STATE M-STATE-ASSERTION) AND THE (STATE M-STATE-ASSERTION)
    *Q-MARK*) = M-QUESTION
```

The MOP class M-QUESTION-CONSTRAINT organizes two types of MOPs, systems and procedures. These are the only types of knowledge representation that are allowed to be slots within a question MOP. Since a failure cause is a type of procedure, it satisfies the requirement of the "requested" slot, and the pattern can be advanced. Note that since the parser is in question mode while parsing this question, the MOP I-M-FAILURE-CAUSE.4416 is not indexed in the generalization indexing hierarchy. This prevents a content-less MOP like this one from being used to draw generalizations from text, particularly since that text has no relevance to describing faults in the domain.

```
Reading FOR
  (WHAT IS THE (REQUESTED I-M-FAILURE-CAUSE.4416) AND THE
    (REQUESTED M-QUESTION-CONSTRAINT) FOR * THE
    GIVEN M-QUESTION-CONSTRAINT)WHEN THE (STATE M-STATE-ASSERTION)
    AND THE (STATE M-STATE-ASSERTION) *Q-MARK*) = M-QUESTION
```

Since the "and the..." part of the phrase is optional, the pattern can still be advanced past the point of the word "for".

```
Reading THE
  (WHAT IS THE (REQUESTED I-M-FAILURE-CAUSE.4416) AND THE
    (REQUESTED M-QUESTION-CONSTRAINT) FOR THE * (GIVEN M-QUESTION-CONSTRAINT)
    WHEN THE (STATE M-STATE-ASSERTION) AND THE (STATE M-STATE-ASSERTION)
    *Q-MARK*) = M-QUESTION

Reading TIME
   (TIME * GENERATION UNIT) = M-GENERIC-TGU

Reading GENERATION
  (TIME GENERATION * UNIT) = M-GENERIC-TGU

Reading UNIT
  (TIME GENERATION UNIT *) = M-GENERIC-TGU referenced
    Specializing: I-M-GENERIC-TGU.3723
```

```
(WHAT IS THE (REQUESTED I-M-FAILURE-CAUSE.4416) AND THE
  (REQUESTED M-QUESTION-CONSTRAINT) FOR THE (GIVEN M-QUESTION-CONSTRAINT)
  * WHEN THE (STATE M-STATE-ASSERTION) AND THE (STATE M-STATE-ASSERTION)
  *Q-MARK*) = M-QUESTION
```

As noted before, systems fulfill the M-QUESTION-CONSTRAINT filler constraint. The time generation unit thus works as a filler for the "given" slot.

```
Reading *Q-MARK*
  (WHAT IS THE (REQUESTED I-M-FAILURE-CAUSE.4416) AND THE
    (REQUESTED M-QUESTION-CONSTRAINT) FOR THE (GIVEN I-M-GENERIC-TGU.3723)
    WHEN THE (STATE M-STATE-ASSERTION) AND THE (STATE M-STATE-ASSERTION)
    *Q-MARK* *) = M-QUESTION referenced
    Specializing: I-M-QUESTION.4421
```

The state assertions are also optional elements of the question phrase, so the pattern can be advanced and recognized. Question MOPs have functional code associated with them that interface to the Memory Creation and Retrieval Module. Once the question MOP is recognized, the slots are passed down to Memory Search and Retrieval, where the query is created and processed.

```
Performing Search:

[1] ERRONEOUS INPUT
[2] PIECE-PART FAILURES
[3] CONTAMINATION
[4] TEMPERATURE ( HIGH OR LOW )
[5] MECHANICAL SHOCK
[6] THERMAL SHOCK

Found good match. Elaborate further anyway? (Y/n) n
```

Once the user has specified that she or he is satisfied with the results of the query, control is returned to the parser, which in this case cleans up any index patterns that were not resolved during the parse.

```
Removing: I-M-IP.4385.4410
Removing: I-M-IP.479.4418
Removing: I-M-IP.481.4417
Removing: I-M-IP.479.4414
Removing: I-M-IP.481.4413
Removing: I-M-IP.4384.4411
```

5.4 Generating Answers to Input Questions

Just as the parser maps text into the conceptual structures that underlie it, some mechanism is needed to map memory structures back into natural language. This is the task of the generator. The generator relies on the same stereotypical patterns of language usage that the parser relies on, since most of the same index patterns that are used to parse text are used to generate back into text.

The algorithm for generation is as follows:

1. Choose a MOP to generate.
2. Find the first index pattern with that MOP as its target that satisfies any imposed constraints (such as syntactical constraints).
3. If no such pattern exists, look for a pattern up that MOPs abstraction hierarchy.
4. A. If no pattern is found, simply return the MOP name (e.g. words return the word name).
   B. Otherwise, for each element of the pattern, recursively apply this generation algorithm.

As an example, consider the generation of the MOP I-M-TIME-OUT-EVENT.144.507. A portion of the indexing hierarchies characterizing this MOP is shown in figure 5.6. The following is a trace by the generation function which details the execution of the algorithm just described:

```
>(generate I-M-TIME-OUT-EVENT.144.507)
 Checking mop I-M-TIME-OUT-EVENT.144.507 for phrases:
   Found: NIL
 Checking mop M-TIME-OUT-EVENT.144 for phrases:
   Found: NIL
 Checking mop M-TIME-OUT-EVENT for phrases:
   Found: (M-IP.358)
 Considering phrase M-IP.358:
         ((1 (M-SYNTAX-CATEGORY RECIPIENT M-SYSTEM)) (2 WILL)
          (3 REACH) (4 A) (5 TIME-OUT) (6 LIMIT) (7 FOR) (8 RECEIPT) (9 OF)
          (10 THE) (11(M-SYNTAX-CATEGORY MESSAGE M-MESSAGE))
          (12 *PERIOD*))
 Using phrase: M-IP.358
 Checking mop I-M-SYSTEM-SM for phrases:
   Found: NIL
 Checking mop M-SM for phrases:
   Found: (M-IP.338)
 Considering phrase M-IP.338:
         ((1 SYSTEM) (2 MANAGEMENT))
 Using phrase: M-IP.338
 Checking mop I-M-TOKEN.491 for phrases:
   Found: NIL
 Checking mop M-TOKEN for phrases:
   Found: (M-IP.339)
 Considering phrase M-IP.339:
         ((1 NETWORK) (2 TOKEN))
 Using phrase: M-IP.339

("SYSTEM" "MANAGEMENT" "WILL" "REACH" "A" "TIME-OUT" "LIMIT" "FOR"
 "RECEIPT" "OF" "THE" "NETWORK" "TOKEN" ".")
```

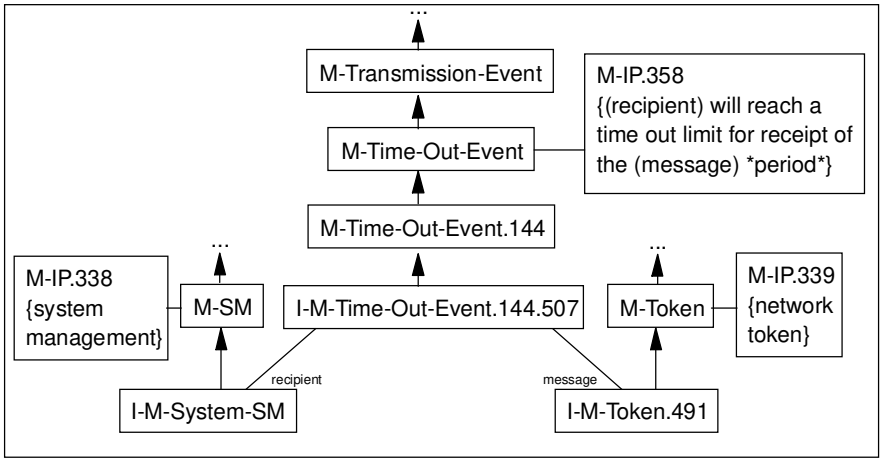

Figure 5.6
Sample Hierarchy for I-M-Time-Out-Event.144.507

One constraint on the generation of patterns is that of syntax. Patterns may be given membership within a syntactic category during pattern definition time using the syntax-category attribute shown previously in figure 5.2. Fillers for slots within patterns may also be syntactically constrained. In order to use a pattern for the generation of a slot filler, it has to be of the same syntactic category as that slot constraint. This allows the generator to generate in a syntactically correct manner, rather than just stringing together syntactically inconsistent phrases. For example, consider another phrase that might have M-TIME-OUT-EVENT as its target: "the timing out of the (system) for the (message)". This phrase might be assigned to a category like VERB-PHRASE. We would not want to use it to generate a complete sentence, since it wouldn't be proper grammatically, whereas the phrase in the annotated trace works fine for this purpose. These constraints can be controlled through the use of syntactic categories. In practice, the index patterns within FANSYS are syntactically unconstrained, since the text has been simple enough so as not to cause problems. Should the lexical domain become more richly enhanced, however, some form of syntactic constraint is needed. The syntactic constraints that can be placed upon a pattern provide for just such measures, should they become necessary.

Although in theory the exact same patterns could be used to generate output that are used to parse text, there are in practice some constraints placed on patterns in terms of whether they should be used only for generation, only for parsing, or both. Most of the patterns in the system are used for both parsing and generation, but a few are modified with :gen and :nogen in the pattern definitions detailed previously in figure 5.2. The :gen specifies that a pattern should be used for generation only, while a :nogen modifier specifies that a pattern should be used only to parse text. An example of a pattern that might be marked with a :nogen modifier is that which characterizes the sentence S1: "indication of a RC failure to start is first detected by the next active node on the network," which is a pointer to the M-NEXT-NODE-DETECTION procedure. (See the detection procedure section of the annotated parse trace for the actual index pattern which is used to characterize S1.) When generating the M-NEXT-NODE-DETECTION procedure, it is preferable to have as much information characterizing that procedure included within the generation; this most certainly includes the steps of the procedure. Generating from the pattern which characterizes S1 would not include that information; it would generate essentially the same sentence as S1. It is therefore marked with the :nogen specifier.

The index pattern that is actually used to generate any sort of procedure (including M-NEXT-NODE-DETECTION) is one similar to IP2: { (1 M-EVENT) (2 M-EVENT) ... } => M-PROCEDURE, where (1), (2), and so on are the actual events that compose the procedure. IP2 generates a list of sentences describing the actual procedure under consideration. This is too abstract of a pattern to be used in parsing procedures, however, since the construction of a procedure is typically more involved than just listing together events (there is some search and recursive construction of the procedure that occurs; see the annotated parse trace for some of the details). IP2 is therefore marked with the :gen modifier, which specifies that it be used only for generation.

Although the :nogen and :gen specifiers exist, they are in practice seldom used, since most of the patterns defined in the system are useful for both parsing and generation. Were a more sophisticated generation algorithm in place, one which considered the space of all available patterns and chose the pattern which best characterized the concept to be generated (rather than just choosing the first pattern it finds), then the :nogen and :gen modifiers could probably be abandoned. Since generation per se has been a secondary concern to the project, a simple generation algorithm with the :gen and :nogen functionality has been chosen to perform generation. Within this limited domain, this approach has proven sufficient for the generation tasks required by FANSYS.

6. Memory Creation and Retrieval

The ISA, Lexical, and Slot-Filler hierarchies described in sections 3 and 4 organize items hierarchically under predefined categories. For example, index patterns are lumped underneath the index pattern MOP, lexical entries are grouped underneath the lexicon MOP, and so forth. In addition, the ISA and Slot-Filler memory hierarchies index cases by only one path that does not contain generalizations across cases. While this organization is useful for the parsing process, it is not an organization which is appropriate for retrieving applicable cases during a question/answering session.

During question/answering FANSYS must be able to quickly retrieve either generalizations of its knowledge, or specific cases from a potentially large database of cases. Our solution to this problem in FANSYS is to overlay an additional indexing structure on top of the ISA memory hierarchy. This additional indexing structure is based upon the memory representation model used in Kolodner's CYRUS system (Kolodner, 1983a, 1983b, 1984). Collectively, these indices form a rich hierarchy of generalizations where many indices point to each case. Since this memory representation allows for indexing and the creation of generalizations, this hierarchy will be referred to as the Generalization-Indexing (GI) memory hierarchy. Although similar to the indices used by Kolodner in CYRUS, the GI indices employed in FANSYS differ in that they allows for attribute values to be represented hierarchically.

In addition to allowing for case retrieval, the GI memory hierarchy allows FANSYS to perform memory tasks similar to those performed by CYRUS, including: (1) generalization across cases, (2) encoding specificity* by grouping similar cases together, (3) fast retrieval by limiting memory traversal to relevant indices, and (4) elaboration strategies which dictate where in memory to search if the initial search query fails. Before discussing the details of the algorithms used in FANSYS for these tasks, the underlying data structures and the method in which comparisons across cases are made must be discussed.

6.1 MOP Attributes

As in the other hierarchies, the fundamental data structure used in the GI memory hierarchy is also the Memory Organization Packet (MOP). To recap, a MOP is a unit which represents a particular concept. By grouping MOPs hierarchically, high-level MOPs represent abstractions of the MOPs below it. There are two types of MOP's: the MOPs located at the bottom of the memory hierarchy are *instances* (denoted I-M-MOPNAME), while the MOPs which organize the instances are *abstractions* (denoted M-MOPNAME).

While the same MOP structure is used in all of the memory hierarchies, the GI memory hierarchy requires the use of several different constructs. The bad-indices of the MOP indicate attributes which do not make good indices. The elaborations of the MOP indicate other MOPs which may aid in memory retrieval. The role of both of these attributes will be discussed in the sections of memory creation and retrieval. The slot of the MOP contains the conceptual definition

---

* Encoding specificity (Tulving, 1972) refers to the psychological phenomenon of memory case retrieval based upon the features present in the original processing of the case. Case retrieval can only occur if these features have been discriminated.

of the MOP. The norm of the MOP contains generalized information about it's sub-MOPs (specializations). Finally, the indices of the MOP contain attribute/value pairs which point to its sub-MOP specializations. In FANSYS, attributes are atomic but values may be defined hierarchically.

Attributes, slots, and the values of indices all use the same list-based tree structure. This representation employs lists which may contain other lists, which may in turn contain other lists. By linking these lists together, detailed tree-like structures can be created. The general format for these lists is *((role filler) (role filler) ...)*, where the role is an atomic attribute, and the filler is either an atomic element or recursively defined as another list of the same structure.

To illustrate this format, consider the following example taken from the failure detection procedure for the ring concentrator. In this procedure, a faulty ring concentrator is detected when the next ring concentrator in the network fails to receive a token from the previous ring concentrator. As a precondition for this procedure, the next ring concentrator must be ready to receive a token from the previous ring concentrator. This is represented below as the READY-TO-RECEIVE event, where the RECIPIENT is the next ring concentrator (I-M-NEXT-RC), the SENDER is the current ring concentrator (I-M-RC), and the object being sent is a token (I-M-TOKEN):

```
(PRECOND    ((HEADER I-M-READY-TO-RECEIVE)
             (SENDER
              ((HEADER I-M-RC)
               (SOFTWARE
                    ((HEADER M-SM)
                     (SOFTWARE M-SOFTWARE-COMPONENT)))
                    (HARDWARE
                     ((HEADER M-TRANSMITTER)
                      (SOFTWARE M-SOFTWARE-COMPONENT)
                      (HARDWARE M-HARDWARE-COMPONENT)))))
             (RECIPIENT
              ((HEADER I-M-NEXT-RC)
               (SOFTWARE
                    ((HEADER M-SM)
                     (SOFTWARE M-SOFTWARE-COMPONENT)))
                     (HARDWARE
                    ((HEADER M-TRANSMITTER)
                     (SOFTWARE M-SOFTWARE-COMPONENT)
                     (HARDWARE M-HARDWARE-COMPONENT)))))
             (OBJECT I-M-TOKEN)))
```

In this example, the top-level list is of the form:

((PRECOND (SENDER (...)) (RECIPIENT (...) (OBJECT I-M-TOKEN)))

The information contained within the ellipses is defined with an identical structure. In this particular example, the SENDER and RECIPIENT are ring concentrators which in turn have two sub-components, a software and a hardware component. The software component is the system management (M-SM), while the hardware component is a transmitter (M-TRANSMITTER). Notice that these components are defined recursively, in that they have their own hardware and software components. If finer detail is required, more components can be added and each component can be further defined in terms of sub-components.

This representation scheme allows FANSYS to accurately represent a variety of devices, actions, events, procedures, or concepts in terms of aggregates or semantic meaning. The slots, norms, and indices of the GI memory hierarchy all use this same format. For instance, if the previous example is a slot, then the conceptual content of the MOP would be the ready-to-receive event for a ring concentrator. If the example is a norm, then a generalization of the MOP would be the ready-to-receive event, and sub-MOPs would represent specializations of ready-to-receive. Finally, if the examples was used in an index, then the ready-to-receive event would serve as the link between another MOP concept.

6.2 Linking MOPs

As described above, the values of indices which link MOPs take the form of detailed lists, where the lists may represent hierarchical concepts. Similarly, the method in which MOPs are linked is also hierarchical. Just like the ISA memory hierarchy, the GI memory hierarchy is also organized in terms of abstractions and instances. Abstract MOPs occupy the internal nodes of the tree, while specific instances occupy the leaves. However, unlike the ISA hierarchy, the GI hierarchy is much more richly indexed. In the GI hierarchy, each instance will be indexed by all possible features of that instance. This results in multiple retrieval paths and "deep learning" of each case.

Indices are internally represented within each MOP by the form: *((Attribute Value Next-MOP) (Attribute Value Next-MOP) ...)*, where Attribute is an atomic feature, Value is a list of the form described in the previous example, and Next-MOP is the name of the MOP which is linked by the index. Using this representation, indices are uni-directional and always point downwards in the hierarchy towards more specialized MOPs.

The rule for linking MOPs together is to use index values which differ from the norms. Cases which are similar to one another will be grouped close together, but these cases will be indexed in the manner which they differ from each other. This indexing strategy allows distinguishing features to be the factor which determines which cases are retrieved. The actual algorithm used to select these indices will be presented in the section on memory creation.

As an example, consider the MOP M-CASE. This MOP organizes all cases stored in memory. Any norms which this MOP may contain apply to all cases in memory. MOPs which are indexed directly below M-CASE are a bit more specialized; MOPs indexed below these MOPs are even more specialized, and so on, until we finally reach the instances at the leaves of the hierarchy. Each MOP contains norms which are the generalizations of all MOPs below it. Furthermore, each MOP is indexed by the way in which it is different from other MOPs. If the

MOPs had the configuration of indices shown below they would encode the sample hierarchy shown graphically in figure 6.1. To keep this example simple only a few indices have been specified. The actual implementation contains many more indices.

M-CASE indices:
((FAILED-COMPONENT ((HEADER I-M-RC) ...) M-MOP.1)
(FAILED-COMPONENT ((HEADER I-M-GW) ...) M-MOP.2)
(FAILURE-CAUSE ((HEADER I-M-THERMAL-SHOCK)...) I-M-MOP.3))

M-MOP.1 indices:
((FAILURE-CAUSE ((HEADER I-M-THERMAL-SHOCK)..) I-M-MOP.4)
(FAILURE-CAUSE ((HEADER I-M-CONTAMINATION)..) I-M-MOP.5)

M-MOP.2 indices:
((FAILURE-MODE ((HEADER I-M-ERRONEOUS-OUTPUT)..) I-M-MOP.6))

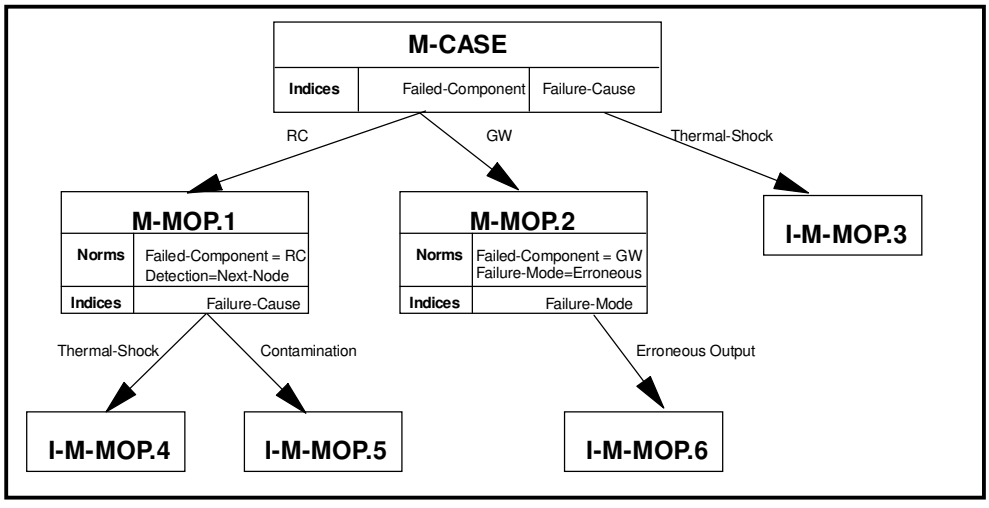

Figure 6.1
Sample GI Memory Hierarchy

In this example, the attribute of FAILED-COMPONENT may take on two different values. MOP.1 is linked to M-CASE if the value of the FAILED-COMPONENT is ring concentrator. MOP.2 is linked to M-CASE if the value of the FAILED-COMPONENT is gateway. I-MOP.3 is linked to M-CASE if the value of the FAILURE-CAUSE is thermal shock. In this example, the ellipses denote that additional details may be specified for the attribute's values, as in the previous example (to specify the hardware components, software components, etc.). Just like the MOP M-CASE, MOPs 1 and 2 have indices of their own pointing to more specialized MOPs. The values of the indices indicate the ways in which MOPs differ from one another. MOPs 1,4, and 5 all involve the failure-component of a ring concentrator. I-MOP's 4 and 5 are further differentiated by the failure-cause. I-MOP 4 is indexed by thermal shock, while I-MOP 5 is indexed by contamination. Finally, generalizations are stored at all sub-MOPs. In this example, I-MOPs 4 and 5 share the failed-device and the failure cause. This generalization is stored as the norms of their common abstraction, MOP 1. Similarly, the norms of MOP 2 contain the generalizations of all sub-MOPs, in this example I-MOP 6.

Unlike the ISA hierarchy, each index in the GI hierarchy has an attribute and a value. As a result, when MOPs are accessed by traversing these indices, we are now accessing MOPs by their meaning. This will greatly speed up the retrieval process by limiting memory traversal to only those indices which correspond to the memory query. Consequently, search will be constrained to only relevant portions of memory. Kolodner describes the attribute/value indices as "locks". In order to access a structure, we must have the "key." In this case, the key corresponds to the value of an index. As more keys become available, more locks can be opened. Analogously, as more features are provided to the system, the greater the probability of accessing a relevant case increases since more indices can be traversed. Kolodner's notes that this behavior mirrors Tulving's encoding specificity hypothesis, as well as several other psychological results (Kolodner, 1983).

It is important to remember that the GI memory hierarchy is not completely separate from the ISA memory hierarchies. Instead, the GI hierarchy is "overlaid" on top of the ISA memory structures. Both the GI hierarchy and the ISA hierarchies share the same root MOPs and the same instance MOPs. The two hierarchies differ in the way the root MOP is connected to the instances. The GI hierarchy uses a rich set of indices, while the ISA hierarchy uses only enumerated indices. This integration is illustrated in figure 3.5 in the system architecture section. An area of future work is to merge these two hierarchies into a single hierarchy which combines features of both memory schemes.

6.3 Comparing Cases

Before discussing the algorithms used for memory creation and retrieval, an important operation which must be understood is how concepts are compared. All slots, indices, and norms are stored in the form of lists specified in the section on MOP attributes. When creating generalizations, adding new MOPs to memory, or determining which indices can be traversed, the comparison operation must be invoked. This operation is made more complicated due to the hierarchical representations of concepts. For example, a standard data processor running a particular program might be represented as ((HEADER I-M-SDP) (SOFTWARE I-M-PROGRAM.1)), while a standard data processor running a different program might be represented as ((HEADER I-M-SDP) (SOFTWARE I-M-PROGRAM.2)). If strict equality were used to compare these two representations, we would end up with the result that the components are not equal. However, both components are standard data processors. A more useful comparison operation would return an answer which indicates how well one item matches with another.

The comparison function implemented in FANSYS returns more meaningful information by computing a percentage of match between two concepts. The algorithm considers which values to compare in a top-down, depth-first manner and then propagates the percent comparison for each value in a bottom-up fashion. An overview of the COMPARE algorithm is described below.

```
Given a list of attribute/value pairs from a target T,
and a query Q, start at the topmost level:
(0) Set M=0
(1) For each attribute/value pair in Q do:
        If T contains the same attribute then
                If T's value = Q's value set M=M+1
                else if T's value is a list
                        Set M=M+ (recurse to step 1 using
                                        the values of Q and T)
(2) Return M/(Number of attribute/value pairs in Q)
```

An example of comparing two cases is given below. The target case is a subset of the example described in the previous section on MOP attributes. The query is similar to the target except some of the slots do not exactly match. A sender is specified in the query but is missing in the target, and one of the ring concentrator's software components within the query is different than the target.

```
Case= ((RECIPIENT
        ((HEADER I-M-NEXT-RC)
         (SOFTWARE
          ((HEADER M-SM)
           (SOFTWARE M-SOFTWARE-COMPONENT)))
         (HARDWARE
          ((HEADER M-TRANSMITTER)
           (SOFTWARE M-SOFTWARE-COMPONENT)
           (HARDWARE M-HARDWARE-COMPONENT)))))
      (OBJECT I-M-TOKEN))

Query= ((RECIPIENT
          ((HEADER I-M-NEXT-RC)
          (SOFTWARE
            ((HEADER M-SM)
             (SOFTWARE M-SOFTWARE-COMPONENT)))
          (HARDWARE
            ((HEADER M-TRANSMITTER)
             (SOFTWARE M-PROGRAM-2)
             (HARDWARE M-HARDWARE-COMPONENT)))))
       (SENDER M-RC)
       (OBJECT I-M-TOKEN))
```

The case and query are depicted graphically in figures 6.2 and 6.3.

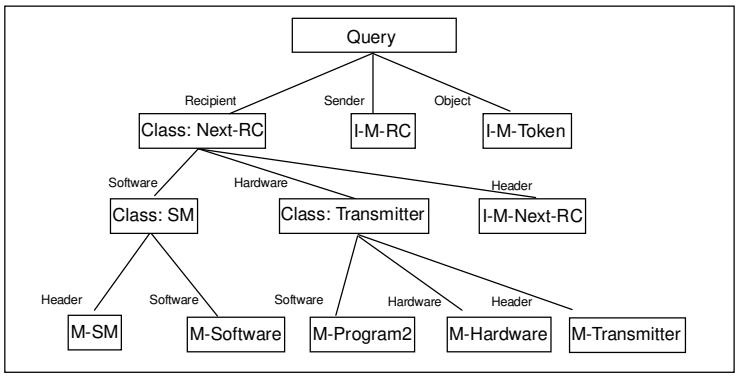

Figure 6.2
Graphical representation of sample query

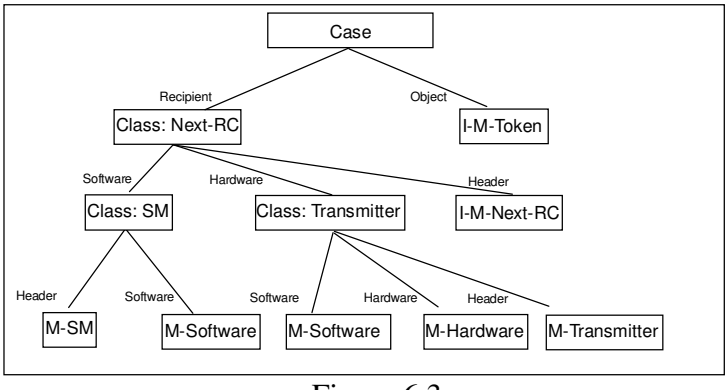

Figure 6.3
Graphical representation of sample case

The top-down stage of the algorithm determines which attributes should be compared. At the top-most level, the query consists of ((RECIPIENT ...) (SENDER ...) (OBJECT ..)). The attributes are RECIPIENT, SENDER, and OBJECT. The algorithm loops through each of these attributes and checks whether or not the case also contains the same attribute. In this example, RECIPIENT is contained within the target case. However, the value does not exactly match the query, and it is a list, so M will be set to M+X, where X is some value determined recursively via Compare((HEADER ..) (SOFTWARE ..) (HARDWARE..)). The next attribute is SENDER. However, the target contains no sender attribute, and so nothing is done. The last attribute is OBJECT. The target does have an OBJECT attribute, and the value matches the query exactly, so M is set to M+1. When the algorithm is finished, M=X+1, and the ratio (X+1)/3 will be returned. Values for the entire computation of the query are shown below, starting from the bottom-up. 1's indicate matches, and ?'s indicate values which haven't been computed yet.

```
((RECIPIENT  ?
        ((HEADER 1)
       (SOFTWARE ?
        ((HEADER 1)
         (SOFTWARE 1)))
       (HARDWARE ?)
        ((HEADER 1)
         (SOFTWARE 0)
         (HARDWARE 1)))))
 (SENDER 0)
 (OBJECT 1))
```

Percentages propagate up from the leaves:

```
((RECIPIENT
        ((HEADER 1)
       (SOFTWARE 2/2)
        ((HEADER 1)
         (SOFTWARE 1)))
       (HARDWARE 2/3)
        ((HEADER 1)
         (SOFTWARE 0)
         (HARDWARE 1)))))
 (SENDER 0)
 (OBJECT 1))
```

These values propagate up again:

```
((RECIPIENT
        ((HEADER 1)
       (SOFTWARE 1)
       (HARDWARE 2/3)))
 (SENDER 0)
 (OBJECT 1))
```

The final match percentage is given as: (((1+1+2/3)/3) + 0 + 1) / 3 = 0.63

The backed-up values are depicted on the graphical representation of the query shown below in figure 6.4. The horizontal slashed line for "Sender" indicates a comparison which is never made since there is no Sender slot in the case.

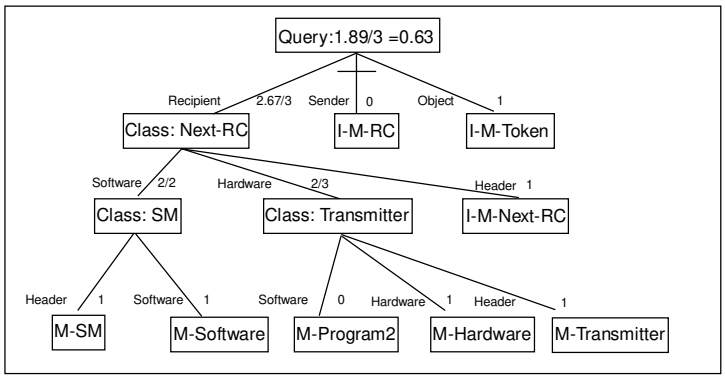

Figure 6.4

Comparing case from figure 6.3 against query from figure 6.2. Numerical values on the right indicate backed-up values. The horizontal line indicates a comparison which is never made since the attributes do not match.

Notice that this process compares the case against the query. If the query is compared against the case, then a different match percentage would be returned.

The percentage returned by the comparison function indicates that the case matches the query with 63% accuracy. While this number indicates how well features of the cases match, it doesn't include any measure of which features are more relevant than others. This problem requires additional semantic knowledge not directly addressed in the current implementation of FANSYS.

An algorithm similar to the one used to compare cases is also used to determine the similarities and the differences between cases. Determining similarities and differences are important in the memory creation process to create new generalizations and new indices.

Computing the similarities between cases uses almost the exact same algorithm used to compare cases, except equivalent values are saved as the algorithm runs. Computing the similarities between the case and query of figures 6.2 and 6.3 yields:

```
Sim = ((RECIPIENT
        ((HEADER I-M-NEXT-RC)
        (SOFTWARE
          ((HEADER M-SM)
           (SOFTWARE M-SOFTWARE-COMPONENT)))
        (HARDWARE
          ((HEADER M-TRANSMITTER)
           (HARDWARE M-HARDWARE-COMPONENT)))))
      (OBJECT I-M-TOKEN))
```

Any attributes or values which are not present in both cases is simply removed from the end result. Finding similarities will be used in determining the norms of MOPs so that generalizations may be created.

Finally, computing the differences between cases also follows a similar algorithm to the comparison algorithm. However, when values of the query and case do not match, the case values are stored and eventually returned. Just like the comparison algorithm, computing the differences of case 1 vs. case 2 is different from computing the differences of case 2 vs. case 1. When two values aren't the same, what should be returned, the mismatch from case1 or the mismatch from case 2? The approach taken in FANSYS is to use the differences with respect the second case. Typically, the function to compute the differences will be called twice, once to get the differences for case 1 and again the get the differences for case 2.

In our example with the query and target case, computing the differences between the case and query with respect to the query returns mismatches from the query:

```
Dif = ((RECIPIENT
       (HARDWARE
         ((SOFTWARE M-PROGRAM2))))
       (SENDER M-RC))
```

Similarly, computing the differences between the query and the case with respect to the case returns mismatches from the case:

```
Dif = ((RECIPIENT
       (HARDWARE
         ((SOFTWARE M-PROGRAM2)))))
```

The attribute/value pair of (SENDER M-RC) is not included since this isn't part of the case, but part of the query.

6.4 Memory Creation Process

With the underlying data structures defined and the algorithms for comparison, computing differences, and computing similarities developed, these components can now be combined into the memory creation process. As described earlier, the GI memory hierarchy must be organized so that MOPs are indexed by their differences, similar cases are grouped together, and the norms of MOPs hold the generalizations of all cases below it. These tasks are discussed by Kolodner in her work on long-term memory (Kolodner, 1983).

To accomplish these tasks within FANSYS, the GI memory system must first be given a case to index. This case will be some type of instance with a set of slots and attribute/value pairs which describe the conceptual content of the case. These instances are generated as the parser processes some text. This means that each instance created by the parser will also be passed to the GI memory system for indexing. In addition to passing the GI memory system a case, the parser will also pass a MOP where indexing should begin. For instances of entire cases this would be M-CASE (the MOP which organizes all cases), for components this would be M-ORU, and so on.

Once a case and a MOP location to begin indexing has been determined, the process of memory creation can begin. First, the case is separated into its attribute/value pairs. At this point, we would ideally like to weed out any irrelevant attributes and retain only the distinctive attributes. This is the difficult feature-selection problem. In the current implementation of FANSYS, non-predictive features have been preselected and stored in the bad-index slot of the MOP. Any attributes which are marked as bad-indices are then discarded. For complete cases, the failure-mode, failed-component, failure-detection procedure, and the failure-correction procedure are all predictive attributes, but the failure-cause and the failure-effects are nonpredictive since virtually all cases share the same values for these attributes. By stripping the nonpredictive attributes, memory will be less cluttered and more meaningful. However, note that it is not crucial to remove these attributes; if they are not removed FANSYS will still function normally, but will have larger memory requirements. While the current implementation of FANSYS requires these attributes to be specified by the designers, future work includes learning nonpredictive attributes automatically through pre-processing or a real-time statistical analysis or clustering of the data.

After relevant indices have been selected, the similarities between the case and the norms of the current MOP are computing using the similarity function. The norms of the MOP are then set to these similarities. This guarantees that the norm will only hold generalized information which applies to all cases which have been processed by the current MOP.

The next step is to traverse any indices whose attribute and value match an attribute and value of the new case. To determine if the index matches an attribute, the previously defined comparison function must be invoked. If the index and the attribute's value match greater than some threshold value, then the index is traversed. In FANSYS, this threshold was arbitrarily set at 0.5. Further experiments are necessary to determine if other threshold values are more appropriate. In general, a low threshold value will make indices very easy to traverse. For example, a threshold of 0 would result in all indices being traversed if the values had anything at all in common, no matter how small. This will result in much generalization, but also more indices will be created leading to potential memory problems. A high threshold value will require a very close match and make indices harder to traverse. While fewer indices would be created, less generalization would be made between cases.

By only traversing matching indices, traversal is restricted to relevant indices. If no indices match, then a new index is created which points to the new case. If matching indices do exist and the indexed MOP is a generalization, then the entire procedure is repeated recursively using the new MOP as the entry point. However, if the indexed MOP is a specific instance, then the differences between this instance and the new instance are computed, and these two cases are then indexed by their differences.

By traversing to the end of the hierarchy along matching indices, we will end up grouping similar cases together. Ultimately, cases are indexed by their differences, so the unique features of a case will be the crucial indices which lead to a case. Notice that this entire process applies to all matching indices at every step of the computation; this results in the creation of many redundant indices - each case will be indexed by all features of the case itself.

The algorithm controlling the memory creation process is as follows:

0. Given a new instance case to add and a starting MOP:
1. Select:
   A) Features to use as indices
   B) Remove features noted as bad indices
2. Create new Instance-MOP with slots of the new case
3. Start at the given entry MOP:
   A) Set norms to similarities between norms, case
   B) For all attribute/value pairs in the new case:
      1) If matching index from the current MOP doesn't exist, create one pointing to the new case.
      2) Else if index exists to M-MOP (abstraction), recurse with the new MOP as the root (step 3), with the case attributes minus the attribute/value pairs which have already been processed
      3) Else if index exists to Instance-MOP then:
         Create new M-MOP
         Calculate similarities between instance, new mop.
         Add similarities as norms to new M-MOP.
         Calculate differences between instance, new mop.
         Index MOPS based on differences from new M-MOP.

An example of the memory creation process is shown in figures 6.5 and 6.6. For each case which is added to memory, the non-predictive attributes are removed and a new instance of the case is created. For entire cases, the entry MOP will be M-Case; i.e., all cases will be indexed under the Case MOP. Next, indices which match the new instance will be traversed. When collisions with other instances occur, new indices will be created based upon the differences between the two instances. When the first case is added to memory, there are no indices from M-Case to traverse. Consequently, the new case is simply indexed by all the attributes of the case. This is shown in figure 6.5 for I-M-RC.1, the ring concentrator case where the Failure Mode is Startup-No-Output. In this figure, the indices of the Failure Mode, Failed Component, and Failure Correction procedure are shown. Indices are also created for the Failure Cause, Detection Procedure, and the Effect, but these are not shown.

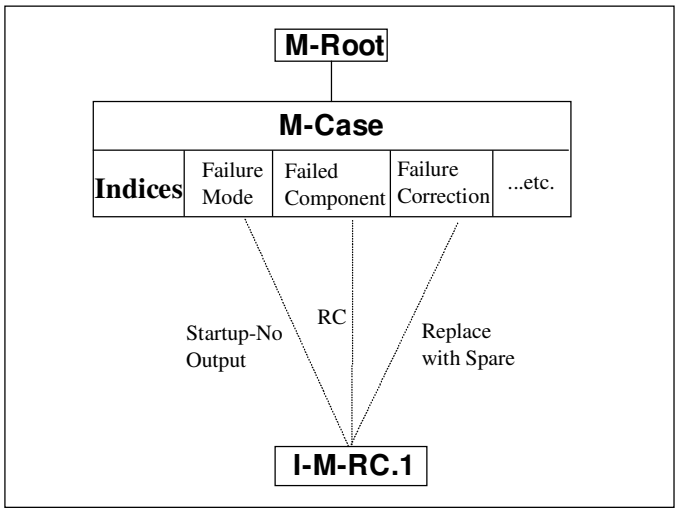

Figure 6.5

GI hierarchy after adding a single ring concentrator case to memory. Only the failure mode, failed component, and failure correction links are shown here.

Figure 6.6 shows how memory organizes itself when a second case is added to memory. In this figure, I-M-RC.1 has already been processed when I-M-GW.1 is given to the system. This new case involves the gateway rather than the ring concentrator. Both cases have the same Failure Mode of Startup-No-Output, and the same long-term Failure Correction Procedure of Replace-with-Spare, but the cases differ in the component which has failed. When I-M-GW.1 is processed, matching indices will be traversed until no more indices exist or a collision occurs. For example, since both cases share the index of Failure Mode, this index will be traversed. A collision will then occur at I-M-RC.1. A new sub-mop, M-Generalization.1 will be created, and the generalizations between I-M-RC.1 and I-M-GW.1 are computed and used as the norms for M-Generalization.1. Next, the differences between I-M-RC.1 and I-M-GW.1 are computed and both instances are indexed from the new generalization based upon these differences. In this example, the cases differ in the Failed Component, so indices of the different failed components are created to access the two instances.

A similar process occurs by traversing the Failure Correction Procedure index, resulting in M-Generalization.2. Since only two cases have been processed, M-Generalization.2 will be identical to M-Generalization.1. However, these MOPs will be different as more cases are processed and different generalizations are computed. Finally, new indices are also created from M-Case for a Failed Component equal to the Gateway, since this index values does not currently exist. This entire process would also occur for the indices which are not shown in the figure (e.g., Failure Detection Procedure, Failure Cause).

In addition to organizing cases according to their differences, the memory creation algorithm also generalizes across all cases and their components. In the example shown, a generalization has been made between a case involving the ring concentrator and a case involving the gateway; both cases share the same Failure Mode and Failure Correction Procedure. By generalizing across all cases, FANSYS is capable of processing generalized questions and also returning generalized answers. Morever, FANSYS may also discover similarities not obvious to humans reading the same cases.

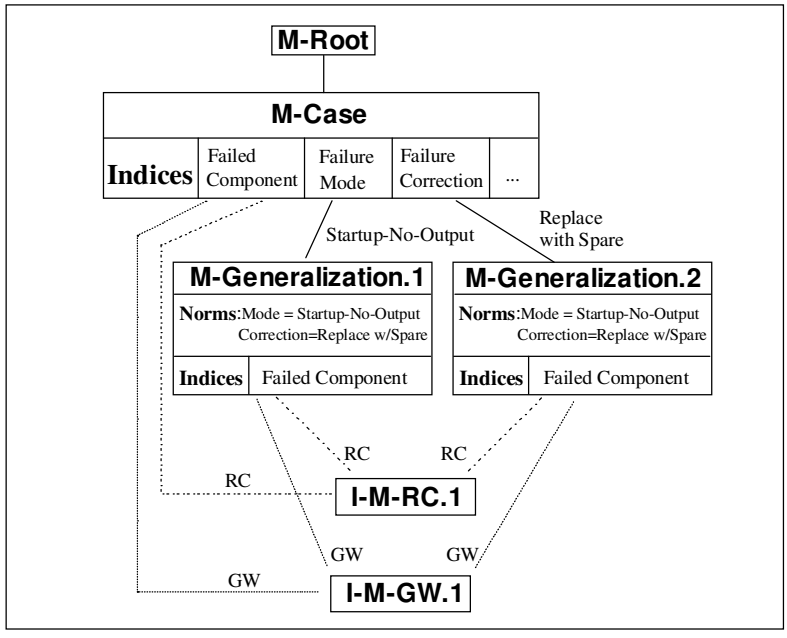

Figure 6.6

GI hierarchy after adding a ring concentrator and gateway case to memory. The generalization MOPs are created to norms holding information common to both cases. The cases themselves are indexed by their differences. In this example, only the failed component is shown. Other indices may be created for the detection procedure, other correction procedures, etc.

An advantage of this indexing scheme is that memory is *self-organizing*. Cases are automatically added to memory such that similar cases are grouped together yet still separated by their differences. Furthermore, cases are indexed by all attributes which compose the case. This allows cases to be retrieved based upon any related aspect of the case rather than mere surface features or predefined keywords. Psychologists refer to this as "deep learning." Furthermore, these indices may be traversed directly, resulting in fast retrieval. Finally, the hierarchical memory structure allows generalizations to be built across all similar cases.

A potential pitfall of this memory scheme is the problem of exponential explosion. As cases with more features are added to the hierarchy, the number of indices created grows exponentially. This was not a problem in the existing implementation of FANSYS since the pre-determined selection of good and bad indices resulted in a streamlined memory representation. However, this would become a serious problem if a large number of cases were added to the system. A solution to this problem is described by Kolodner in her work with long-term memory (Kolodner, 1983). Her solution controls exponential growth by making generalizations. If a MOP indexes a majority of the same cases that are also indexed by its parent, then this MOP is simply removed, and its norms added to the norms of the parent. This eliminates an entire subtree of the hierarchy, effectively controlling the number of MOPs and indices.

Implementation techniques could also be used to cut down on memory costs. The current representation uses expanded lists within slots, indices, and norms. Many of these lists are identical, particularly indices. The same index value may be used in many different indices within many different MOPs. If only one copy of each expanded index is stored and pointers are used

to reference this definition in lieu of the expanded definition, then there would be a large savings in the amount of occupied memory.

6.5 Memory Retrieval Process

The basic retrieval process is simplified once a memory hierarchy has been creating using the algorithm previously described in section 6.5. First, a query must be presented to the system. A query consists of *given* data (variables whose values are provided), and *requested* data (variables whose values are unknown.) The value of the requested data is what the user would like to know. The given data simply consists of a set of attribute/value pairs, and the requested data an attribute without a value. When FANSYS is provided with a query consisting of attribute and value pairs, GI indices are compared to these attributes and values, and matching indices are traversed. Any instances which are reached by the traversal process are returned. Based upon the structures formed by the memory creation process, all indices which point to instances are features of the instances. Since only indices which match the search query are traversed during retrieval, all instances that are retrieved must be relevant to the search query and these instances are accessible without enumeration of indices. Kolodner discusses the benefits of this approach in her work on reconstructive memory (Kolodner, 1983).

The current implementation of FANSYS traverses matching indices in a depth-first manner. The depth-first search is not performed among all indices, but only indices which match the input. This drastically reduces the number of nodes which must be searched, making retrieval very fast. In effect, the attribute/value indices prevent the enumeration of indices and instead allows indices to be content-addressable. This is a desirable property, particularly when a large number of cases need to be represented in memory. On average, the retrieval time will remain constant with respect to the number of cases which are added to memory. Consequently, retrieval time will not suffer when scaling up to large knowledge bases.

Retrieval is also facilitated by the rich indexing scheme. Each case is indexed by all features of the case. This allows case retrieval based upon any distinguishing features. For example, many hierarchical systems may index a case about the ring concentrator only under the category of ring concentrators. This works fine as long as these cases are retrieved in the context of ring concentrators. However, if one wants to retrieve these cases based upon other features (such as the failure-detection method, the failure-mode, etc.) then retrieval is much more difficult because the system has to figure out that the ring concentrator index applies to the new query. In FANSYS, each case is indexed by all features of the case, and the problem of finding appropriate indices is avoided.

Before any indices may be traversed, a suitable entry-point in the parsing hierarchy must be selected where search can begin. A suitable entry-point would be a MOP which subsumes all of the data given in the question. The motivation for choosing such a MOP is that the answer is probably indexed somewhere under this MOP if attributes of the query and the requested information are also indexed under this MOP. While this may not always be true, this assumption

does provide a good place to begin search. As an example, if the goal of the system is to retrieve an entire case, search will typically begin at M-CASE, the MOP which organizes all cases. On the other hand, if the goal of the system is to retrieve an orbital replacement unit, then M-ORU, the MOP which organizes all ORU's, is a logical place to begin search.

The strategy used in FANSYS to select the entry point is to use the MOP which is an abstraction of all data provided in the query. This process begins by starting at the MOPs representing the requested data. If one of these MOPs is an abstraction of the given data and the requested data, then that MOP is used as the entry point. If no MOPs organize all the known data, then the process is repeated with the immediate abstractions of the MOPs compared in the previous step. The first MOP which organizes all data provided in the question is used as the entry-point. If no entry-point is found, the entire process is repeated starting from the MOPs representing the given data. This procedure finds the most specific MOP which also subsumes the given and requested data in a bottom-up fashion. Note that this entire process is computed using the ISA and Slot-Filler hierarchy, not the GI hierarchy.

Once the entry point has been determined, the GI indices which match the input query are traversed and all instances found are returned. The basic retrieval algorithm is outlined below:

0) Supplied a list of requested items and given items:
1) Find an appropriate entry point to begin searching.
   This is the most specific MOP which is an abstraction
   of the given and requested data. Find this MOP searching
   from the bottom-up beginning at the MOPs representing
   the requested or the given data.
2) Try direct retrieval
      Traverse all indices where the attribute/value match
      an attribute/value of the query (the given data).
      If a generalization is reached, repeat the process
      starting at the new MOP (go back to step 2).
      If an instance is reached, save it.
3) Compare all retrieved cases to the input query, and
   return those which match 100% or within some user-
   specified percentage.

As an example of this process, consider the query, "What is the failure correction procecedure for the gateway when the failure mode is loss of output - failure to start?" An annotated memory search and retrieval trace for this query is shown below:

Question: What is the failure correction procedure for the gateway when the failure mode is loss of output - failure to start?

Given Data: (MODE M-STARTUP-NO-OUTPUT)
(FAILED-COMPONENT M-GW)
Requested Data: (M-FAILURE-CORRECTION)

Memory search and retrieval is first given the attributes and values representing the question. In this example, the user is asking for the failure correction procedure (M-FAILURE-CORRECTION) where the mode is loss of output - failure to start (M-STARTUP-NO-OUTPUT) and the failed component is the gateway (M-GW). The parsing step is not shown here.

```
Searching bottom-up for Entry-MOP based on requested:
Entry-MOP=M-CASE
```

The system has found an Entry-MOP by searching bottom-up from the requested data (M-FAILURE-CORRECTION). The most specific MOP which subsumes M-FAILURE-CORRECTION, M-GW, and M-STARTUP-NO-OUTPUT) is M-CASE. The algorithm can now begin by searching for GI indices from M-CASE which match the query.

```
Performing Direct Search. Indices:
                        ((MODE M-STARTUP-NO-OUTPUT)
                       (FAILED-COMPONENT M-GW))
Current MOP: M-CASE
  Input matches indices to: (COMP.2499 COMP.2982)
```

By traversing the index MODE=M-STARTUP-NO-OUTPUT from M-Case, MOP COMP.2499 can be accessed. This MOP generalizes across all cases where the Mode is Startup-No-Output. The index FAILED-COMPONENT=M-GW leads to MOP COMP.2982. This MOP generalizes across all cases where the Failed Component is the
Gateway.

```
Current MOP: COMP.2499
  Input matches indices to: (I-M-GW.1)
Current MOP: I-M-GW.1
Current MOP: COMP.2982
  Input matches indices to: (I-M-GW.1)
Found: (M-CASE COMP.2499 COMP.2982 I-M-GW.1)
```

The traversal process is repeated from each sub-MOP of M-CASE. Both paths lead to the same instance, I-M-GW.1, which is the first gateway case. One path goes from M-CASE to COMP.2499 to I-M-GW.1 by traversing the index MODE-M-STARTUP-NO-OUTPUT from M-CASE to COMP.2499, and then the index FAILED-COMPONENT=M-GW from COMP.2499 to I-M-GW.1. The other path traverses a failed component index first, and then the mode index second to access the case. Note that although the multiple paths to an instance may seem redundant, this indexing scheme allows generalizations to be built across components, failure modes, failure correction procedures, etc. The traversal path for this example is depicted graphically in figure 6.7. Only the paths which are actually traversed are shown; all other indices are not considered since the attributes and values do not match those provided in the query.

After traversal has finished, all instances which completely match the input are returned and the portions of the case which match the requested data are generated in English.

```
Percent Match of Values to Query: ((M-CASE 0) (COMP.2499 0) (COMP.2982) (I-M-GW.1 1))
Picking out those that match 100%
Results of Direct Search: (I-M-GW.1)
```

[1] SHORT TERM CORRECTION PROCEDURE : THE SYSTEM MANAGEMENT SELECTS A BACKUP GATEWAY TO REPLACE THE GATEWAY FROM A LOOK-UP TABLE MAINTAINED WITHIN THE DMS . THE SYSTEM MANAGEMENT POWERS DOWN THE GATEWAY . THE SYSTEM MANAGEMENT POWERS UP THE BACKUP GATEWAY . CORRECTION PROCEDURE : THE SYSTEM MANAGEMENT LOGICALLY DISCONNECTS THE GATEWAY FROM THE DMS NETWORK . THE SYSTEM MANAGEMENT LOGICALLY CONNECTS THE BACKUP GATEWAY TO THE DMS NETWORK .

[2] LONG TERM CORRECTION PROCEDURE : THE CREW PHYSICALLY DISCONNECTS THE GATEWAY FROM THE DMS NETWORK . THE CREW PHYSICALLY CONNECTS THE BACKUP GATEWAY TO THE DMS NETWORK . THE CREW LOGICALLY DISCONNECTS THE GATEWAY FROM THE DMS NETWORK . THE CREW LOGICALLY CONNECTS THE BACKUP GATEWAY TO THE DMS NETWORK .

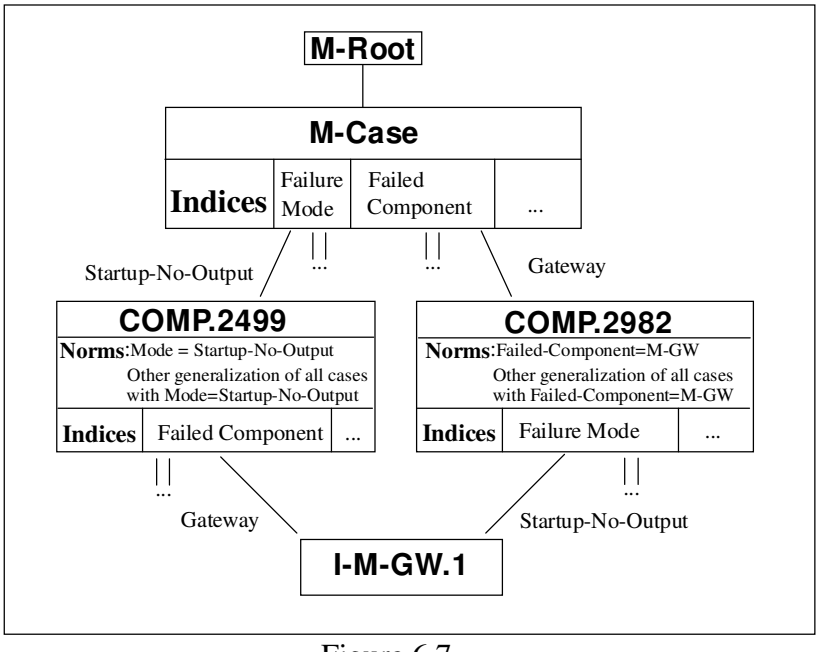

Figure 6.7

Traversal paths for example query given Failure-Mode=Startup-No-Output and Failed-Component=M-GW starting from Entry MOP=M-CASE

A second trace is shown below. In this example, the user is asking FANSYS to specify details of the lookup-backup correction procedure:

Question: What are the correction steps for the lookup backup procedure when the failed component is the standard data processor?

Given Data: (FAILURE-CORRECTION M-LOOKUP-BACKUP)
(FAILED-COMPONENT M-SDP)
Requested Data: (M-CORRECTION-AND-GROUP)

```
Searching bottom-up for Entry-MOP based on requested:
M-OR-GROUP
M-AND-GROUP
No MOP found which subsumes given data
```

Starting from the requested data, a MOP was not found which subsumes all data provided in the question. By performing another search from the given data, a suitable MOP is found. This MOP turns out to be M-LOOKUP-BACKUP, one of the MOPs actually specified by the user. As a result, there will be no indices of FAILURE-CORRECTION=M-LOOKUP-BACKUP starting from M-LOOKUP-BACKUP, but there will be indices for the failed component.

```
Searching bottom-up for Entry-MOP based on given:
Entry MOP=M-LOOKUP-BACKUP

Performing Direct Search. Indices:
                        (FAILURE-CORRECTION M-LOOKUP-BACKUP)
                        (FAILED-COMPONENT M-SDP)

Current MOP: M-LOOKUP-BACKUP
  Input matches indices to: (I-M-LOOKUP-BACKUP.3061)
Current MOP: I-M-LOOKUP-BACKUP.3061

Found: (M-LOOKUP-BACKUP I-M-LOOKUP-BACKUP.3061)
Percent Match of Values to Query: ((M-LOOKUP-BACKUP 0) (I-M-LOOKUP-BACKUP.3061 1))
Picking out those that match 100%
:
Results of Direct Search: (I-M-LOOKUP-BACKUP.3061)

[1]     THE SYSTEM MANAGEMENT SELECTS A BACKUP STANDARD
        DATA PROCESSOR TO REPLACE THE STANDARD DATA PROCESSOR
        FROM A LOOK-UP TABLE MAINTAINED WITHIN THE DMS . THE SYSTEM
        MANAGEMENT POWERS DOWN THE STANDARD DATA PROCESSOR.
        THE SYSTEM MANAGEMENT POWERS UP THE BACKUP STANDARD DATA
        PROCESSOR. CORRECTION PROCEDURE : THE SYSTEM MANAGEMENT
        LOGICALLY DISCONNECTS THE STANDARD DATA PROCESSOR FROM
        THE DMS NETWORK . THE SYSTEM MANAGEMENT LOGICALLY CONNECTS
        THE BACKUP STANDARD DATA PROCESSOR TO THE DMS NETWORK.
```

In this example, we are not retrieving an entire failure case instance, but an individual instance of the lookup-backup failure correction procedure. As illustrated in figure 6.8, this instance is indexed directly under M-LOOKUP-BACKUP by traversing only a single index. By finding an appropriate entry-point MOP to begin search, the number of MOPs which need to be examined can be drastically reduced.

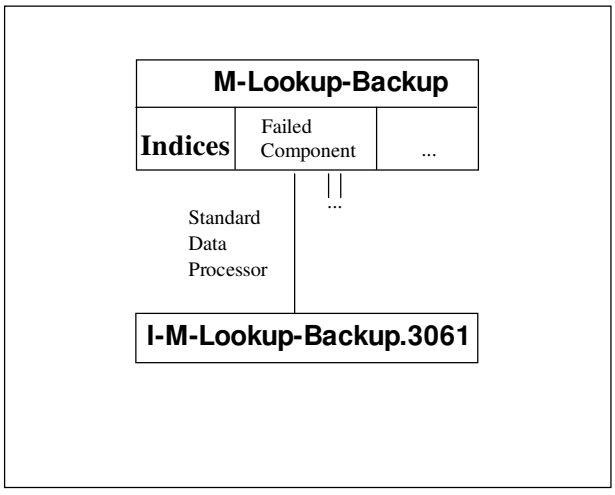

Figure 6.8

Traversal path for example query given Failed-Component=Standard Data Processor with the Entry MOP=M-Lookup-Backup

The basic retrieval algorithm works well when the QA module is given enough information to traverse indices all the way down to the instances. However, the system should also be capable of handling more difficult questions where the input may not give enough information to access a specific case. One solution is to *elaborate* on the input query. In general, elaboration is the process of generating related items to use for search based upon a particular input query. For example, when asked to name all 50 states of the United States, many people will elaborate on the question by thinking of states they have visited, states where friends or relatives live, postal abbreviations, or perhaps by visualizing a map and mentally traversing the states. None of these retrieval techniques were directly specified by the input query, but were conjured up by searching for a context where an appropriate case may be found.

While Kolodner specifies a number of elaboration techniques in her work on memory retrieval (Kolodner, 1983), the current implementation of FANSYS uses only a set of simple elaboration strategies. The first strategy removes some of the constraints of search by allowing all indices to be traversed which have the same attributes as the requested information. This amounts to adding to the search query all possible requested items. By allowing more indices to be traversed, there is a greated chance of retrieving an instance. Furthermore, the new indices which are traversed are related to the input question, so we are still restricting search to relevant portions of memory. As an example, if we are searching for a ring concentrator, then all ring concentrator indices will be traversed even if the value doesn't match an attribute/value given in the query. This is often just what is needed to retrieve cases, since the requested item is typically not known and hence not present in the query. This elaboration technique is called *elaborating on the requested information*.

If elaborating on the requested information still fails to retrieve cases, then the search can be broadened by allowing even more indices to be traversed. This is accomplished in FANSYS by retrieving the attributes which are contained in the *elaborations* slot of the each MOP traversed. The information in the elaborations slot specifies additional attributes and other MOPs where search may be conducted. Any indices which have these attributes are allowed to be traversed, even if the value of the index does not match attribute/values of the query. Currently,

the elaborations are pre-defined in FANSYS to include the failure-mode, failure-detection procedures, and failure correction procedures. A topic for future work would include methods of learning appropriate MOPs and attributes for elaboration.

Finally, if the above two elaboration strategies still fail to retrieve any applicable cases, then a depth-first-search of the entire memory can be employed, retrieving all cases. Each case can then be matched with the query, and the most relevant returned. With a large memory, this will be a time-consuming process and should be done only as a last resort. In our experiments, a complete search of memory was never necessary.

The retrieval process, accounting for elaboration, is summarized below:

1) Try basic retrieval
2) If no matches found, allow elaboration on the requested data. Run the retrieval algorithm again, but allow traversal of any indices matching the requested data.
3) If no matches are found, allow elaboration on all attributes contained in the elaborations slot of each MOP traversed. Run the retrieval algorithm again, but allow traversal of any indices matching these new attributes. Also run the retrieval algorithm starting at all MOPs specified in the elaborations slot.
4) If no matches are found, perform a depth-first-search and retrieve all instances.
5) Compare all retrieved cases to the input query, and return those which match 100% or within some user-supplied percentage. This could also be modified to return the highest N matches.

An example of retrieval with elaboration is shown in the trace below. In this example, the user is asking for the failed component when the heartbeat detection procedure is used. The only information given in the query is the detection procedure. This is not enough information to access cases. FANSYS will have to elaborate and traverse indices not specified in the question.

```
Question:       What is the failed component when the failure detection procedure
                is the heartbeat detection procedure?

Given Data:             (FAILURE-DETECTION M-HEARTBEAT-DETECTION)
Requested Data:         (M-ORU)

Searching bottom-up for Entry-MOP based on requested:
Entry-MOP=M-CASE

Performing Direct Search. Indices: (DETECTION M-HEARTBEAT-DETECTION)
Current MOP: M-CASE
  Input matches indices to: (COMP.4325)
Current MOP: COMP.4325
Found: NIL
Values: NIL
```

COMP.4325 is the MOP which generalizes across all heartbeat detection procedures; i.e., it is accessed through the index DETECTION=HEARTBEAT-DETECTION.

```
Picking out those that match 100%
Results of Direct Search: NIL
None found.
```

No cases are retrieved since the single index of DETECTION=HEARTBEAT-DETECTION is not sufficient to access a case. Consequently, the system gets "stuck" at COMP.4325. The traversal is depicted in figure 6.9 in the encircled area. FANSYS will next lessen the search constraints by allowing traversal of any indices which have an attribute of FAILED-COMPONENT. This is accomplished by performing the search again with a new wildcard added to the search query.

```
Elaborating using wildcard for requested information.

 * matches with anything.  Indices:
                                (DETECTION M-HEARTBEAT-DETECTION)
                                (FAILED-COMPONENT *)

Performing Search. Entry MOP=M-CASE
Current MOP: M-CASE
  Input matches indices to: (COMP.4199 COMP.3569 COMP.2992 COMP.1321 COMP.4325)
Current MOP:COMP.3569
Current MOP: COMP.1321
Current MOP: COMP.4199
  Input matches indices to: (I-M-SDP.2)
Current MOP: I-M-SDP.2
Current MOP: COMP.2982
  Input matches indices to: (I-M-GW.2)
Current MOP: I-M-GW.2
```

COMP.3569 and COMP.1321 respectively organize cases for the time generation unit and the ring concentrator. Since the query has been expanded to traverse any index with a FAILED-COMPONENT both of these MOPs will be examined. However, none of these MOPs have indices which match the query, so traversal halts. However, COMP.4199 organizes all cases for the standard data processor. This MOP does have indices which match the input data of DETECTION=M-HEARTBEAT-DETECTION which lead to the instance I-M-SDP.2. Similarly, the MOP which organizes all gateway cases also leads to an instance, I-M-GW.2.

```
Current MOP: COMP.4325
  Input matches indices to: (I-M-GW.2 I-M-SDP.2)
Current MOP: I-M-GW.2
Current MOP: I-M-SDP.2
```

Before elaboration, FANSYS was only able to traverse to COMP.4325, the generalization for HEARTBEAT-DETECTION. However, the system is now capable of reaching instances due to the expanded search query. All failed-component indices are traversed, leading to the GW and

SDP instances previously found. These instances are returned and the requested information generated from the retrieved cases. The paths traversed before and after elaboration are shown in figure 6.9.

Found: (COMP.4325 I-M-GW.2 M-CASE COMP.4199 COMP.2982 I-M-SDP.2)
Percent Match of Values to Query:
((COMP.4325 0) (I-M-GW.2 1) (M-CASE 0) (COMP.4199 0)
(COMP.2982 0) (I-M-SDP.2 1))
Picking out those that match 100%
Results of elaboration: (I-M-GW.2 I-M-SDP.2)

[1] GATEWAY
[2] STANDARD DATA PROCESSOR

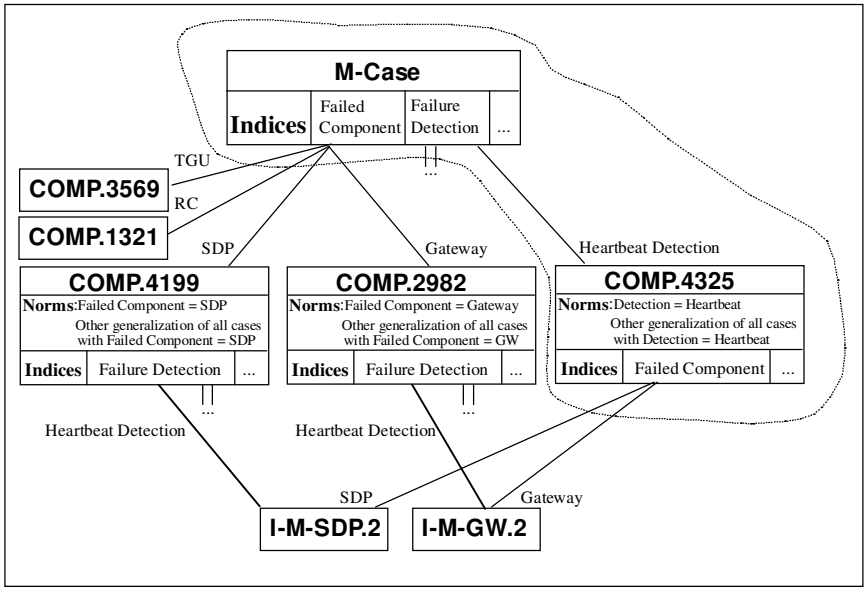

Figure 6.9

Traversal path for example query with elaboration given Failure-Detection=Heartbeat Detection with the Entry MOP=M-Case. The encircled area represents MOPs traversed before elaboration. After elaboration all paths shown are traversed.

A final annotated trace depicting a case where FANSYS uses a last resort strategy of depth-first search along all unspecified indices is shown below. In this example, the system is given a single failure mode, failure during operation with erroneous output, and is asked to return all of the failure causes associated with this mode. The given information and requested information alone are not enough to reach any specific instances since many cases share the same causes when the mode is erroneous output. Consequently, FANSYS must expand upon unspecified indices to access cases.

Question: What is the failure cause when the failure mode is failure during operation - erroneous output?

Given Data: (FAILURE-MODE M-OPERATIONAL-ERRONEOUS-OUTPUT)
Requested Data: (M-FAILURE-CAUSE)

```
Searching bottom-up for Entry-MOP based on requested:
Entry-MOP=M-CASE

Performing Direct Search. Indices: ((MODE M-OPERATIONAL-ERRONEOUS-OUTPUT))

Current MOP: M-CASE
Input matches indices to:          (COMP.1902)
Found: NIL
Values: NIL
Picking out those that match 100%
Results of Direct Search: NIL
None found.
```

Direct search traverses the single index MODE=M-OPERATIONAL-ERRONEOUS-OUTPUT from M-CASE to COMP.1902 and then stops. Since no instances are reached, elaboration using the requested information is employed, allowing FANSYS to traverse any CAUSE indices.

```
Elaborating using requested information.
* matches with anything.  Indices:        (MODE M-OPERATIONAL-ERRONEOUS-OUTPUT)
                                           (CAUSE *)
Performing Search. Entry MOP=M-CASE
Current MOP: M-CASE
Input matches indices to:          (COMP.1326 COMP.4376 COMP.1325 COMP.1324
                                   COMP.1902 COMP.1323 COMP.1322)
Current MOP: COMP.1326
Current MOP: COMP.4376
...
Current MOP: COMP.1323
Input matches indices to: (I-M-GW.3)
Current MOP: I-M-GW.3
Current MOP: COMP.1322
Input matches indices to: (I-M-TGU.3)
Current MOP: I-M-TGU.3
Found:  (COMP.1326 COMP.1325 COMP.1324 COMP.1323 COMP.1322 I-M-GW.3 M-CASE
        COMP.4376 I-M-TGU.3)
Values: ((COMP.1326 0) (COMP.1325 0) (COMP.1324 0) (COMP.1323 0) (COMP.1322 0)
        (I-M-GW.3 1) (M-CASE 0) (COMP.4376 0) (I-M-TGU.3 1))
Picking out those that match 100%
Results of elaboration:  (I-M-GW.3 I-M-TGU.3)

[1] PIECE-PART FAILURES
[2] ERRONEOUS INPUT

Found good match. Elaborate further anyway? (Y/n) Y
```

After allowing traversal of all CAUSE indices, additional MOPs are traversed and two instances are returned. These two instances are specified uniquely by the failure causes of Piece-Part Failures and Erroneous Input. However, additional cases may exist which are not retrieved by the first elaboration strategy. To retrieve these cases, FANSYS can elaborate upon all unspecified slots, illustrated in the final section of the trace below:

```
Elaborating on unspecified entry slots...
Indices: ((FAILED-COMPONENT *) (MODE M-OPERATIONAL-ERRONEOUS-OUTPUT)
        (CAUSE *) (DETECTION *) (CORRECTION *) (EFFECT *))
Performing Search. Entry MOP=M-CASE
Current MOP: M-CASE
Input matches indices to: (COMP.3569 COMP.2499 ... COMP.1902)
Current MOP: COMP.3569
        Input matches indices to: (I-M-SDP.1 I-M-SDP.2 I-M-SDP.3)
Current MOP: I-M-SDP.1
Current MOP: I-M-SDP.2
Current MOP: I-M-SDP.3
Current MOP: COMP.2499
Input matches indices to: (I-M-GW.1 I-M-GW.2 I-M-GW.3)
....
Found: (COMP.3569 I-M-RC.3 I-M-TGU.1 COMP.4376 ...)
Values: ((COMP.3569 0) (I-M-RC.3 1) (I-M-TGU.1 2591/2940) (COMP.4376 0) ...)

Picking out those that match 100%
Results of secondary elaboration: (I-M-SDP.3 I-M-GW.3 I-M-RC.3 I-M-TGU.3 I-M-TGU.3)

[1] PIECE-PART FAILURES
[2] ERRONEOUS INPUT
[3] CONTAMINATION
[4] TEMPERATURE ( HIGH OR LOW )
[5] MECHANICAL SHOCK
[6] THERMAL SHOCK
```

In the final section of the trace, redundant sections have been removed for brevity. Only the first matching index from M-CASE to COMP.3569, is expanded. This index leads to all cases involving standard data processors. The other indices are traversed similarly. By allowing traversal of unspecified indices, FANSYS is able to retrieve all matching cases. In this example, additional failure causes are found which were not returned from elaboration using only the requested information. However, a large number of indices are traversed which do not lead to relevant cases. For example, the trace indicates that I-M-SDP.1 and I-M-SDP.2 are examined. These cases do not match the search criteria since the failure mode is not erroneous-output, and are subsequently not returned. Traversal to these irrelevant cases is time-consuming and inefficient. Although only a few general questions require this last resort depth-first search strategy for a complete answer, future work for FANSYS includes the elimination of this last resort strategy and the inclusion of alternate context search and more intelligent elaboration strategies.

7. User Interface

The prototype version of FANSYS was primarily designed to experiment with new paradigms in knowledge representation and the modeling of conceptual processes. As a result, user interface issues have received secondary priority. An implementation version of FANSYS would certainly require an additional emphasis placed upon the user interface. User surveys, a design rationale, task analysis, and user testing are all important phases of interface design which needs to be performed. (Blattner, 1992; Gentner & Grudin, 1990; MacLean et. al, 1991). Results from any of these phases can significantly affect the development of the final system.

While the user interface employed in FANSYS is currently in the preliminary stages of development, it does address a number of design issues. First, FANSYS utilizes the X window system through the GARNET UIMS developed at Carnegie Mellon University (Myers, 1991). This allows for integration with other unix-based applications and provides a foundation for future development. Second, an emphasis has been made to simplify the question/answering sessions since this is anticipated to be the most frequently used aspect of the system. This emphasis is manifested through the use of point-and-click buttons for input and a natural language generator for output. Third, the capability to parse questions in natural language has been provided. Finally, a graphical browsing tool has been developed to aid knowledge engineers in examining and modifying memory.

The user interface of FANSYS is composed of the desktop shown in figure 7.1. There are two main windows. The top window is a trace window which maintains a record of the processes used by FANSYS during text comprehension and question answering. This window is used by FANSYS for output of technical information only. The bottom window is an input/output window which displays input case descriptions, input questions, and answers to these questions. This is the primary window; users unfamiliar with the underlying model of FANSYS may elect to hide the trace window and work only with the input/output window. The overall control of the program is handled through a button panel located in the lower right hand corner of the screen. This panel allows the user to enter a search query, inspect memory with a graphical browser, get help about the system, or exit the system.

The first action that must be performed with the user interface is to parse textual descriptions of cases to build up the knowledge base, as described in section 5. As these case descriptions are parsed, the trace window displays the processes involved in understanding the input text. A sample of this process is shown in figure 7.2, which shows the behavior of the system as it begins to parse the first ring concentrator case.

Once the knowledge base has been constructed, question answering may begin. In a typical session the cases will be parsed offline and the knowledge base stored. A user may then start FANSYS directly in question/answering mode. Queries are given to the system in either of two ways: constructing a query through a series of point-and-click buttons, or entering a query via natural language. The point-and-click buttons are simple enough for a novice to use and also gives the user an idea of what types of questions may be asked. Alternately, the natural language interface allows for more types of queries to be created and creates a context in which dialogue is possible.

Insert Figure 7.1

Insert Figure 7.2

The point-and-click interface is invoked by clicking on the button labeled "Enter Search Query." This will bring up a sequence of button menus which will allow the user to create a query. This query is constructed in terms of "given information" and "requested information". For example, to learn what the failure cause may be for the ring concentrator when the failure mode is startup-no-output, the user will click on the button "Ring Concentrator" to indicate the component which has failed, and "Startup-No-Output" to indicate the failure mode. Together, this data comprises the given information. Finally, the user will click on the button "Failure Cause" to indicate the requested information. Examples of the button menus are shown in figures 7.3-7.9.

Figure 7.3 is the first pop-up window which appears after the user has clicked on the "Enter Search Query" button. This window consists of a number of radio-style buttons which allows the user to select the unknown variables (i.e., the requested data) in the query. In this example, the user is asking the system to determine the failure mode and the failure cause. These menus give the user the opportunity to select or deselect buttons, so any action is easily "undone." When the user is satisfied with the selected choices, the "Continue" button brings up the next menu which prompts the user to enter the given information.

The next menu shown in figure 7.4 prompts the user to enter the component which has failed. If the component is now known, the user may click on the button labeled "Unknown." This menu is always presented to the user since it is anticipated that most queries will contain a known component which has failed. Currently, FANSYS allows the user to select only a single failed component. Upon selecting a component, the menu shown in figure 7.5 appears.

Figure 7.5 depicts a controlling menu which allows the user to give any further information about the search query which may be known. If the failure mode is known, then the user would click upon the "Failure Mode" button. Similarly, if the failure causes are known, then the user would click upon the "Failure Cause" button. Each button will bring up a separate menu allowing the user to specify further data. These menus are shown in figures 7.6-7.9. Note that when only one choice is appropriate, the interface only allows one choice to be made. For example, the user may not specify a query which has two different failure modes since no case has two modes (only an "and" of the input is currently supported; attributes may not yet be "or'd" together). When multiple choices are appropriate, as in specifying the failure causes of figure 7.7, a radio-style button panel is presented to allow many choices to be made. After the user has specified further information, the menu shown in figure 7.5 will re-appear, allowing the user to specify even more information or start the search.

Insert Figure 7.3

Insert Figure 7.4

Insert Figure 7.5

Insert Figure 7.6

Insert Figure 7.7

Insert Figure 7.8

Insert Figure 7.9

Once all the information has been given to the system, the button "Start Search" instructs FANSYS to search memory for relevant cases which conform to the query using the retrieval strategies discussed in section 6. A trace of the processing steps is shown in the top trace window, while the query and answer is shown in the bottom question/answer window. Figure 7.1 shows the final result of a sample query where the user has asked FANSYS to determine the detection procedure when the failure cause is thermal shock and contamination, the failure mode is loss of output - failure during operation, and the failed component is the standard data processor. The detection procedure corresponding to the case matching the query is output in the bottom question/answer window via a natural language generator. After cases have been retrieved, the user will be asked if the system should stop or continue searching for more cases which may match the input query.

In addition to using buttons to create a search query, a button also exists to give the user help. This button brings up a scrollable window which describes the overall system, how to use the interface to answer questions, and how queries should be constructed. Future versions of FANSYS will incorporate context-sensitive help. A sample of the user help window is shown in figure 7.10. This particular snapshot shows the format for posing questions to the system via natural language through the question shell.

The question shell is an alternative method of creating queries by allowing users to enter questions in English. As described in section 5, questions are constrained by the format in which they may be posed. Refer to figure 7.10 for a description of the question format enforced by the system. In this figure, items within parentheses denote variables, where several different values may fill that variable. For example, the "given-item" may be "ring concentrator", "gateway", or any other component which may fail. The brackets denote optional patterns which may repeat. For example, a user may ask "What is the failure mode and the failure cause..." to ask the system to return many different answers.

Once a question has been entered, it is processed in order to convert punctuation into the appropriate lexical symbols used by FANSYS (for example, a "." becomes "*PERIOD*"). The sentence is then passed to the parser, which processes it just like any other piece of text (e.g., a case description). Once the question MOP corresponding to the query has been built as described in section 5, the query is constructed and is then passed to the memory search and retrieval module. The retrieved answers are generated in English, just as with the X-Windows interface. An important point worth noting is that the query that is generated is identical to that generated by the X-Windows interface. The only difference is the manner in which the information within the query is specified -- via a point and click mechanism, or via natural language. For instance, figure 7.11 shows the input/output behavior of FANSYS when a question is being processed after it has been typed into the question shell. (This question is the same question used in section 5.3 to illustrate the input/output behavior of FANSYS when parsing a question.) The top window is a trace of the parsing process for the question, while the bottom window shows the results of the query after the memory search module has retrieved the answer.

Insert Figure 7.10

Insert Figure 7.11

The final component of the user interface is a graphical browser called the Memory Inspector. The Inspector allows a knowledge engineer to inspect and modify memory. The user may click on boxes which represent MOPs. The boxes show the contents of each MOP and the indices connected to other MOPs. By simply clicking on MOPs and indices, memory can be traversed and the hierarchy visually inspected. A sample of the Inspector is shown in figure 7.12. This figure shows the MOP representing M-CASE in the upper-left corner. The user may then choose to expand on any slots or fillers of M-CASE. Clicking upon "SPECS" brings up all specializations of M-CASE; i.e., all of the cases in memory. These are shown in the scrollable window below the MOP of M-CASE. The user may further expand upon one of these cases by simply clicking on it. In the figure, I-M-RC.1, the first ring concentrator case, has been expanded and its slots shown in the upper-right corner. In addition to providing a convenient way to examine and visualize memory, any of these values may also be modified by clicking on them and typing in new values.

Insert Figure 7.12

8. Current Status

All of the material discussed in the previous sections has been implemented in a prototype version of FANSYS developed at the U.C. Davis AI lab on Sun workstations. Although only a prototype, a significant effort has been spent in implementation. The source code, written in the CMU Common Lisp programming language, occupies approximately 234K of disk space. The data files used to represent the domain occupies approximately 75K. CMU Common Lisp is a public domain Common Lisp implementation under development at Carnegie Mellon University and is available by anonymous FTP from lisp-rt1.slisp.cs.cmu.edu. Although FANSYS was developed using CMU Common Lisp, the core routines for parsing and memory search/retrieval will run under any version of Common Lisp (e.g., Kyoto Common Lisp, Ibuki Common Lisp, or Franz Allegro Common Lisp). However, some of the user interface routines for interprocess control are dependent upon CMU Common Lisp. The graphical user interface routines require the GARNET User Interface Management System. GARNET is another public domain product developed at CMU and is available by anonymous FTP from a.gp.cs.cmu.edu. GARNET requires the CLX X Library package to be installed on the Lisp platform used.

The current implementation of the parser is capable of reading the first ring concentrator case where the failure mode is loss of output - failure to start. The case text shown in figure 2.1 is an abstraction of the original failure case description. The abstracted case has been further converted to the format shown in the figure; this conversion allows FANSYS to process punctuation more easily.

While the current implementation of the parser has only been developed to parse the ring concentrator case, a total of twelve cases have been hand-coded to test the knowledge representation and memory search/retrieval schemes. These cases cover the descriptions for the ring concentrator, gateway, standard data processor, and the time generation unit. To input these cases, the underlying knowledge to understand the cases is first developed. Next, instantiated knowledge structures representing the cases are presented to the system. These cases are shown in Appendix A. FANSYS then instantiates these knowledge structures and creates GI indices to access the cases for question/answering.

After the twelve cases have been loaded, FANSYS occupies approximately 10 Mb of memory. The core of CMU Common Lisp and the GARNET and CLX interface code occupies another 15 Mb, bringing the total memory requirements to 25 Mb. While the memory requirements are large, little effort has been done to optimize the time and space efficient of the code. More efficient memory representation implementation, particularly in the GI indices where redundancy is high, would result in dramatic memory compaction.

9. Future Work

The goal of FANSYS has been to design a system capable of integrating knowledge constructs with processing strategies to comprehend textual input and answer questions in the domain of failure analysis within the Data Management System of Space Station Freedom. The prototype system has demonstrated the feasibility and power of the approach described in this report. However, as work has progressed, many areas have been discovered where FANSYS can be improved. These areas include the user interface, integration with other tools for fault management, integrating memory structures used for parsing and retrieval, the scope of the implementation, investigating the use of parallel machines, machine acquisition of domain-specific knowledge and search strategies, incorporating model-based and functional-based reasoning strategies, generating belief inferences, and subjective comprehension and verification of input text.

9.1 User Interface

The point-and-click buttons of the user interface currently help make the system intuitive and easy to use. However, the interface has been built upon the system architecture. A deployable system will need to be based upon the tasks required by the users. This may require redesigning portions of the interface and the system itself. Furthermore, additional functionality needs to be designed into the interface so that more complex editing and more flexible question queries can be posed to FANSYS.

9.2 Integration with Other Tools

Additional flexibility can be obtained by integrating FANSYS with other failure analysis tools. For example, FANSYS could provide explanations in natural language to users of NASA's FEAT (Failure Environment and Analysis Tool) (Iverson and Patterson-Hine, 1990; Patterson-Hine and Iverson, 1990; Stevenson, Miller, and Austin, 1991). FEAT provides excellent reasoning capabilities for determining the causes and effects of a failure, while FANSYS provides natural language and the context of a case from which explanations can be based.

9.3 Integration of Memory Structures

Another important area for future work is a closer integration of the knowledge structures used during parsing and those generated in the GI hierarchy through the generalization process. Both hierarchies encode the similar knowledge, but in a different fashion, resulting in redundancy within the system. A more efficient approach would integrate the two memory systems into a single hierarchy which is self-organizing and also capable of supporting the parsing, generalization, and retrieval processes.

9.4 Implementation Scope

One of the next steps for future work is to extend the scope of the project so that more cases can be parsed and understood. This will involve additional work in knowledge representation to encode the cases for the console and cupola. Further work in knowledge representation will also be necessary if a finer granularity of understanding is desired. Continued work on defining additional lexical patterns, inference strategies, and processing MOPs is necessary to handle additional textual cases. Currently, the system encodes approximately 120 lexical patterns and a few domain specific processing strategies, characterizing the parts of the domain necessary to understanding the first ring concentrator case. More lexical knowledge must be added to the system to parse the remaining cases. Question answering sessions could also be made more general and less constrained by defining additional patterns to handle natural language queries from a user of the system. Additionally, the generator would benefit from a more elaborated syntactical system for specifying patterns and from better strategies for choosing which patterns to use during generation. Although lexical and processing knowledge has been encoded for the first case alone, the mechanisms developed for the first case should prove general enough to make the parsing of additional cases involve little more than elaboration of lexical and processing knowledge.

Performance improvements are also possible within FANSYS if its code is optimized. Throughout the implementation, little effort has been made to generate code which is efficient in both runtime and memory requirements. The speed of execution during parsing could be improved by integrating some of the memory search and retrieval techniques. More intelligent elaboration strategies will also improve the speed of execution during memory retrieval. In particular, the last resort strategy of depth-first search should be replaced with more intelligent elaboration strategies. These strategies would require knowledge of the domain to search alternate areas or concepts within memory that apply to the original query. By integrating the memory search functionality with the parser, more efficient access and indexing of MOPs during parsing would be facilitated, speeding up the performance of the parser substantially. Finally, the use of a parallel machine would result in a dramatic increase in memory creation and retrieval speed since many of the traversal paths could then be examined simultaneously.

9.5 Knowledge Acquisition

Self-organization and learning are also extensions that need to be addressed in FANSYS. Although the GI hierarchy is self-organizing during the creation processes, the processes that govern memory retrieval are fixed. A significant improvement to FANSYS would include the capability to learn elaboration strategies as the system is used. As questions are posed and cases retrieved, the system can be trained as to which cases and procedures relate to each other. When presented with a general question or an unknown question, FANSYS could rely upon previous cases for elaboration strategies which may be appropriate.

FANSYS is currently able to read and build the conceptual representation of input text that describes failure diagnosis and repair in a given flight system. However, FANSYS should also be able to build a coherent model of (1) the causal relationships that exits among components of the flight system, (2) the effects and causes of failures in that system, and (3) the procedures used to deal with those failures. This domain model must be dynamically built as a result of

remindings (Schank, 1982) that occur when similar textual inputs are processed. FANSYS should also account for how cross-contextual remindings occur when reading input descriptions of failure analysis in different flight systems, and how memory search and retrieval processes must be able to adapt to memory changes that result from these remindings. To model this reminding process, FANSYS must organize and maintain failure-analysis memories by using a hierarchical organization scheme of the type used the OCCAM system (Pazzani, 1988) in addition to the GI hierarchy.

FANSYS must also be able to handle input text that is not directly dependent on the knowledge constructs and lexical entries initially encoded in the system. To deal with this knowledge engineering bottleneck, FANSYS must be able to dynamically acquire and augment its knowledge during text comprehension. Consequently, the model of text comprehension and question answering implemented in FANSYS will include explanation-based and case-based strategies, such as those described in (Kolodner, 1988) and (Schank and Riesbeck, 1989), for dynamically acquiring and encoding knowledge constructs associated with a given domain or a given natural language.

9.6 Model and Functional Based Reasoning

Another extension which would increase the power of FANSYS is the addition of model-based and functional-based reasoning (Hodges, 1989). These techniques allow for an entirely new level of analysis. In model-based reasoning, the system creates its own model prototype based on the input data. For example, if the system were given that "component A is connected to component B via the network", then an internal model of two connected components via a network would be created. Consequently, the system can then draw inferences and conclusions based on the model. This type of reasoning would be necessary if the user wanted to know what would happen if component A were switched with component B. In this case, the system could switch the components in its model and discover the ensuing results. Similarly, in functional-based reasoning, conclusions are drawn based upon the functions of various devices. Providing FANSYS with both types of reasoning will require the development of new memory structures which accommodate these and the reasoning approaches previously discussed.

9.7 Subjective Comprehension of Input Text and Questions

Comprehension of input text and questions in FANSYS must also be influenced by the *ideological perspective* (Carbonell, 1981) that FANSYS may have about diagnosis and repair procedures in a given domain. That is, FANSYS must attempt to understand descriptions of failure analysis and relate their conceptual content to its own beliefs and justifications involving related and/or similar failures. If FANSYS reads descriptions that are inconsistent with its beliefs, then FANSYS must be able to generate counterarguments as it reads the input text or questions. In order to account for this process of subjective comprehension, FANSYS must have its own ideology, as well as strategies for determining inconsistencies between beliefs in long-term memory and input text, and strategies for selecting counterarguments. These counterargument strategies should be based on the argument-planning knowledge underlying the taxonomy of argument units proposed by Alvarado (1990).

9.8 Belief Inferences Based on Past Failure Analysis

FANSYS must be able to answer questions about causes and effects of failures, and about procedures for repairing failures. Answers to those questions must be found by accessing indices into FANSYS's long-term memory and then traversing memory links associated with those indices. There are questions, however, that require beliefs to be inferred from beliefs and justifications already existing in memory. For example, consider a hypothetical question of the type: "Would FANSYS agree/disagree with advise seeker A over the use of repair procedure P when dealing with failure F?" Here, FANSYS must generate plausibly held beliefs, given its beliefs and justifications in possibly unrelated failure situations. That is, FANSYS must be able to use memories from a previous failure situation to aid in understanding how a repair procedure might be used in a new failure situation. Getting FANSYS to handle hypothetical questions will require modeling the process of belief inference in relation to models of ideology and long-term memory organization.

10. Conclusions

This technical report has described the design and implementation of FANSYS, a failure-analysis system capable of reading segments of manuals dealing with failures on the data management system (DMS) of NASA's Space Station Freedom, and answering questions regarding such failures. FANSYS includes a case-based parser and a model of memory organization, search, and retrieval that are used to create a library of failure-analysis cases. This library is later accessed by FANSYS when answering questions about failures that may occur in the DMS. A major benefit derived from this work has been the formulation of a computational framework for (1) integrating domain-dependent knowledge with case-based parsing strategies within text comprehension systems, (2) representing and indexing failure-analysis cases, (3) creating generalizations among these cases, and (4) using search and retrieval strategies to answer questions about failure causes and effects, as well as failure detection and correction procedures. This computational framework may provide a foundation for developing sophisticated advice-giving and decision-making systems that help operate and maintain NASA's spacecraft.

The experience of designing and implementing a model of text comprehension and question answering for failure analysis has shed light on some of the basic problems any intelligent computer system must address: knowledge representation, organization, application, and retrieval. The model has helped increase understanding of representational constructs and processing strategies needed to analyze system failures. As such, this research has been a step towards modeling basic components that must be present in natural language processing systems.

References

Alvarado, S. J. (1990). *Understanding Editorial Text: A Computer Model of Argument Comprehension*. Boston, MA: Kluwer Academic Publishers.

Alvarado, S. J. (1992). Argument Comprehension. In S. C. Shapiro (Ed.), *Encyclopedia of Artificial Intelligence* (Second edition). New York: Wiley.

Bartlett, R. Davis, G. Grant, T. Gibson, J. Hedges, R. Johnson, M. Liu, Y. Patterson-Hine, A. Sliwa, N. Sowizral, H. and Yan, J. (1992). "Space Station Freedom Data Management System Growth and Evolution Report," *NASA Technical Memorandum 103869*. Ames Research Center.

Blattner, M. M. and Dannenberg, R. B. (1992). *Multimedia Interface Design*. New York, NY: ACM Press.

Carbonell, J. G. (1981). *Subjective Understanding: Computer Models of Belief Systems.* Ann Arbor, MI: UMI Research Press.

Charniak, E. (1977). "A framed PAINTING: The Representation of a Commonsense Knowledge Fragment," *Cognitive Science*, 1, 355-394.

Charniak, E. (1978). "On the Use of Framed Knowledge in Language Comprehension," *Artificial Intelligence*, 11, 225-265.

de Kleer, J. and Williams, B. (1987). "Diagnosing Multiple Faults," *Artificial Intelligence*. v32 no. 1, 97-130.

Dyer, M. G. (1983). *In-Depth Understanding: A Computer Model of Integrated Processing for Narrative Comprehension.* Cambridge, MA: MIT Press.

Dyer, M. G. and Lehnert, W. G. (1982). Question Answering for Narrative Memory. In J. F. Le Ny and W. Kintsch (Eds.), *Language and Comprehension.* Amsterdam. North-Holland.

Flowers, M., McGuire, R., and Birnbaum, L. (1982). Adversary Arguments and the Logic of Personal Attacks. In W. G. Lehnert and M. G. Ringle (Eds.), *Strategies for Natural Language Processing.* Hillsdale, NJ: Lawrence Erlbaum.

Gentner, D. and Grudin, J. (1990). "Why Good Engineers (Sometimes) Create Bad Interfaces," *Proceedings of CHI 90*, ACM Press, 277-282.

Hammond, K., and Hurwitz, N. (1988). "Extracting Diagnostic Features from Explanations," *Proceedings of the Workshop on Case-Based Reasoning (DARPA).* San Mateo, CA: Morgan-Kaufmann Publishers.

Hendler, J. A. (1988). *Integrating Marker-Passing and Problem-Solving: A Spreading Activation Approach to Improved Choice in Planning.* Hillsdale, NJ: Lawrence Erlbaum.

Hodges, J. (1989). "Device Representation for Modeling Improvisation in Mechanical Use Situations," *Proc. 11th Ann Meeting Cognitive Science Soc.* Hillsdale, NJ: Lawrence Erlbaum.

International Business Machines Corporation (July, 1990). "Space Station Program Data Management System," *89IBMX0367R1*, (DR SA-25.1I), Houston, TX.

Iverson, D. L. and Patterson-Hine, F. A. (1990). Object-Oriented Fault Tree Models Applied to System Diagnosis. *Proceedings of the SPIE Applications of Artificial Intelligence VIII Conference.* Orlando, Florida.

Kolodner, J. (1983a). "Maintaining Organization in a Dynamic Long-Term Memory," *Cognitive Science*, 7, 243-280.

Kolodner, J. (1983b). "Reconstructive Memory: A Computer Model," *Cognitive Science*, 7, 281-328.

Kolodner, J. (1984). *Retrieval and Organizational Strategies in Conceptual Memory: A Computer Model*. Hillsdale, NJ: Lawrence Erlbaum.

Kolodner, J. (Ed.) (1988). *Case-Based Reasoning*. San Mateo, CA: Morgan Kaufmann. Lebowitz, M. (1988). "The use of Memory in Text Processing," *Communications of the ACM*, V31 N12, 48-62.

Lehnert, W. G. (1978). *The Process of Question Answering: A Computer Simulation of Cognition.* Hillsdale, NJ: Lawrence Earlbaum.

Liu, H., Tan, A., Lim, J., & Teh, H. (1991). "Practical Application of a Connectionist Expert System --- The INSIDE Story," *World Congress on Expert Systems*, Orlando, Florida: Pergamon Press, 1-21.

McDonnell Douglas Space Systems Company. (March, 1990). "Failure Modes and Effects Analysis (FMEA) for the Data Management System (DMS)," *MDC H4563A,* (DR SA-06.1), Huntington Beach, CA.

McDonnell Douglas Space Systems Company. (July, 1990). "Integrated Operations Requirements For Distributed Systems," *MDC H4380,* (DR SY-46.1), Volume 1, Huntington Beach, CA.

MacLean, A., Bellotti, V., Young, R. and Moran, T. (1991). "Reaching through Analogy: A Design Rationale Perspective on Roles of Analogy," *Proceedings of CHI 91*, ACM Press, 167-172.

Minksy, M. (1975). "Framework for Representing Knowledge," In P. Winston (Ed.), *Psychology of Computer Vision*. New York: McGraw-Hill.

Minksy, M. (1977). "Frame-System Theory," In P. Johnson and O. Wason (Eds.), *Thinking: Readings in Cognitive Science*. Cambridge, MA: MIT Press.

Myers, B. (1991). "Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs," *Proceedings of UIST '91*, 211-220.

Norvig, P. (1989). "Marker Passing as a Weak Method for Text Inferencing," *Cognitive Science*, 13, 569-620.

Pazzani, M. J. (1988). *Learning Causal Relationships: An Integration of Empirical and Explanation-Based Learning Methods* (Ph.D. Dissertation). Computer Science Department. University of California, Los Angeles.

Patterson-Hine, F. and Iverson, D. (1990). "An Integrated Approach to System Design, Reliability, and Diagnosis," *NASA Technical Memorandum 102861.* Ames Research Center.

Reed, N. & Johnson, P. (1990). "Generative Knowledge for Computer Troubleshooting," *European Conference on Artificial Intelligence '90*, 535-540.

Riesbeck, C. K. and Martin, C. E. (1986). "Direct Memory Access Parsing." In. Kolodner, J. and Riesbeck, R. (Eds.), *Experience, Memory, and Reasoning*. Hillsdale, NJ: Lawrence Erlbaum.

Riesbeck, C. & Schank, R. (1989). *Inside Case-based Reasoning*. Hillsdale, NJ: Lawrence Erlbaum.

Schank, R. C. (Ed.) (1975). *Conceptual Information Processing*. Amsterdam: North-Holland.

Schank, R. C. and Abelson, R. (1977). *Scripts, Plans, Goals, and Understanding*. Hillsdale, NJ: Lawrence Erlbaum.

Schank, R. C. (1978). "What Makes Something 'Ad Hoc.'" *Proceedings of Theoretical Issues in Natural Language Processing-2*. University of Illinois, Urbana-Champaign, pp. 8-13.

Schank, R. C. and Carbonell, J. G. (1979). "Re: The Gettysburg Address, Representing Social and Political Acts." In N. Findler (Ed.), *Associative Networks*. New York: Academic Press.

Schank, R. C. (1982). *Dynamic Memory: A Theory of Reminding and Learning in Computers and People.* Cambridge: Cambridge University Press.

Stevenson, R., Miller, J., and Austin, M. (1991). "Failure Environment Analysis Tool (FEAT) Development Status," *Proceedings of AIAA '91*, 676-682.

Struss, P. (1988). "Extensions to ATMS-based Diagnosis," In Gero (Ed.), *Artificial Intelligence Engineering: Diagnosis and Learning*. Amsterdam: Elsevier, 3-28.

Tulving, E. (1972). "Episodic and semantic memory," in E. Tulving & W. Donaldson (Eds.), *Organization of memory*, New York, NY: Academic Press.

Wilensky, R. (1986). "Knowledge Representation - A Critique and a Proposal." In. Kolodner, J. and Riesbeck, R. (Eds.), *Experience, Memory, and Reasoning*. Hillsdale, NJ: Lawrence Erlbaum.